# Chapter 6: Synchronization

## Operating Systems

Race conditions · Critical-section requirements · Peterson and the memory model · Memory barriers · test_and_set, CAS · Atomic variables · Spinlocks and mutexes · Semaphores · Classic problems · Monitors and condition variables · POSIX and Java APIs · Priority inversion, lock-free

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/os/slides/Chapter_06_Synchronization.html

# Learning Outcomes

### Problem

Explain race conditions, the critical section, and the three requirements a solution must meet: mutual exclusion, progress, bounded waiting.

### Software and the memory model

Evaluate the lock variable, strict alternation, and Peterson's solution; explain how reordering on modern CPUs breaks them and how memory barriers help.

### Hardware support

Build locks and counters with `test_and_set`, `compare_and_swap`, XCHG, and atomic variables.

### Tools

Compare spinlocks, mutexes, semaphores, monitors, and condition variables; apply the Mesa-style `while` rule.

### Classic problems

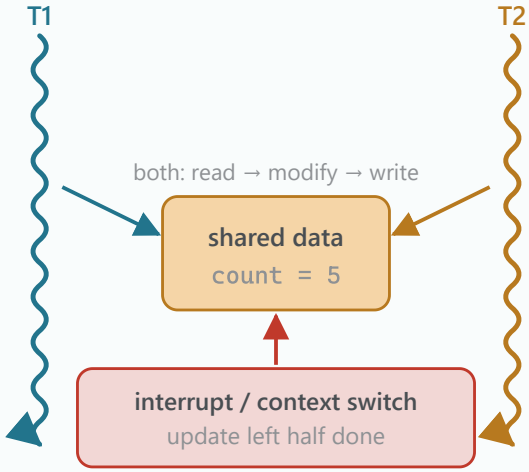
Solve the bounded-buffer, readers–writers, and dining-philosophers problems; recognize the risks of starvation and deadlock.

### Real systems

Use the POSIX and Java APIs; analyze priority inversion (Mars Pathfinder) and the basics of lock-free data structures.

# Why Synchronization?

- **Synchronization:** coordinating how concurrent processes or threads access shared data so that it **stays consistent**.
- A process can be interrupted at **any point** in its instruction stream, leaving an update half done.
- On a multicore system no interrupt is needed: two cores can write the same memory **truly simultaneously**.
- Symptoms:
  - inconsistent data, lost updates, a corrupted linked list;
  - a non-reentrant function's shared buffer getting overwritten;
  - threads waiting on each other forever (**deadlock**);
  - a thread that never gets its turn (**starvation**).



# Race Condition: fork() and pid




Adapted from Silberschatz Figure 6.2

The kernel keeps the next free pid in `next_available_pid`: **2615**. P0 and P1 call `fork()` at the same time.

Step 1 / 4

The same race arises in user programs when `count++` interleaves at the register level.

# Race Condition: fork() and pid



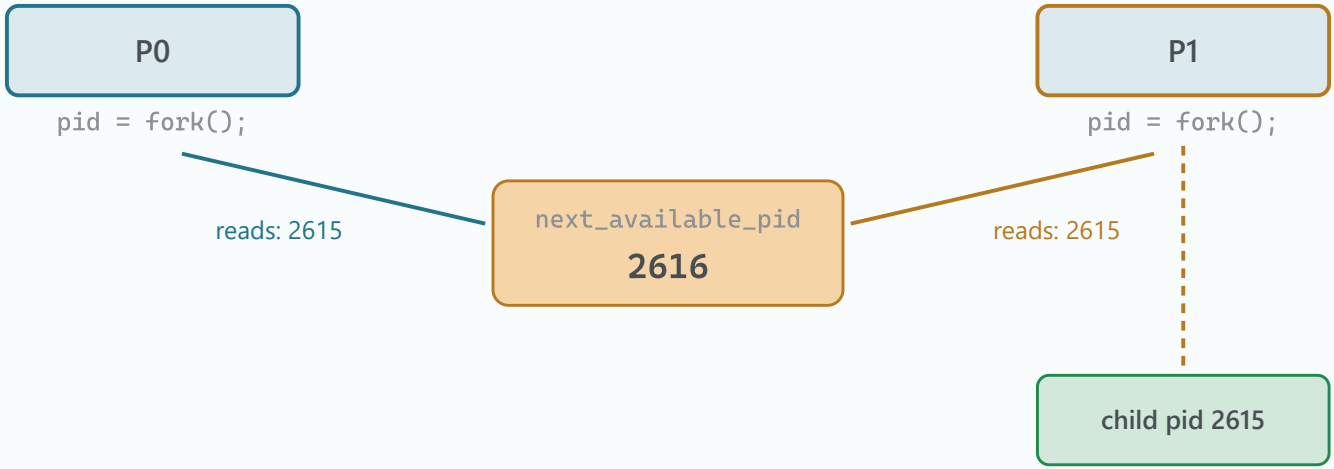

Adapted from Silberschatz Figure 6.2

P0's kernel code reads the value (2615) but is interrupted before it can increment and store it.



The same race arises in user programs when `count++` interleaves at the register level.

# Race Condition: fork() and pid




Adapted from Silberschatz Figure 6.2

P1 also reads 2615, gives it to its child, and writes 2616.



The same race arises in user programs when `count++` interleaves at the register level.

# Race Condition: fork() and pid



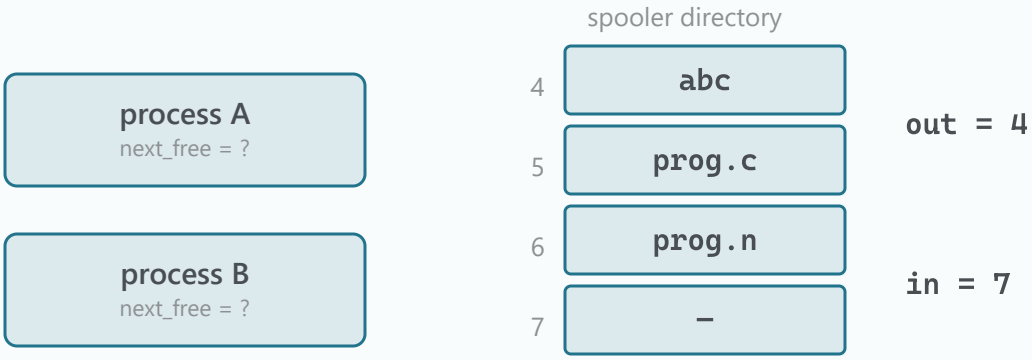

Adapted from Silberschatz Figure 6.2

P0 resumes and gives its child **2615 as well**. The outcome depends on the order of execution: a **race condition**.



The same race arises in user programs when `count++` interleaves at the register level.

# Race Condition: Printer Spooler Directory

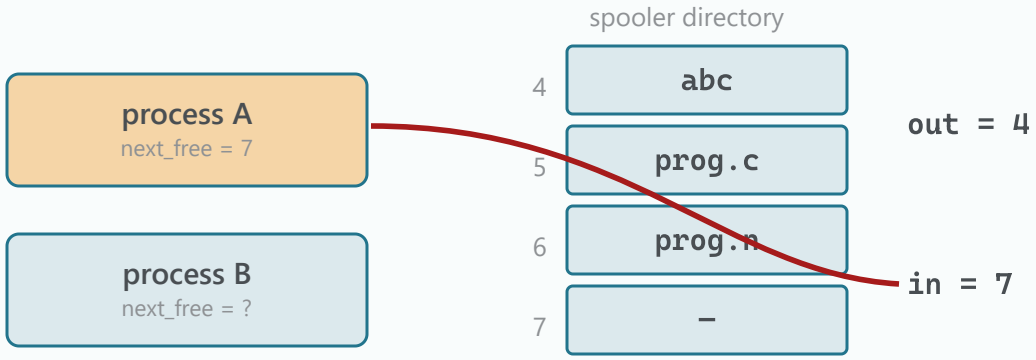



Files to print go into slots in order. `out` is the next file to print, `in` the first free slot.

Step 1 / 6

**Race condition:** two or more processes access shared data and the final result depends on the order of execution.

# Race Condition: Printer Spooler Directory

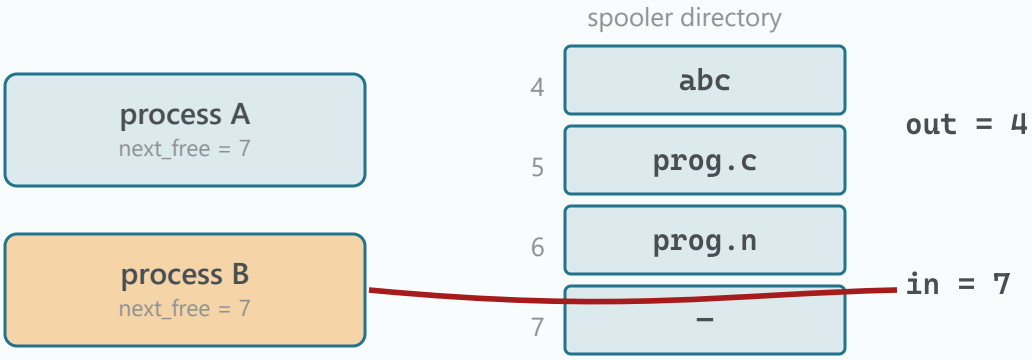



A reads `in` (7) into its local `next_free`; right then its time slice expires.



**Race condition:** two or more processes access shared data and the final result depends on the order of execution.

# Race Condition: Printer Spooler Directory

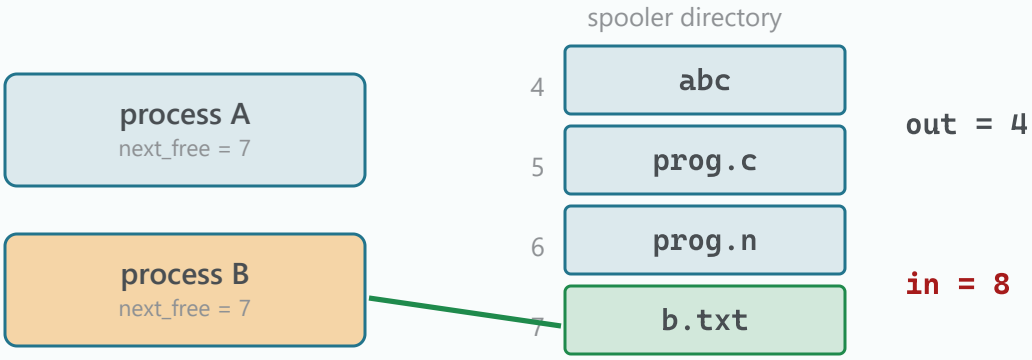



B also reads `in` : 7 again!

Step 3 / 6

**Race condition:** two or more processes access shared data and the final result depends on the order of execution.

# Race Condition: Printer Spooler Directory

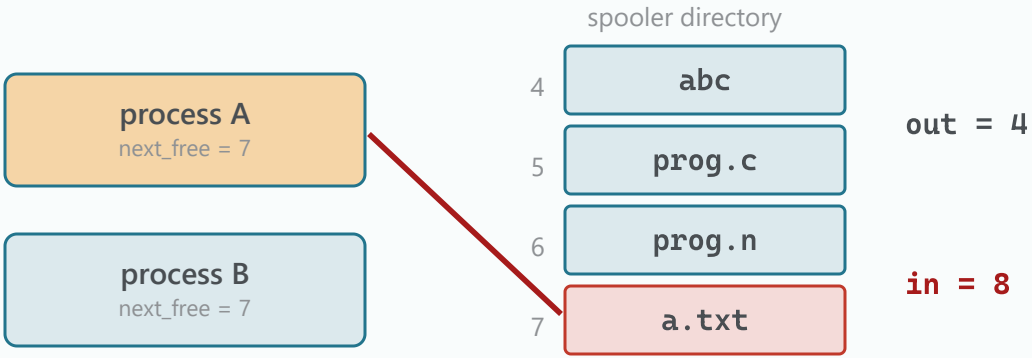



B writes its file name into slot 7 and sets `in = 8`.



**Race condition:** two or more processes access shared data and the final result depends on the order of execution.

# Race Condition: Printer Spooler Directory

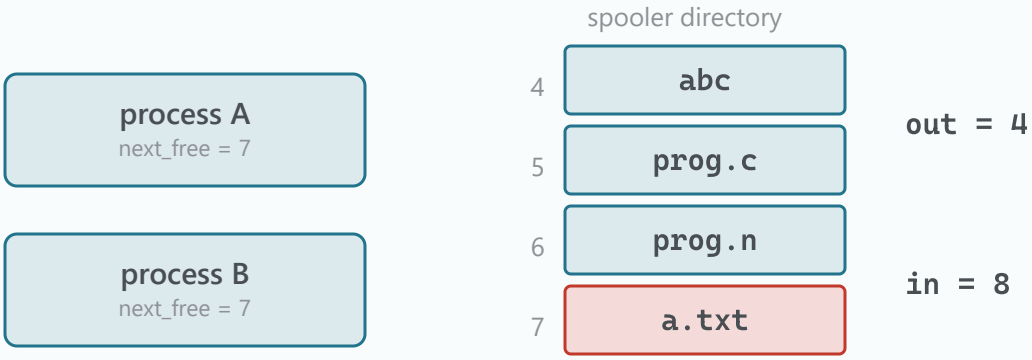



A resumes: since `next_free = 7`, it writes slot 7, **overwriting** B's entry; `in = 8`.



**Race condition:** two or more processes access shared data and the final result depends on the order of execution.

# Race Condition: Printer Spooler Directory




**B's file is never printed!**

The directory looks consistent, but B's output is gone. The result depends on who runs when: a **race condition**.



**Race condition:** two or more processes access shared data and the final result depends on the order of execution.

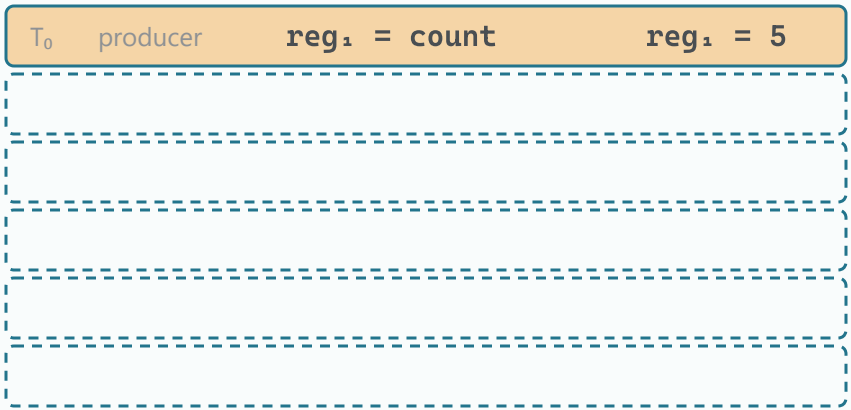
# Non-reproducible Execution

```
/* Process 1 */ /* Process 2 */
repeat          repeat
  n = n + 1;      print(n);
                  n = 0;
```

Possible orders with n = 5:

- `n=n+1; print; n=0` → prints 6
- `print; n=0; n=n+1` → prints 5, n=1
- `print; n=n+1; n=0` → prints 5, increment lost

Even one line of C ( `count++` ) is three machine instructions:




Initially `count = 5` . For `count++` the producer first loads the value into a register.

Step 1 / 6

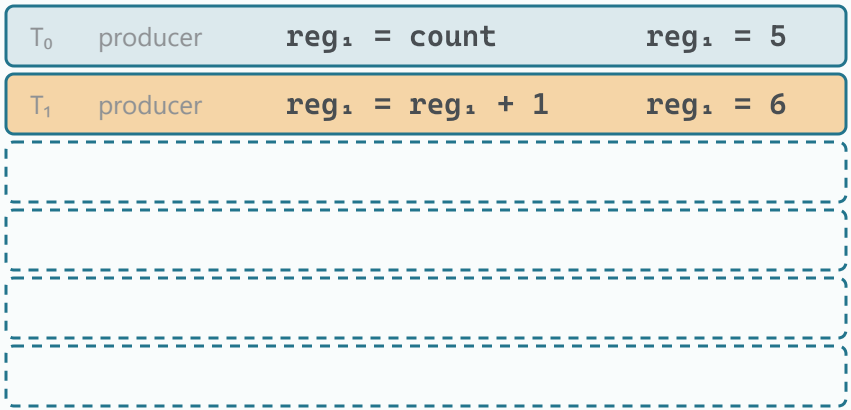
# Non-reproducible Execution

```
/* Process 1 */ /* Process 2 */
repeat          repeat
  n = n + 1;      print(n);
                  n = 0;
```

Possible orders with n = 5:

- `n=n+1; print; n=0` → prints 6
- `print; n=0; n=n+1` → prints 5, n=1
- `print; n=n+1; n=0` → prints 5, increment lost

Even one line of C ( `count++` ) is three machine instructions:




The producer increments its register to 6, then is interrupted before storing it.

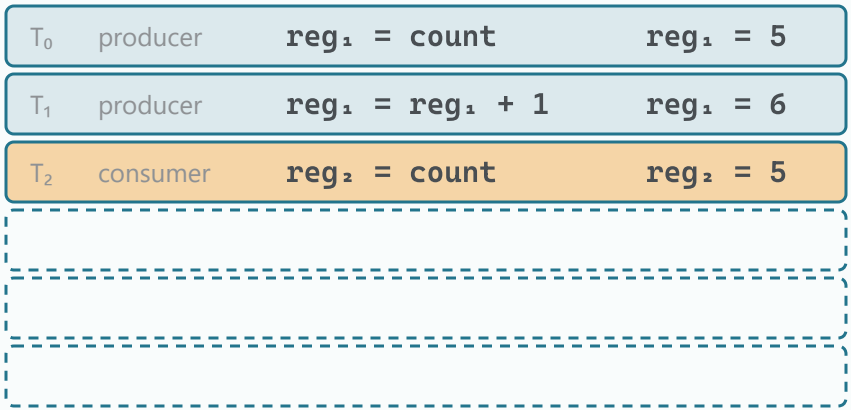
# Non-reproducible Execution

```
/* Process 1 */ /* Process 2 */
repeat          repeat
  n = n + 1;      print(n);
                  n = 0;
```

Possible orders with n = 5:

- `n=n+1; print; n=0` → prints 6
- `print; n=0; n=n+1` → prints 5, n=1
- `print; n=n+1; n=0` → prints 5, increment lost

Even one line of C ( `count++` ) is three machine instructions:




The consumer runs `count--` : memory still holds 5.

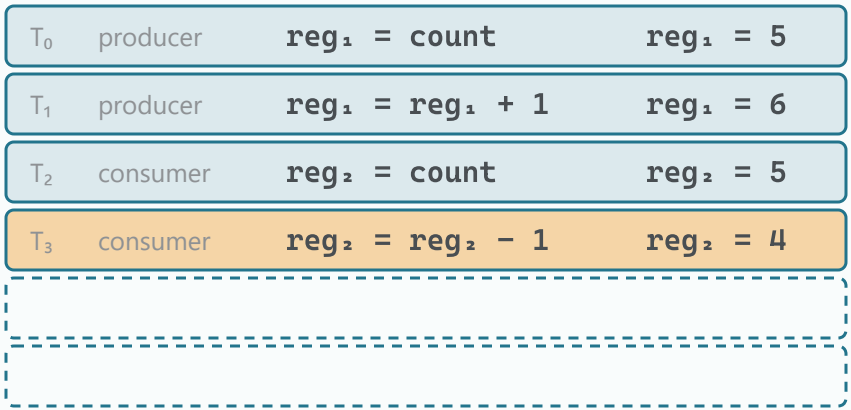
# Non-reproducible Execution

```
/* Process 1 */ /* Process 2 */
repeat          repeat
  n = n + 1;      print(n);
                  n = 0;
```

Possible orders with n = 5:

- `n=n+1; print; n=0` → prints 6
- `print; n=0; n=n+1` → prints 5, n=1
- `print; n=n+1; n=0` → prints 5, increment lost

Even one line of C ( `count++` ) is three machine instructions:




The consumer decrements its register: 4.

# Non-reproducible Execution

```
/* Process 1 */ /* Process 2 */
repeat          repeat
  n = n + 1;      print(n);
                  n = 0;
```

Possible orders with n = 5:

- `n=n+1; print; n=0` → prints 6
- `print; n=0; n=n+1` → prints 5, n=1
- `print; n=n+1; n=0` → prints 5, increment lost

Even one line of C ( `count++` ) is three machine instructions:

| | | | |
|---|---|---|---|
| $T_0$ | producer | $reg_1$ = count | $reg_1$ = 5 |
| $T_1$ | producer | $reg_1$ = $reg_1$ + 1 | $reg_1$ = 6 |
| $T_2$ | consumer | $reg_2$ = count | $reg_2$ = 5 |
| $T_3$ | consumer | $reg_2$ = $reg_2$ − 1 | $reg_2$ = 4 |
| $T_4$ | producer | count = $reg_1$ | count = 6 |

The producer stores: `count = 6` .

# Non-reproducible Execution

```
/* Process 1 */ /* Process 2 */
repeat          repeat
  n = n + 1;      print(n);
                  n = 0;
```

Possible orders with n = 5:

- `n=n+1; print; n=0` → prints 6
- `print; n=0; n=n+1` → prints 5, n=1
- `print; n=n+1; n=0` → prints 5, increment lost

Even one line of C ( `count++` ) is three machine instructions:

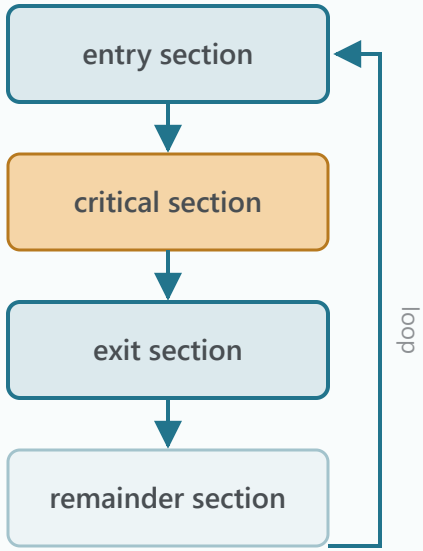
| | | | |
|---|---|---|---|
| $T_0$ | producer | $reg_1$ = count | $reg_1$ = 5 |
| $T_1$ | producer | $reg_1$ = $reg_1$ + 1 | $reg_1$ = 6 |
| $T_2$ | consumer | $reg_2$ = count | $reg_2$ = 5 |
| $T_3$ | consumer | $reg_2$ = $reg_2$ − 1 | $reg_2$ = 4 |
| $T_4$ | producer | count = $reg_1$ | count = 6 |
| $T_5$ | consumer | count = $reg_2$ | count = 4 |

The consumer overwrites it: `count = 4` . The right answer was 5; another order could give 6.

# The Critical-Section Problem

- n processes `{P0, P1, …, Pn−1}`; each has a **critical section** in which it accesses shared data: writing a global, updating a table, writing a file…
- **Rule:** while one process is in its critical section, no other process may be in *its* critical section.
- Each process requests entry in its **entry section** and announces its exit in its **exit section**.
- **Race condition:** the result depends on execution order. **Mutual exclusion:** only one access to shared data at a time.

```
while (true) {
    /* entry section             */
        /* critical section      */
    /* exit section              */
        /* remainder section       */
}
```




request → use → release

# Three Requirements for a Solution

**1. Mutual exclusion**

While $P_i$ is in its critical section, no other process may be in its own.

**2. Progress**

If no process is in its critical section, only those *not in their remainder sections* decide who enters next, and the choice cannot be postponed indefinitely.

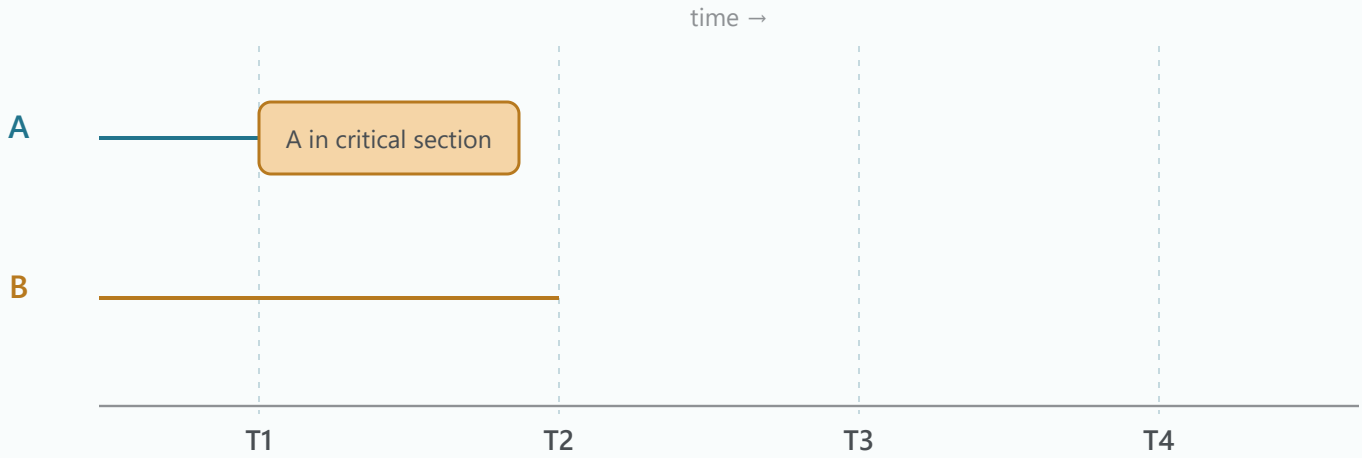
**3. Bounded waiting**

Once a process asks to enter, others may enter only a **bounded** number of times before it does (no starvation).

| Tanenbaum's four conditions | Equivalent |
|---|---|
| No two processes may be in their critical sections at once. | Mutual exclusion |
| No assumptions about CPU speed or count. | Nonzero speed; relative speeds unknown |
| No process outside its critical section may block others. | Progress |
| No process should wait forever. | Bounded waiting |

Silberschatz 10e §6.2; Tanenbaum MOS §2.3.2. Exclusion is a *safety* property; the other two are *liveness* properties.

# Critical-Section Timeline

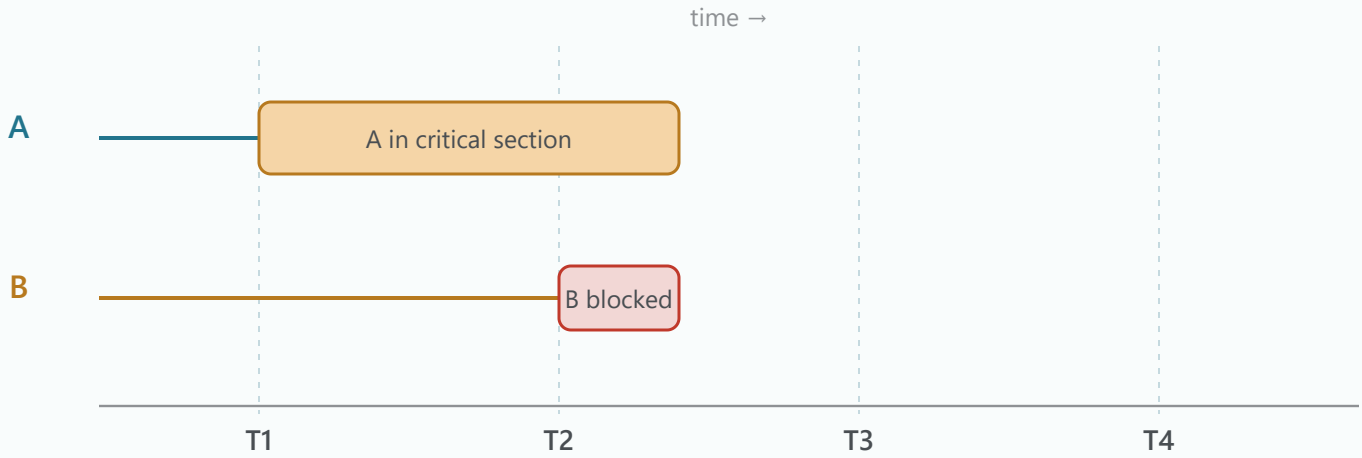



**T1:** A enters its critical section.

Step 1 / 4

Adapted from Tanenbaum MOS Figure 2-22.

# Critical-Section Timeline

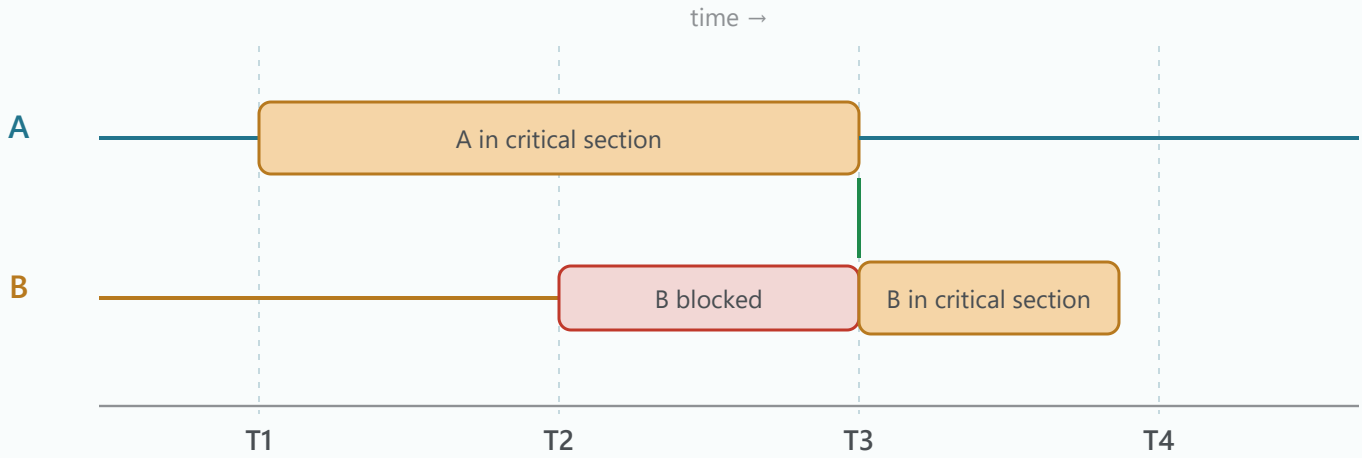



**T2:** B tries to enter; the section is taken, so B **blocks** (or busy-waits).



Adapted from Tanenbaum MOS Figure 2-22.

# Critical-Section Timeline

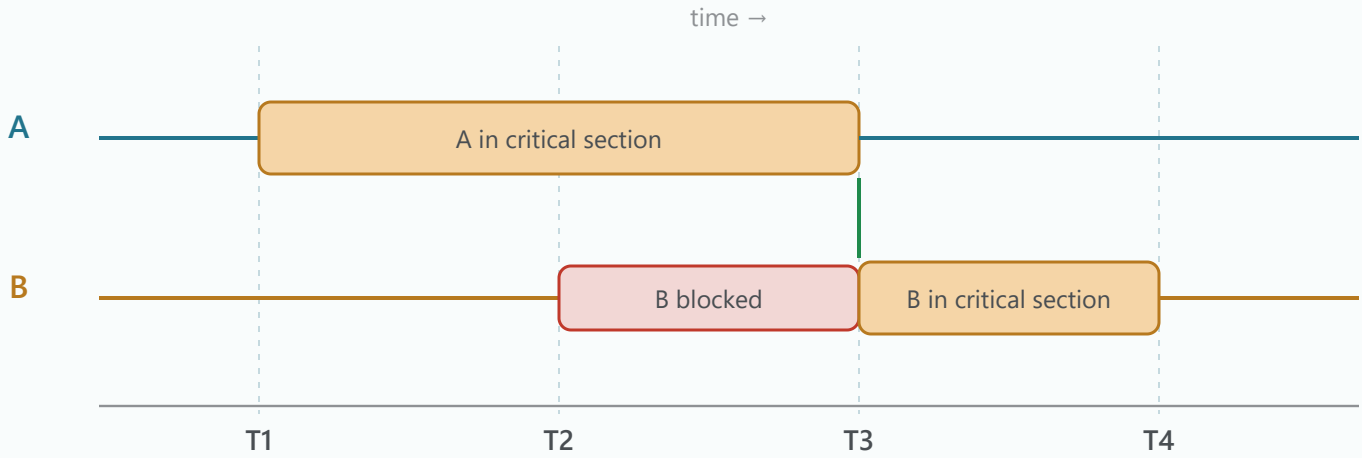



**T3:** A leaves; the waiting B enters at once. At no moment are two processes inside.



Adapted from Tanenbaum MOS Figure 2-22.

# Critical-Section Timeline



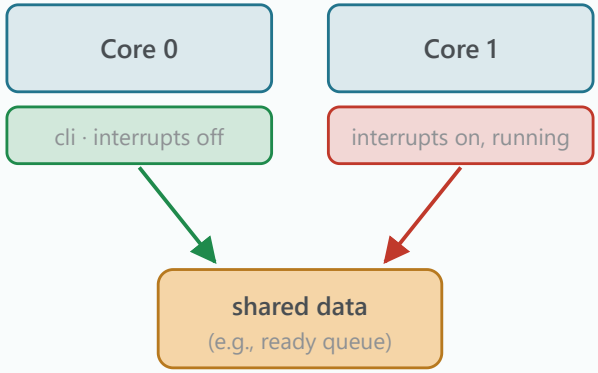

**T4:** B leaves. A correct solution must guarantee this *under any schedule*.



Adapted from Tanenbaum MOS Figure 2-22.

# Disabling Interrupts

- A process disables interrupts before entering its critical section and re-enables them on exit: with no timer interrupt, **no context switch can occur**.
- **Problems:**
  - A process that forgets to re-enable interrupts (or loops forever) **hangs the system**.
  - I/O and timer interrupts wait meanwhile; a long critical section makes the system unresponsive.
  - **Useless on multicore:** the other cores keep running.
  - `cli` / `sti` are privileged instructions; user programs cannot execute them.
- Inside the kernel, **brief** local use is common: Linux `local_irq_save()`, and `spin_lock_irqsave()` for data shared with an interrupt handler.




disabling interrupts stops only the local core

# Lock Variable

```
1  int lock = 0;            /* shared */
2
3  void enter(void) {
4    while (lock != 0)
5      ;                    /* busy wait */
6    lock = 1;
7  }
8  void leave(void) {
9    lock = 0;
10 }
11
12 /* each process: enter(); critical(); leave(); */
```

P0 — outside critical section

P1 — outside critical section

Shared `lock` is 0: the section is free.

Step 1 / 6

**VARIABLES**

`lock = 0`

The lock itself is a shared variable, and protecting it would need yet another lock. A software lock alone is not enough.

# Lock Variable

```
1  int lock = 0;            /* shared */
2
3  void enter(void) {
▶    while (lock != 0)
5      ;                    /* busy wait */
6    lock = 1;
7  }
8  void leave(void) {
9    lock = 0;
10 }
11
12 /* each process: enter(); critical(); leave(); */
```

**P0** lock != 0 ? → no

**P1** ready

P0 reads `lock` : 0. It exits the loop...

Step 2 / 6

**VARIABLES**

`lock = 0`

The lock itself is a shared variable, and protecting it would need yet another lock. A software lock alone is not enough.

# Lock Variable

```
1  int lock = 0;            /* shared */
2
3  void enter(void) {
▶    while (lock != 0)
5      ;                    /* busy wait */
6    lock = 1;
7  }
8  void leave(void) {
9    lock = 0;
10 }
11
12 /* each process: enter(); critical(); leave(); */
```

**P0** interrupted (before line 6)

**P1** lock != 0 ? → no

...but is interrupted before writing `lock = 1`. P1 also reads 0.

Step 3 / 6

**VARIABLES**

`lock = 0`

The lock itself is a shared variable, and protecting it would need yet another lock. A software lock alone is not enough.

# Lock Variable

```
1  int lock = 0;            /* shared */
2
3  void enter(void) {
4    while (lock != 0)
5      ;                    /* busy wait */
▶    lock = 1;
7  }
8  void leave(void) {
9    lock = 0;
10 }
11
12 /* each process: enter(); critical(); leave(); */
```

P0 — ready

P1 — lock = 1 → entered

P1 sets the lock to 1 and enters its critical section.

Step 4 / 6

**VARIABLES**

`lock = 1`

The lock itself is a shared variable, and protecting it would need yet another lock. A software lock alone is not enough.

# Lock Variable

```
1  int lock = 0;            /* shared */
2
3  void enter(void) {
4    while (lock != 0)
5      ;                    /* busy wait */
▶    lock = 1;
7  }
8  void leave(void) {
9    lock = 0;
10 }
11
12 /* each process: enter(); critical(); leave(); */
```

**P0** lock = 1 → entered

**P1** in critical section

P0 resumes; it **already tested** the lock, so it also writes 1 and enters.

Step 5 / 6

**VARIABLES**

`lock = 1`

The lock itself is a shared variable, and protecting it would need yet another lock. A software lock alone is not enough.

# Lock Variable

```
1  int lock = 0;            /* shared */
2
3  void enter(void) {
4    while (lock != 0)
5      ;                    /* busy wait */
▶    lock = 1;
7  }
8  void leave(void) {
9    lock = 0;
10 }
11
12 /* each process: enter(); critical(); leave(); */
```

P0 — in critical section

P1 — in critical section

two processes in the critical section!

Mutual exclusion is broken: **test** and **set** are separate steps. The fix is a single atomic instruction for both (TSL).

Step 6 / 6

**VARIABLES**

`lock = 1`

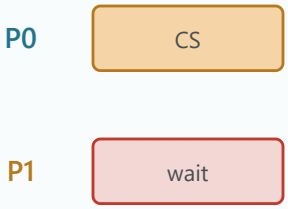
The lock itself is a shared variable, and protecting it would need yet another lock. A software lock alone is not enough.

# Strict Alternation

```
/* P0 */
while (true) {
  while (turn != 0) ;  /* wait */
  critical_section();
  turn = 1;
  noncritical();
}
```

```
/* P1 */
while (true) {
  while (turn != 1) ;  /* wait */
  critical_section();
  turn = 0;
  noncritical();
}
```

`turn = 0`



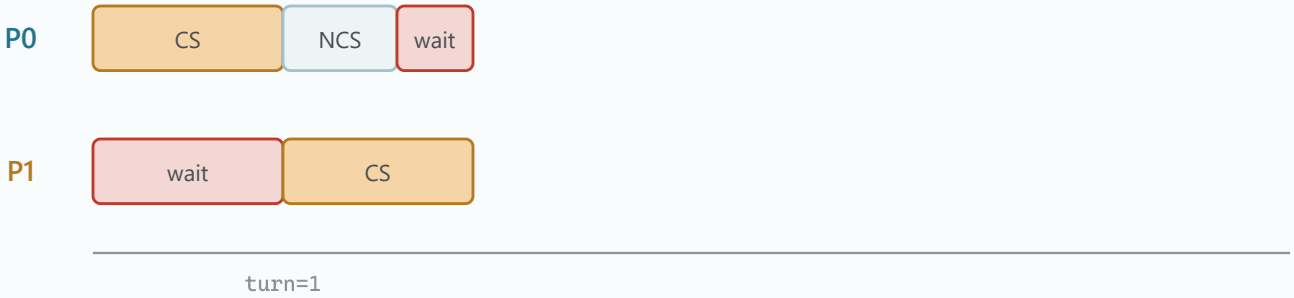

`turn = 0` : P0 enters, P1 busy-waits (**CS** critical section, **NCS** noncritical section).

# Strict Alternation

```
/* P0 */
while (true) {
  while (turn != 0) ;  /* wait */
  critical_section();
  turn = 1;
  noncritical();
}
```

```
/* P1 */
while (true) {
  while (turn != 1) ;  /* wait */
  critical_section();
  turn = 0;
  noncritical();
}
```

`turn = 1`

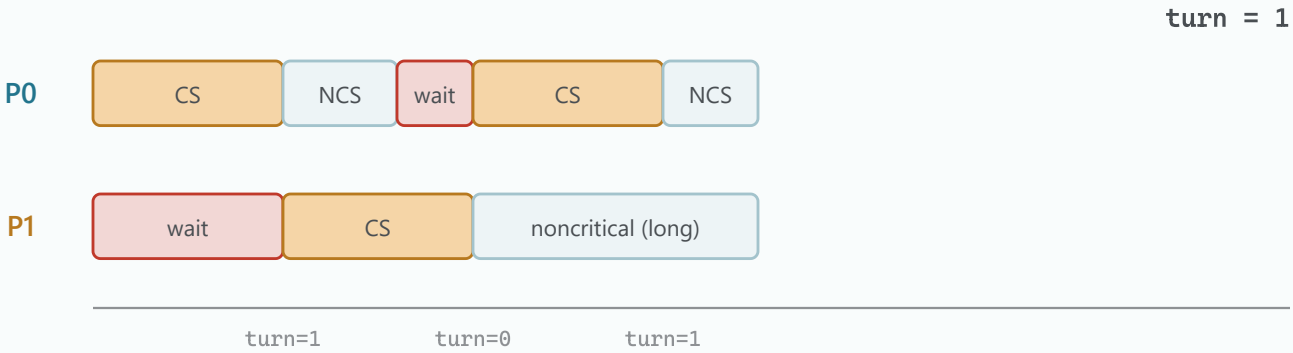



On exit P0 sets `turn = 1`; P1 enters. Mutual exclusion holds.

# Strict Alternation

```
/* P0 */
while (true) {
  while (turn != 0) ;  /* wait */
  critical_section();
  turn = 1;
  noncritical();
}
```

```
/* P1 */
while (true) {
  while (turn != 1) ;  /* wait */
  critical_section();
  turn = 0;
  noncritical();
}
```

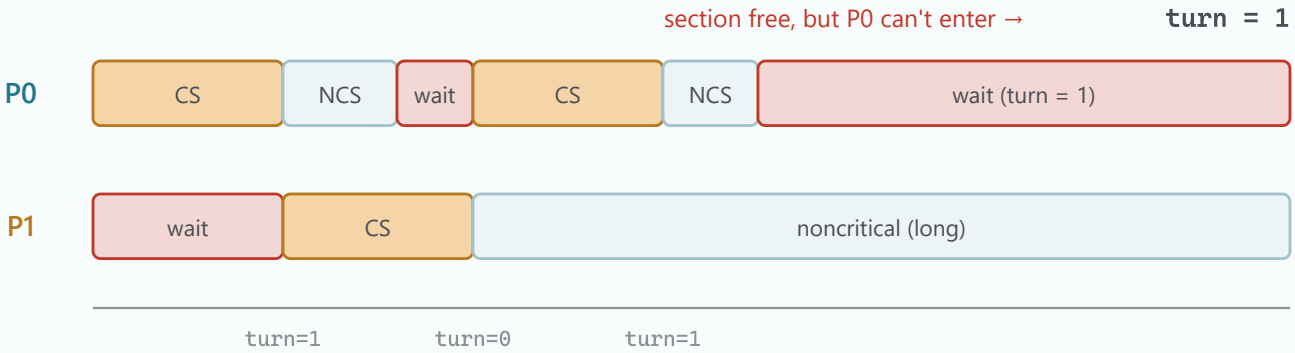



P1 leaves ( `turn = 0` ) and starts a long noncritical stretch. P0 enters and leaves once more: `turn = 1` .

# Strict Alternation

```
/* P0 */
while (true) {
  while (turn != 0) ;  /* wait */
  critical_section();
  turn = 1;
  noncritical();
}
```

```
/* P1 */
while (true) {
  while (turn != 1) ;  /* wait */
  critical_section();
  turn = 0;
  noncritical();
}
```




P0 wants in again, but it is P1's turn. The section is free, yet **P1, outside its critical section, blocks P0**: **progress** is violated.

SOFTWARE APPROACHES

# Peterson's Solution

```
/* shared */
bool flag[2] = { false, false };
int  turn;

/* P_i  (j = 1 - i) */
while (true) {
  flag[i] = true;     /* I want in      */
  turn = j;           /* after you      */
  while (flag[j] && turn == j)
    ;                 /* busy wait      */
  /* critical section */
  flag[i] = false;
  /* remainder section */
}
```

G. L. Peterson, 1981. Works for two processes only; for n processes there is the bakery algorithm.

### Mutual exclusion

If both were inside, `flag[0] = flag[1] = true`; but `turn` cannot be 0 and 1 at once. Whoever wrote `turn` last waits.

### Progress

If $P_j$ is not interested, `flag[j] = false`: $P_i$ enters without waiting.

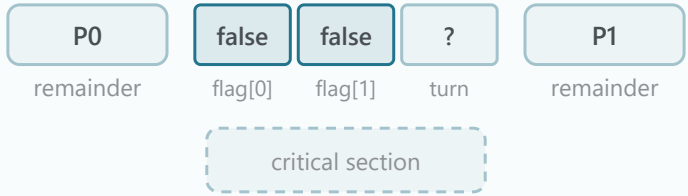
### Bounded waiting

If $P_j$ leaves and wants back in, it sets `turn = i`; $P_i$ waits for at most **one** entry.

The cost: **busy waiting** burns CPU. Worse, the proof assumes *sequential consistency*, which modern hardware does not provide →

# Peterson's Solution: Code Trace

```
▶ bool flag[2] = { false, false };
▶ int turn;
3  void enter(int i) {
4    int j = 1 - i;
5    flag[i] = true;    /* I want in */
6    turn = j;          /* after you */
7    while (flag[j] && turn == j)
8      ;                /* busy wait */
9  }
10 void leave(int i) {
11   flag[i] = false;
12 }
```




Both processes are in their remainder sections; both flags are `false`. Memory is assumed sequentially consistent (all operations fit one global order that preserves each thread's program order).

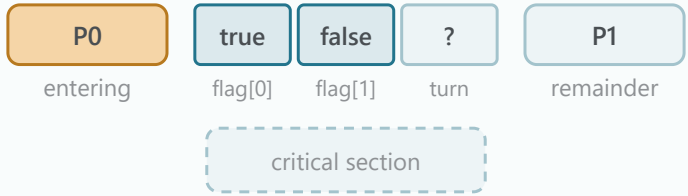
Step 1 / 13

**VARIABLES**

`flag[0] = false` `flag[1] = false` `turn = ?` `P0 = remainder` `P1 = remainder`

# Peterson's Solution: Code Trace

```
1  bool flag[2] = { false, false };
2  int turn;
3  void enter(int i) {
4    int j = 1 - i;
▶    flag[i] = true;    /* I want in */
6    turn = j;          /* after you */
7    while (flag[j] && turn == j)
8      ;                /* busy wait */
9  }
10 void leave(int i) {
11   flag[i] = false;
12 }
```




P0 wants in: `flag[0] = true`.

Step 2 / 13

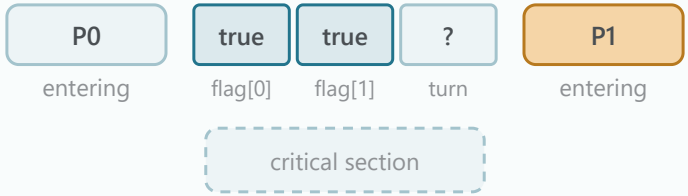
**VARIABLES**

`flag[0] = true` `flag[1] = false`

`turn = ?` `P0 = entering` `P1 = remainder`

# Peterson's Solution: Code Trace

```
1  bool flag[2] = { false, false };
2  int turn;
3  void enter(int i) {
4    int j = 1 - i;
▶    flag[i] = true;    /* I want in */
6    turn = j;          /* after you */
7    while (flag[j] && turn == j)
8      ;                /* busy wait */
9  }
10 void leave(int i) {
11   flag[i] = false;
12 }
```




P1 asks at almost the same moment: `flag[1] = true`.

Step 3 / 13

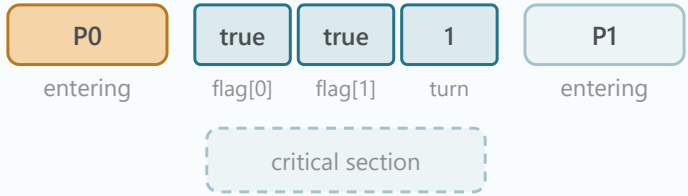
**VARIABLES**

`flag[0] = true` `flag[1] = true`

`turn = ?` `P0 = entering` `P1 = entering`

# Peterson's Solution: Code Trace

```
1  bool flag[2] = { false, false };
2  int turn;
3  void enter(int i) {
4    int j = 1 - i;
5    flag[i] = true;    /* I want in */
▶    turn = j;          /* after you */
7    while (flag[j] && turn == j)
8      ;                /* busy wait */
9  }
10 void leave(int i) {
11   flag[i] = false;
12 }
```




P0 is polite: `turn = 1` ("after you").

Step 4 / 13

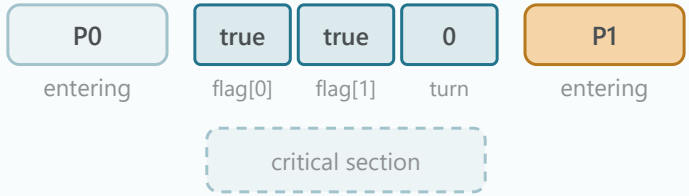
**VARIABLES**

`flag[0] = true` `flag[1] = true`

`turn = 1` `P0 = entering` `P1 = entering`

# Peterson's Solution: Code Trace

```
1  bool flag[2] = { false, false };
2  int turn;
3  void enter(int i) {
4    int j = 1 - i;
5    flag[i] = true;    /* I want in */
▶    turn = j;          /* after you */
7    while (flag[j] && turn == j)
8      ;                /* busy wait */
9  }
10 void leave(int i) {
11   flag[i] = false;
12 }
```




P1 writes `turn = 0` too. **P1 wrote `turn` last**, so P1 is the one that waits.

Step 5 / 13

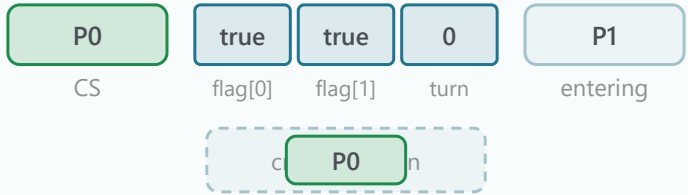
**VARIABLES**

`flag[0] = true` `flag[1] = true`

`turn = 0` `P0 = entering` `P1 = entering`

# Peterson's Solution: Code Trace

```
1  bool flag[2] = { false, false };
2  int turn;
3  void enter(int i) {
4    int j = 1 - i;
5    flag[i] = true;    /* I want in */
6    turn = j;          /* after you */
▶    while (flag[j] && turn == j)
8      ;                /* busy wait */
9  }
10 void leave(int i) {
11   flag[i] = false;
12 }
```




P0: `flag[1] && turn == 1` is false since turn is 0; it exits the loop and **enters its critical section**.

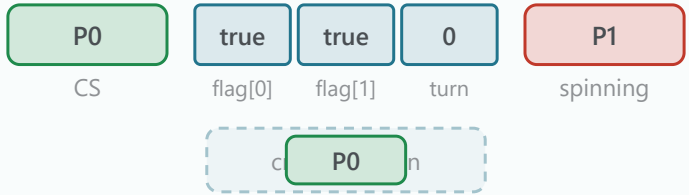
Step 6 / 13

VARIABLES

`flag[0] = true` `flag[1] = true` `turn = 0` `P0 = CS` `P1 = entering`

# Peterson's Solution: Code Trace

```
1  bool flag[2] = { false, false };
2  int turn;
3  void enter(int i) {
4    int j = 1 - i;
5    flag[i] = true;    /* I want in */
6    turn = j;          /* after you */
▶    while (flag[j] && turn == j)
▶      ;                /* busy wait */
9  }
10 void leave(int i) {
11   flag[i] = false;
12 }
```




P1: `flag[0] && turn == 0` is true → busy-waits. Mutual exclusion holds.

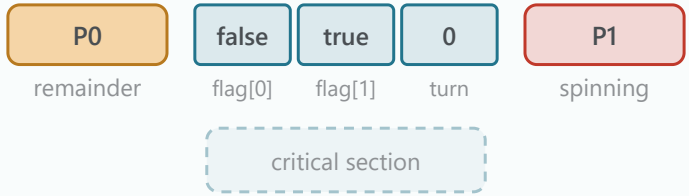
Step 7 / 13

**VARIABLES**

`flag[0] = true` `flag[1] = true` `turn = 0` `P0 = CS` `P1 = spinning`

# Peterson's Solution: Code Trace

```
1  bool flag[2] = { false, false };
2  int turn;
3  void enter(int i) {
4    int j = 1 - i;
5    flag[i] = true;    /* I want in */
6    turn = j;          /* after you */
7    while (flag[j] && turn == j)
8      ;                /* busy wait */
9  }
10 void leave(int i) {
▶    flag[i] = false;
12 }
```




P0 leaves: `flag[0] = false`.

Step 8 / 13

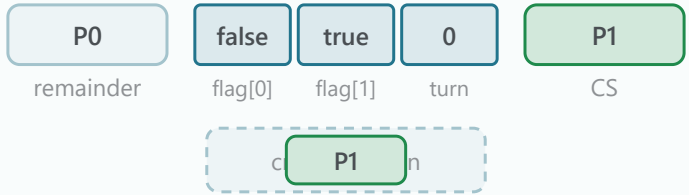
**VARIABLES**

`flag[0] = false` `flag[1] = true`

`turn = 0` `P0 = remainder` `P1 = spinning`

# Peterson's Solution: Code Trace

```
1  bool flag[2] = { false, false };
2  int turn;
3  void enter(int i) {
4    int j = 1 - i;
5    flag[i] = true;    /* I want in */
6    turn = j;          /* after you */
▶    while (flag[j] && turn == j)
8      ;                /* busy wait */
9  }
10 void leave(int i) {
11   flag[i] = false;
12 }
```




P1's condition is now false ( `flag[0]` is false) → P1 enters. Progress holds.

Step 9 / 13

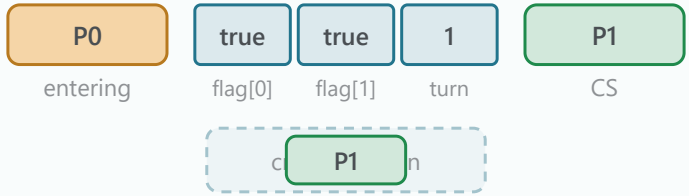
**VARIABLES**

`flag[0] = false` `flag[1] = true`

`turn = 0` `P0 = remainder` `P1 = CS`

# Peterson's Solution: Code Trace

```
1  bool flag[2] = { false, false };
2  int turn;
3  void enter(int i) {
4    int j = 1 - i;
▶    flag[i] = true;    /* I want in */
▶    turn = j;          /* after you */
7    while (flag[j] && turn == j)
8      ;                /* busy wait */
9  }
10 void leave(int i) {
11   flag[i] = false;
12 }
```




P0 asks again: `flag[0] = true`, `turn = 1`.

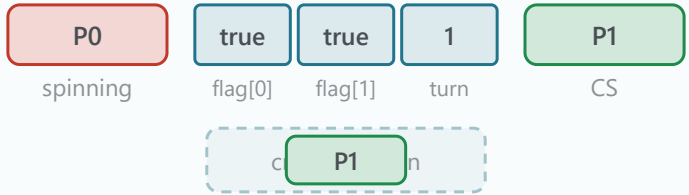
Step 10 / 13

**VARIABLES**

`flag[0] = true` `flag[1] = true` `turn = 1` `P0 = entering` `P1 = CS`

# Peterson's Solution: Code Trace

```
1  bool flag[2] = { false, false };
2  int turn;
3  void enter(int i) {
4    int j = 1 - i;
5    flag[i] = true;    /* I want in */
6    turn = j;          /* after you */
▶    while (flag[j] && turn == j)
▶      ;                /* busy wait */
9  }
10 void leave(int i) {
11   flag[i] = false;
12 }
```




P0: `flag[1] && turn == 1` is true → waits. Now P0 wrote turn last.

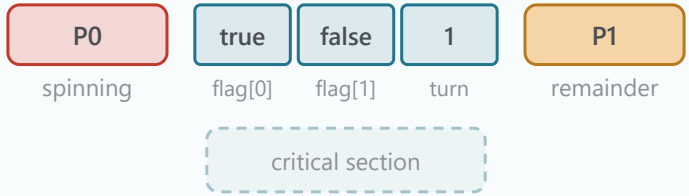
Step 11 / 13

**VARIABLES**

`flag[0] = true` `flag[1] = true` `turn = 1` `P0 = spinning` `P1 = CS`

# Peterson's Solution: Code Trace

```
1  bool flag[2] = { false, false };
2  int turn;
3  void enter(int i) {
4    int j = 1 - i;
5    flag[i] = true;    /* I want in */
6    turn = j;          /* after you */
7    while (flag[j] && turn == j)
8      ;                /* busy wait */
9  }
10 void leave(int i) {
▶    flag[i] = false;
12 }
```




P1 leaves: `flag[1] = false`.

Step 12 / 13

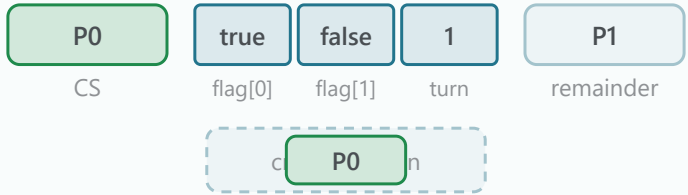
**VARIABLES**

`flag[0] = true` `flag[1] = false`

`turn = 1` `P0 = spinning` `P1 = remainder`

# Peterson's Solution: Code Trace

```
1  bool flag[2] = { false, false };
2  int turn;
3  void enter(int i) {
4    int j = 1 - i;
5    flag[i] = true;    /* I want in */
6    turn = j;          /* after you */
▶    while (flag[j] && turn == j)
8      ;                /* busy wait */
9  }
10 void leave(int i) {
11   flag[i] = false;
12 }
```




P0's condition turns false and it enters. Had P1 asked again, it would set `turn = 0` and wait: P0 waits for at most **one** entry.

Step 13 / 13

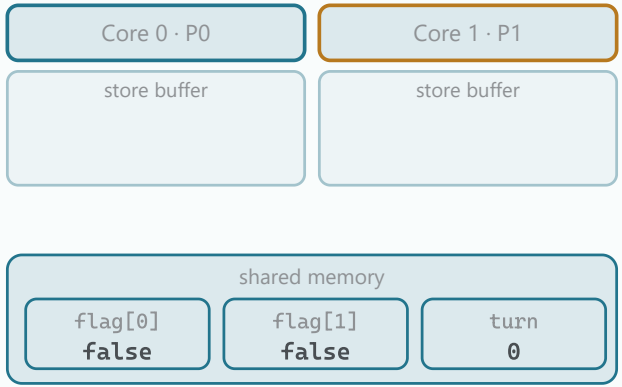
**VARIABLES**

`flag[0] = true` `flag[1] = false`

`turn = 1` `P0 = CS` `P1 = remainder`

# Why Peterson Fails on a Modern CPU

```
1 /* P_i  (i = 0 or 1, j = 1 - i) */
2 flag[i] = true;
3 turn = j;
4 while (flag[j] && turn == j)
5   ;
6 /* critical section */
7 flag[i] = false;
```




Initially both flags in memory are `false`. Each core's stores go first to its own **store buffer**.

Step 1 / 6

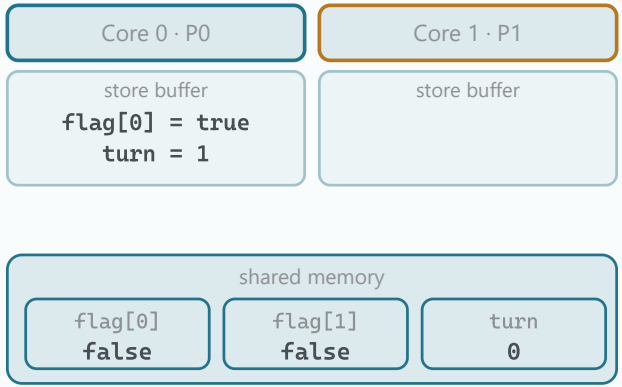
**VARIABLES**

—

The compiler may also hoist that load. C11 `_Atomic` with the default `memory_order_seq_cst`, or Java `volatile` fields, restore the order.

# Why Peterson Fails on a Modern CPU

```
1 /* P_i  (i = 0 or 1, j = 1 - i) */
▶ flag[i] = true;
▶ turn = j;
4 while (flag[j] && turn == j)
5   ;
6 /* critical section */
7 flag[i] = false;
```




P0 performs both stores; they **sit in the buffer**, not yet visible in memory.

Step 2 / 6

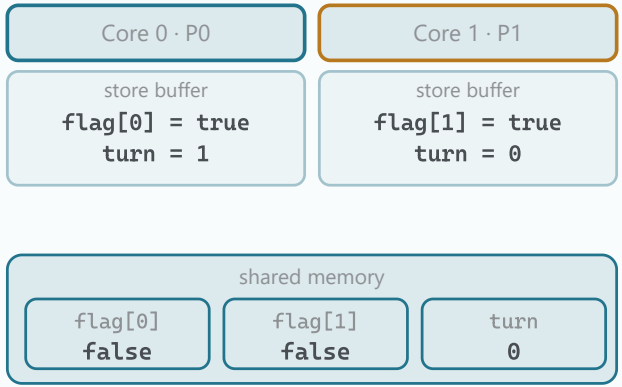
**VARIABLES**

—

The compiler may also hoist that load. C11 `_Atomic` with the default `memory_order_seq_cst`, or Java `volatile` fields, restore the order.

# Why Peterson Fails on a Modern CPU

```
1 /* P_i  (i = 0 or 1, j = 1 - i) */
▶ flag[i] = true;
▶ turn = j;
4 while (flag[j] && turn == j)
5   ;
6 /* critical section */
7 flag[i] = false;
```




P1 does the same. Even x86 (TSO) may let a store complete **after** a later load.

Step 3 / 6

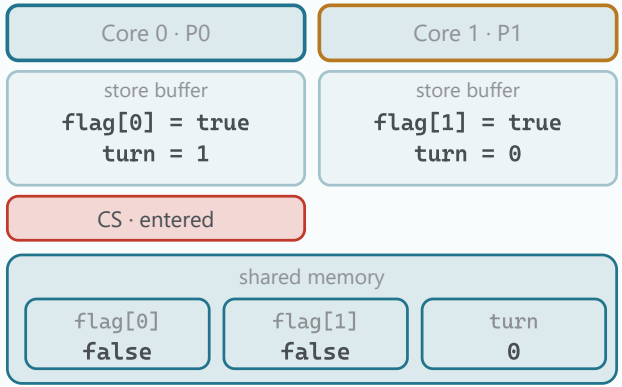
**VARIABLES**

—

The compiler may also hoist that load. C11 `_Atomic` with the default `memory_order_seq_cst`, or Java `volatile` fields, restore the order.

# Why Peterson Fails on a Modern CPU

```
1 /* P_i  (i = 0 or 1, j = 1 - i) */
2 flag[i] = true;
3 turn = j;
▶ while (flag[j] && turn == j)
5   ;
6 /* critical section */
7 flag[i] = false;
```




P0 reads `flag[1]` from memory: **false** → leaves the loop.

Step 4 / 6

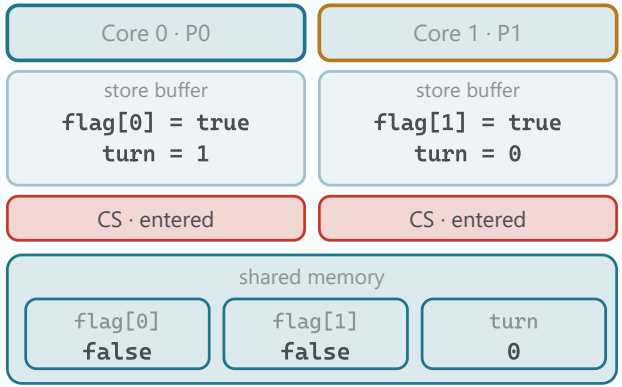
VARIABLES

`P0: flag[1] = false`

The compiler may also hoist that load. C11 `_Atomic` with the default `memory_order_seq_cst`, or Java `volatile` fields, restore the order.

# Why Peterson Fails on a Modern CPU

```
1 /* P_i  (i = 0 or 1, j = 1 - i) */
2 flag[i] = true;
3 turn = j;
▶ while (flag[j] && turn == j)
5   ;
6 /* critical section */
7 flag[i] = false;
```




P1 also reads `flag[0]` as **false** and enters: **both are in the critical section!**

Step 5 / 6

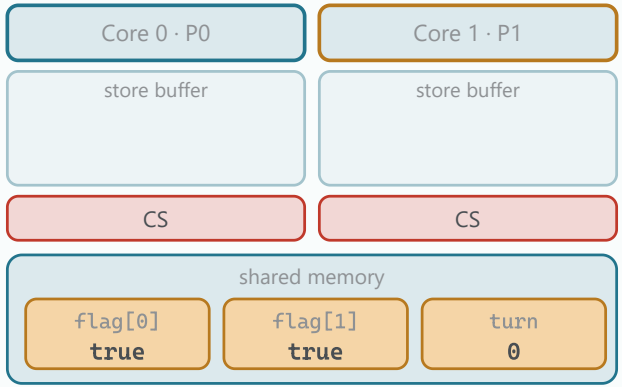
VARIABLES

`P0: flag[1] = false` `P1: flag[0] = false`

The compiler may also hoist that load. C11 `_Atomic` with the default `memory_order_seq_cst`, or Java `volatile` fields, restore the order.

# Why Peterson Fails on a Modern CPU

```
1 /* P_i  (i = 0 or 1, j = 1 - i) */
2 flag[i] = true;
3 turn = j;
4 while (flag[j] && turn == j)
5   ;
▶ /* critical section */
7 flag[i] = false;
```




The buffers drain later, but the damage is done. Fix: a **full memory barrier** between the store and the load, or sequentially consistent atomics.

Step 6 / 6

**VARIABLES**

—

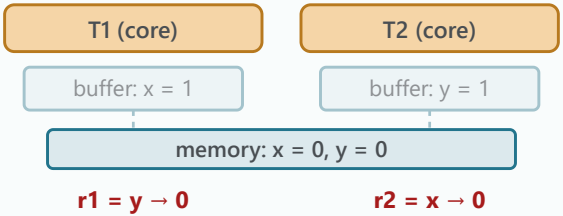
The compiler may also hoist that load. C11 `_Atomic` with the default `memory_order_seq_cst`, or Java `volatile` fields, restore the order.

# Memory Models and Reordering

```
/* x = y = 0 */
/* T1 */          /* T2 */
x = 1;            y = 1;
r1 = y;           r2 = x;

/* r1 == 0 && r2 == 0 ?  */
```

Intuitively at least one should see 1. Yet on x86, ARM, and RISC-V, **r1 = r2 = 0** can be observed (the store-buffer litmus test).




| Model | Reordering allowed |
|---|---|
| **Sequential consistency** (SC) | None; all operations fit one global order that preserves each thread's program order (Lamport 1979). |
| x86-64 (TSO) | Only *store → later load*; stores stay ordered among themselves. |
| ARMv8, RISC-V (weak) | Loads and stores may pass each other when there is no dependency. |
| Compiler | May move, cache, or remove any access if the single-threaded result is unchanged. |

In C, `volatile` gives threads **neither atomicity nor ordering**. The right tools: C11 `<stdatomic.h>`, Java `volatile` / `java.util.concurrent`, locks.

# Memory Barriers

Thread 1 (reader)

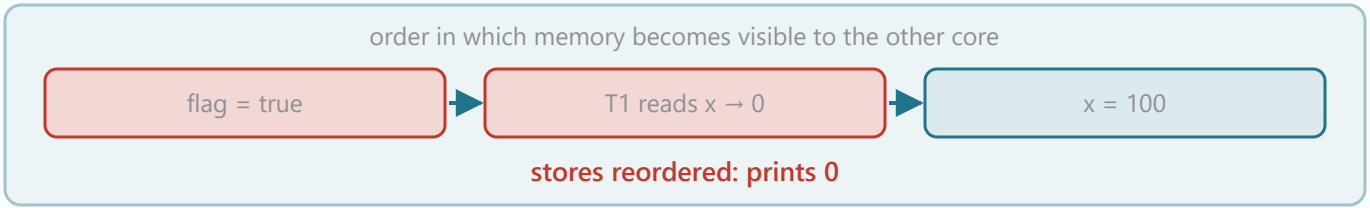
```
while (!flag) ;

print(x);
```

Thread 2 (writer)

```
x = 100;

flag = true;
```




No barrier: on a weak model T2's stores may become visible in another order; T1 sees `flag` true while `x` is still 0.

Step 1 / 4

The example from Silberschatz 10e §6.4.1. Lock and semaphore implementations already contain these barriers: code protected by a lock needs no extra fence.

# Memory Barriers

Thread 1 (reader)

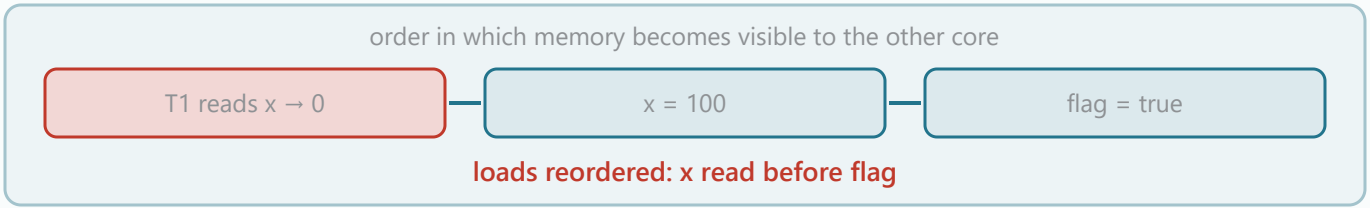
```
while (!flag) ;

print(x);
```

Thread 2 (writer)

```
x = 100;

flag = true;
```




Or T1 itself may read `x` **before** `flag` (load–load reordering).



The example from Silberschatz 10e §6.4.1. Lock and semaphore implementations already contain these barriers: code protected by a lock needs no extra fence.

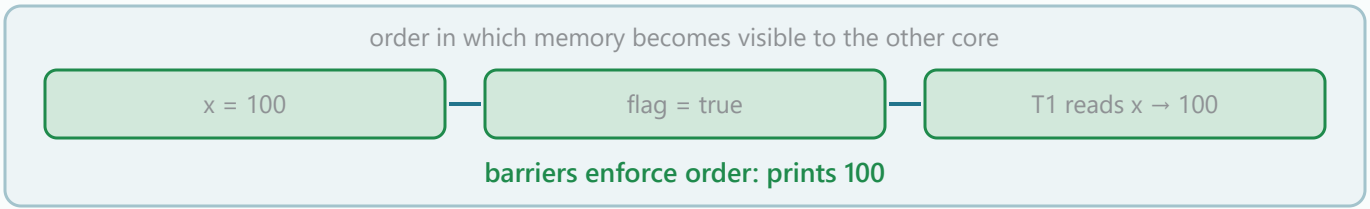
# Memory Barriers

Thread 1 (reader)

```
while (!flag) ;
memory_barrier();
print(x);
```

Thread 2 (writer)

```
x = 100;
memory_barrier();
flag = true;
```




**Memory barrier** (fence): no later load or store starts until all earlier ones complete. T2: `x` first; T1: `flag` first.



The example from Silberschatz 10e §6.4.1. Lock and semaphore implementations already contain these barriers: code protected by a lock needs no extra fence.

# Memory Barriers

**Thread 1 (reader)**

```
while (!flag) ;
memory_barrier();
print(x);
```

**Thread 2 (writer)**

```
x = 100;
memory_barrier();
flag = true;
```

order in which memory becomes visible to the other core

x = 100 — store-release flag — load-acquire flag → x = 100

**C11 release/acquire or Java volatile gives the same guarantee**

In real code: x86 `mfence`, ARM `dmb`, Linux `smp_mb()`, C11 `atomic_thread_fence`; release/acquire atomics usually suffice.



The example from Silberschatz 10e §6.4.1. Lock and semaphore implementations already contain these barriers: code protected by a lock needs no extra fence.

# Peterson in Java: A Correct Implementation

```
public class Peterson {
  static volatile boolean flag0, flag1;
  static volatile int turn;
  static int counter = 0;

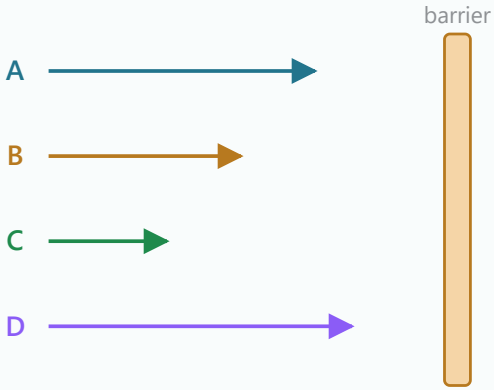
  static void lock(int i) {
    if (i == 0) { flag0 = true; turn = 1;
      while (flag1 && turn == 1) Thread.onSpinWait(); }
    else        { flag1 = true; turn = 0;
      while (flag0 && turn == 0) Thread.onSpinWait(); }
  }
  static void unlock(int i) {
    if (i == 0) flag0 = false; else flag1 = false;
  }
  /* main: two threads, i = 0 and i = 1,
     each 100_000 times: lock(i); counter++; unlock(i);
     join both, print counter */
}
```

- Three common mistakes:
  - `volatile boolean[] flag` makes only the **array reference** volatile, not its elements.
  - `getId() % N` can give two threads the **same** `i`.
  - A `print` inside the critical section distorts the measurement.
- In Java, `volatile` reads and writes are **sequentially consistent**; with separate volatile fields Peterson works correctly.
- `Thread.onSpinWait()` (Java 9+) hints to the CPU that it is spinning (x86 `pause`).

OUTPUT (JDK 21, THREE RUNS)

```
counter = 200000
counter = 200000
counter = 200000
```

# Synchronization Barrier




each process computes its own phase

A parallel computation split into phases (e.g., a grid simulation): each process computes one slice.

Step 1 / 3

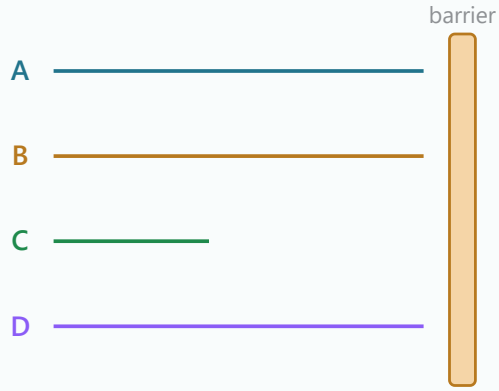
```
var b = new CyclicBarrier(3,
  () -> System.out.println(
      "phase " + phase[0]++ + " done"));
/* 3 threads, each:
   for (f = 0; f < 2; f++) {
     compute(f); b.await();
   } */
```

OUTPUT (JDK 21)

```
phase 0 done
phase 1 done
```

"Barrier" names two different ideas: a **synchronization barrier** brings a group of processes together ( `pthread_barrier_wait` , `CyclicBarrier` ); a **memory barrier** enforces the order of one core's memory accesses.

# Synchronization Barrier




A, B, D blocked at barrier · C running

A process that finishes calls `barrier_wait()` and **blocks**; everyone waits for the slowest (C).

Step 2 / 3

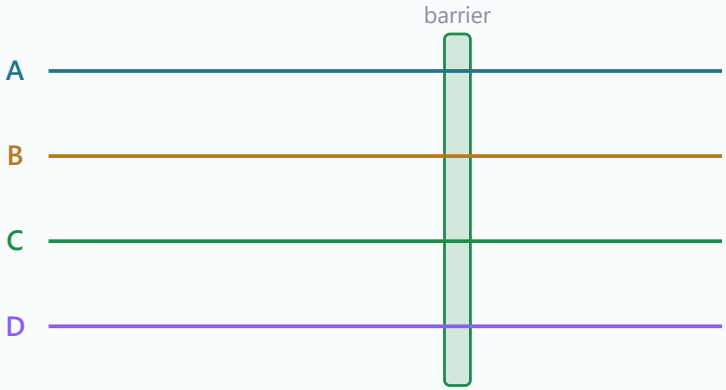
```
var b = new CyclicBarrier(3,
  () -> System.out.println(
     "phase " + phase[0]++ + " done"));
/* 3 threads, each:
   for (f = 0; f < 2; f++) {
     compute(f); b.await();
   } */
```

OUTPUT (JDK 21)

```
phase 0 done
phase 1 done
```

"Barrier" names two different ideas: a **synchronization barrier** brings a group of processes together ( `pthread_barrier_wait` , `CyclicBarrier` ); a **memory barrier** enforces the order of one core's memory accesses.

# Synchronization Barrier




**last process arrived: all move to next phase**

When the Nth process arrives the barrier opens; all move to the next phase together.

Step 3 / 3

```
var b = new CyclicBarrier(3,
  () -> System.out.println(
      "phase " + phase[0]++ + " done"));
/* 3 threads, each:
   for (f = 0; f < 2; f++) {
     compute(f); b.await();
   } */
```

OUTPUT (JDK 21)

```
phase 0 done
phase 1 done
```

"Barrier" names two different ideas: a **synchronization barrier** brings a group of processes together ( `pthread_barrier_wait` , `CyclicBarrier` ); a **memory barrier** enforces the order of one core's memory accesses.

# Atomic Hardware Instructions

```
/* runs as ONE atomic instruction */
bool test_and_set(bool *target) {
  bool old = *target;
  *target = true;
  return old;
}

/* usage: a simple spinlock */
bool lock = false;
while (test_and_set(&lock))
  ;            /* busy wait */
/* critical section */
lock = false;
```

- The trouble with software solutions: *read* and *write* are separate steps. Hardware provides instructions that test and modify a memory word **atomically**.
- Atomicity holds on multicore too: the cache-coherence protocol gives one core exclusive ownership of the line for the whole instruction (older systems locked the memory bus).

| Architecture | Atomic read-modify-write |
|---|---|
| x86-64 | `XCHG`, `LOCK CMPXCHG`, `LOCK XADD`, `LOCK BTS` |
| ARMv8 | `LDXR/STXR` (LL/SC); v8.1 LSE: `CAS`, `LDADD`, `SWP` |
| RISC-V “A” | `LR/SC`, `AMOSWAP`, `AMOADD` |
| Tanenbaum's hypothetical CPU | `TSL REGISTER,LOCK` |

# Critical Section with TSL

```
1 enter_region:
2   TSL REGISTER,LOCK   | R = LOCK; LOCK = 1
3   CMP REGISTER,#0     | was LOCK 0?
4   JNE enter_region    | no: try again
5   RET                 | entered critical
  section
6
7 leave_region:
8   MOVE LOCK,#0        | LOCK = 0
9   RET
```

| CPU 0 | CPU 1 |
| --- | --- |
| REGISTER = – | REGISTER = – |
| outside critical section | outside critical section |

LOCK (memory)

**0**

Lock is free: `LOCK = 0`.

Step 1 / 6

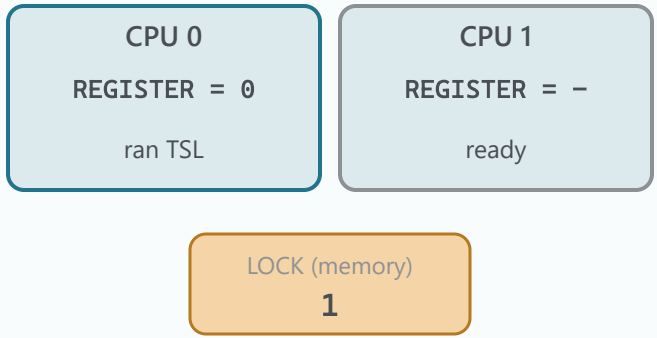
**VARIABLES**

`LOCK = 0`

Tanenbaum MOS Figure 2-25. TSL works on multicore too: unlike disabling interrupts, it also blocks other CPUs from that word.

# Critical Section with TSL

```
1 enter_region:
▶   TSL REGISTER,LOCK   | R = LOCK; LOCK = 1
3   CMP REGISTER,#0     | was LOCK 0?
4   JNE enter_region    | no: try again
5   RET                 | entered critical
  section
6
7 leave_region:
8   MOVE LOCK,#0        | LOCK = 0
9   RET
```




CPU 0 runs `TSL`: old value (0) into the register, 1 into LOCK, **in one indivisible step**.

Step 2 / 6

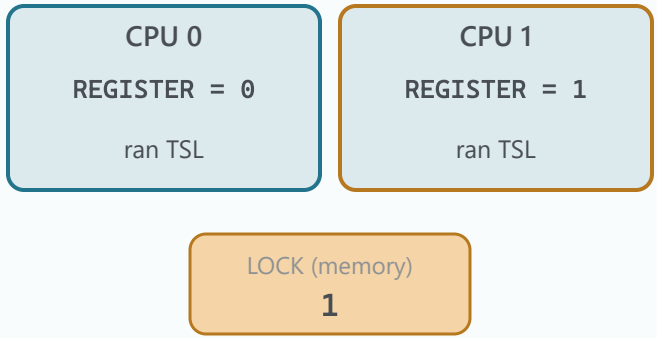
**VARIABLES**

`LOCK = 1` `R0 = 0`

Tanenbaum MOS Figure 2-25. TSL works on multicore too: unlike disabling interrupts, it also blocks other CPUs from that word.

# Critical Section with TSL

```
1 enter_region:
▶   TSL REGISTER,LOCK   | R = LOCK; LOCK = 1
3   CMP REGISTER,#0     | was LOCK 0?
4   JNE enter_region    | no: try again
5   RET                 | entered critical
  section
6
7 leave_region:
8   MOVE LOCK,#0        | LOCK = 0
9   RET
```




CPU 1 also runs `TSL`: its register gets **1**; LOCK is already 1.

Step 3 / 6

**VARIABLES**

`LOCK = 1` `R0 = 0` `R1 = 1`

Tanenbaum MOS Figure 2-25. TSL works on multicore too: unlike disabling interrupts, it also blocks other CPUs from that word.

# Critical Section with TSL

```
1 enter_region:
2   TSL REGISTER,LOCK   | R = LOCK; LOCK = 1
▶   CMP REGISTER,#0     | was LOCK 0?
▶   JNE enter_region    | no: try again
5   RET                 | entered critical
  section
6
7 leave_region:
8   MOVE LOCK,#0        | LOCK = 0
9   RET
```

**CPU 0**

**REGISTER = 0**

CMP: 0 → RET, in CS

**CPU 1**

**REGISTER = 1**

JNE → loop (spin)

LOCK (memory)

**1**

CPU 0 saw zero: it is in its critical section. CPU 1 saw nonzero: it **busy-waits**.

Step 4 / 6

**VARIABLES**

`LOCK = 1` `R0 = 0` `R1 = 1`

Tanenbaum MOS Figure 2-25. TSL works on multicore too: unlike disabling interrupts, it also blocks other CPUs from that word.

# Critical Section with TSL

```
1 enter_region:
2   TSL REGISTER,LOCK   | R = LOCK; LOCK = 1
3   CMP REGISTER,#0     | was LOCK 0?
4   JNE enter_region    | no: try again
5   RET                 | entered critical
  section
6
7 leave_region:
▶   MOVE LOCK,#0        | LOCK = 0
9   RET
```

**CPU 0**

**REGISTER = 0**

MOVE LOCK,#0 → left

**CPU 1**

**REGISTER = 1**

still trying

LOCK (memory)

**0**

On exit CPU 0 sets LOCK to 0 with an ordinary store.

Step 5 / 6

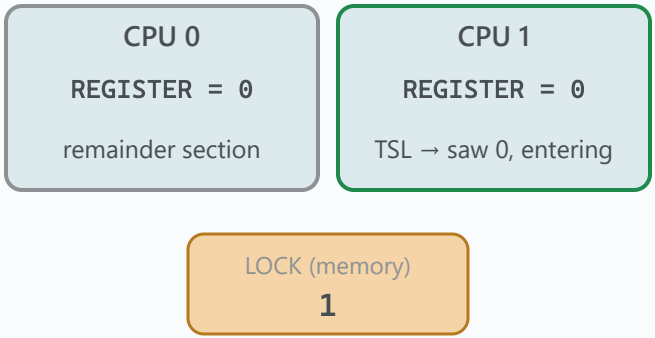
**VARIABLES**

LOCK = **0** R0 = **0** R1 = **1**

Tanenbaum MOS Figure 2-25. TSL works on multicore too: unlike disabling interrupts, it also blocks other CPUs from that word.

# Critical Section with TSL

```
1 enter_region:
▶   TSL REGISTER,LOCK   | R = LOCK; LOCK = 1
3   CMP REGISTER,#0     | was LOCK 0?
4   JNE enter_region    | no: try again
5   RET                 | entered critical
  section
6
7 leave_region:
8   MOVE LOCK,#0        | LOCK = 0
9   RET
```




CPU 1's next TSL reads 0 and acquires the lock. The two CPUs are never inside together.

Step 6 / 6

**VARIABLES**

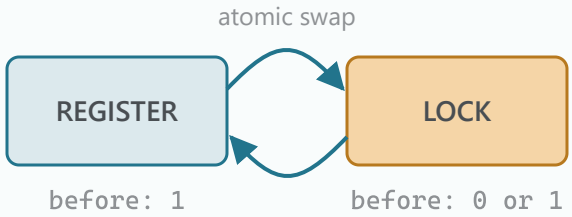
`LOCK = 1` `R0 = 0` `R1 = 0`

Tanenbaum MOS Figure 2-25. TSL works on multicore too: unlike disabling interrupts, it also blocks other CPUs from that word.

# The XCHG Instruction

```
enter_region:
  MOVE REGISTER,#1     | put 1 in register
  XCHG REGISTER,LOCK   | swap register and LOCK
  CMP REGISTER,#0      | was LOCK 0?
  JNE enter_region     | no: loop
  RET                  | entered critical section
leave_region:
  MOVE LOCK,#0         | LOCK = 0
  RET
```

- Swaps the contents of two operands **in one step**; the effect is the same as TSL.
- The instruction Intel x86 uses for locking: `XCHG` with a memory operand is implicitly `LOCK`-prefixed, hence atomic.
- C11 equivalent: `atomic_exchange(&lock, 1)`.




**after: REGISTER = old LOCK**

**LOCK = 1 (always)**

old LOCK 0 means lock acquired

# A Lock with compare_and_swap

```
1  int compare_and_swap(int *v, int exp, int nv) {
2    int old = *v;           /* these three lines */
3    if (old == exp)         /* are ONE atomic op */
4      *v = nv;
5    return old;
6  }
7
▶ int lock = 0;
9  void enter(void) {
10   while (compare_and_swap(&lock, 0, 1) != 0)
11     ;                     /* busy wait */
12 }
13 void leave(void) { lock = 0; }
```

T0 outside

T1 outside

The lock is free.

Step 1 / 6

VARIABLES

`lock = 0`

Silberschatz 10e §6.4.2. x86: `LOCK CMPXCHG`; C11: `atomic_compare_exchange_strong`; Java: `AtomicInteger.compareAndSet`.

# A Lock with compare_and_swap

```
1  int compare_and_swap(int *v, int exp, int nv) {
▶    int old = *v;            /* these three lines */
▶    if (old == exp)          /* are ONE atomic op */
▶      *v = nv;
5    return old;
6  }
7
8  int lock = 0;
9  void enter(void) {
▶    while (compare_and_swap(&lock, 0, 1) != 0)
11     ;                      /* busy wait */
12 }
13 void leave(void) { lock = 0; }
```

T0 — CAS(0→1): old 0 → won

T1 — ready

T0's CAS finds `lock` at 0, writes 1, and **returns 0**: it leaves the loop and is in its critical section.

Step 2 / 6

VARIABLES

`lock = 1` `T0.old = 0`

Silberschatz 10e §6.4.2. x86: `LOCK CMPXCHG` ; C11: `atomic_compare_exchange_strong` ; Java: `AtomicInteger.compareAndSet` .

# A Lock with compare_and_swap

```
1  int compare_and_swap(int *v, int exp, int nv) {
▶    int old = *v;            /* these three lines */
▶    if (old == exp)          /* are ONE atomic op */
4      *v = nv;
5    return old;
6  }
7
8  int lock = 0;
9  void enter(void) {
▶    while (compare_and_swap(&lock, 0, 1) != 0)
11     ;                      /* busy wait */
12 }
13 void leave(void) { lock = 0; }
```

T0 — in critical section

T1 — CAS: old 1 ≠ 0 → no write

T1's CAS finds 1: not the expected value, so it **writes nothing** and returns 1.

Step 3 / 6

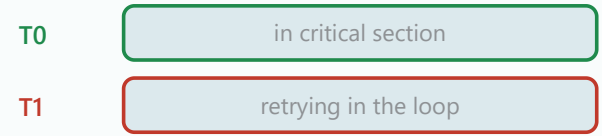
VARIABLES

`lock = 1` `T1.old = 1`

Silberschatz 10e §6.4.2. x86: `LOCK CMPXCHG`; C11: `atomic_compare_exchange_strong`; Java: `AtomicInteger.compareAndSet`.

# A Lock with compare_and_swap

```
1  int compare_and_swap(int *v, int exp, int nv) {
2    int old = *v;           /* these three lines */
3    if (old == exp)         /* are ONE atomic op */
4      *v = nv;
5    return old;
6  }
7
8  int lock = 0;
9  void enter(void) {
▶    while (compare_and_swap(&lock, 0, 1) != 0)
▶      ;                     /* busy wait */
12 }
13 void leave(void) { lock = 0; }
```




T1 busy-waits; each attempt returns without changing the memory word.

Step 4 / 6

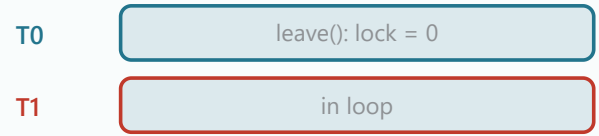
VARIABLES

`lock = 1` `T1.old = 1`

Silberschatz 10e §6.4.2. x86: `LOCK CMPXCHG`; C11: `atomic_compare_exchange_strong`; Java: `AtomicInteger.compareAndSet`.

# A Lock with compare_and_swap

```
1  int compare_and_swap(int *v, int exp, int nv) {
2    int old = *v;           /* these three lines */
3    if (old == exp)         /* are ONE atomic op */
4      *v = nv;
5    return old;
6  }
7
8  int lock = 0;
9  void enter(void) {
10   while (compare_and_swap(&lock, 0, 1) != 0)
11     ;                     /* busy wait */
12 }
▶  void leave(void) { lock = 0; }
```




T0 sets the lock to 0 on exit (in real code, a store with release ordering).

Step 5 / 6

VARIABLES

`lock = 0`

Silberschatz 10e §6.4.2. x86: `LOCK CMPXCHG`; C11: `atomic_compare_exchange_strong`; Java: `AtomicInteger.compareAndSet`.

# A Lock with compare_and_swap

```
1  int compare_and_swap(int *v, int exp, int nv) {
▶    int old = *v;            /* these three lines */
▶    if (old == exp)          /* are ONE atomic op */
▶      *v = nv;
5    return old;
6  }
7
8  int lock = 0;
9  void enter(void) {
▶    while (compare_and_swap(&lock, 0, 1) != 0)
11     ;                      /* busy wait */
12 }
13 void leave(void) { lock = 0; }
```

T0 — remainder section

T1 — CAS(0→1) won → entered

T1's next CAS succeeds. Note: this lock does not guarantee **bounded waiting**; an unlucky thread may lose every time.

Step 6 / 6

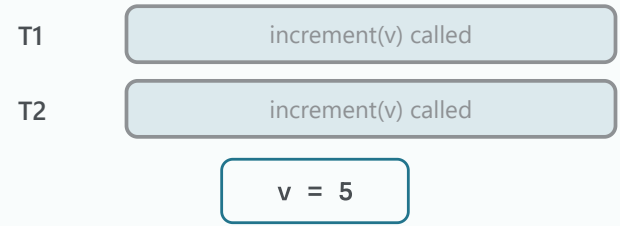
VARIABLES

`lock = 1` `T1.old = 0`

Silberschatz 10e §6.4.2. x86: `LOCK CMPXCHG`; C11: `atomic_compare_exchange_strong`; Java: `AtomicInteger.compareAndSet`.

# Atomic Increment with CAS

```
▶ void increment(int *v) {
2   int temp;
3   do {
4     temp = *v;                /* read                */
5   } while (temp != compare_and_swap(v, temp, temp +
    1));
6 }                             /* write if unchanged */
7
8 /* C11: if the counter is atomic_int, one line:
    atomic_fetch_add(v, 1); */
```




Two threads want to increment the same counter; initially `v = 5`.

Step 1 / 7

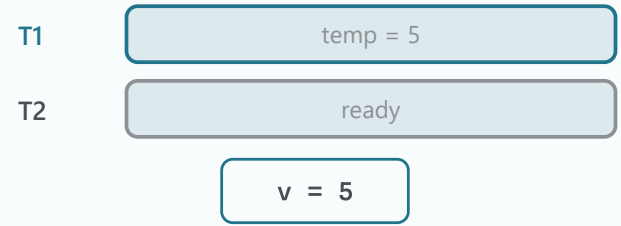
**VARIABLES**

v = **5**

This "read → compute → CAS → retry on failure" loop is the basic pattern of lock-free data structures.

# Atomic Increment with CAS

```
1 void increment(int *v) {
2   int temp;
3   do {
▶     temp = *v;              /* read                */
5   } while (temp != compare_and_swap(v, temp, temp +
  1));
6 }                           /* write if unchanged */
7
8 /* C11: if the counter is atomic_int, one line:
  atomic_fetch_add(v, 1); */
```




T1 reads the value: `temp = 5`. It may be interrupted here; there is no lock.

Step 2 / 7

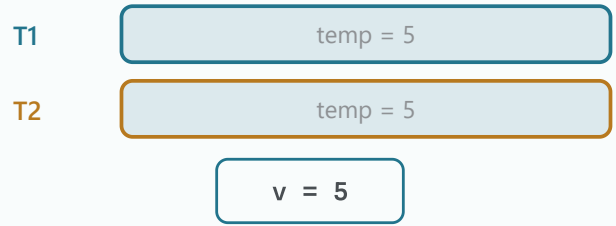
**VARIABLES**

`v = 5` `T1.temp = 5`

This "read → compute → CAS → retry on failure" loop is the basic pattern of lock-free data structures.

# Atomic Increment with CAS

```
1 void increment(int *v) {
2   int temp;
3   do {
▶     temp = *v;              /* read                */
5   } while (temp != compare_and_swap(v, temp, temp +
  1));
6 }                           /* write if unchanged */
7
8 /* C11: if the counter is atomic_int, one line:
  atomic_fetch_add(v, 1); */
```




T2 also reads 5. With a plain `v++` one increment would be lost.

Step 3 / 7

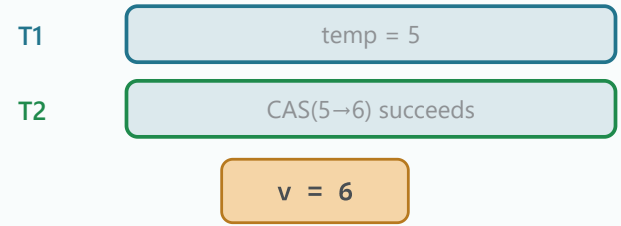
**VARIABLES**

`v = 5` `T1.temp = 5` `T2.temp = 5`

This "read → compute → CAS → retry on failure" loop is the basic pattern of lock-free data structures.

# Atomic Increment with CAS

```
1 void increment(int *v) {
2   int temp;
3   do {
4     temp = *v;                /* read                */
  } while (temp != compare_and_swap(v, temp, temp +
1));
6 }                             /* write if unchanged */
7
8 /* C11: if the counter is atomic_int, one line:
  atomic_fetch_add(v, 1); */
```




T2's CAS finds v still 5, writes 6, and returns 5: equal to `temp` → done.

Step 4 / 7

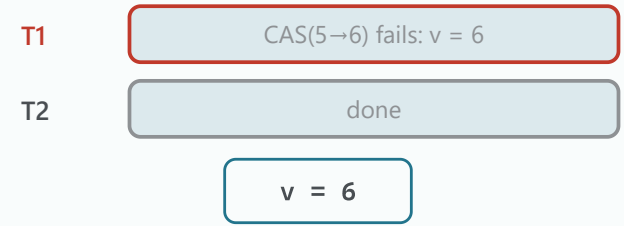
**VARIABLES**

`v = 6` `T1.temp = 5` `T2.temp = 5`

This "read → compute → CAS → retry on failure" loop is the basic pattern of lock-free data structures.

# Atomic Increment with CAS

```
1 void increment(int *v) {
2   int temp;
3   do {
4     temp = *v;                /* read                */
▶    } while (temp != compare_and_swap(v, temp, temp +
  1));
6 }                             /* write if unchanged */
7
8 /* C11: if the counter is atomic_int, one line:
  atomic_fetch_add(v, 1); */
```




T1's CAS finds v at 6, not the expected 5; it writes nothing and returns 6 → **retry**.

Step 5 / 7

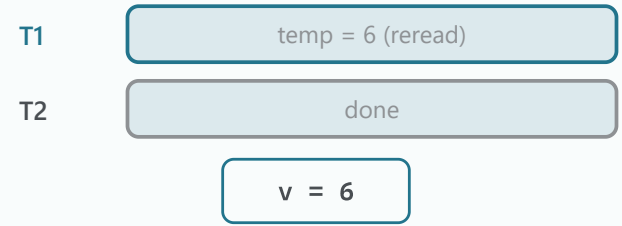
**VARIABLES**

`v = 6` `T1.temp = 5` `T2.temp = 5`

This "read → compute → CAS → retry on failure" loop is the basic pattern of lock-free data structures.

# Atomic Increment with CAS

```
1 void increment(int *v) {
2   int temp;
3   do {
▶     temp = *v;              /* read                */
5   } while (temp != compare_and_swap(v, temp, temp +
  1));
6 }                           /* write if unchanged */
7
8 /* C11: if the counter is atomic_int, one line:
  atomic_fetch_add(v, 1); */
```




T1 reads the current value.

Step 6 / 7

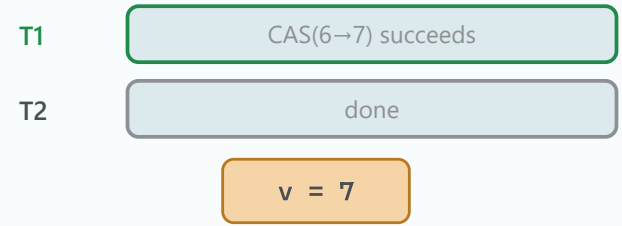
**VARIABLES**

`v = 6` `T1.temp = 6`

This "read → compute → CAS → retry on failure" loop is the basic pattern of lock-free data structures.

# Atomic Increment with CAS

```
1 void increment(int *v) {
2   int temp;
3   do {
4     temp = *v;                /* read                */
  } while (temp != compare_and_swap(v, temp, temp +
1));
6 }                             /* write if unchanged */
7
8 /* C11: if the counter is atomic_int, one line:
  atomic_fetch_add(v, 1); */
```




This time it succeeds: **v = 7**. No increment was lost and no one held a lock (a *lock-free* update).

Step 7 / 7

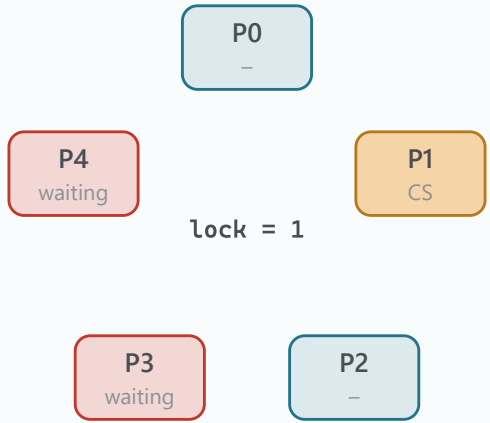
VARIABLES

`v = 7` `T1.temp = 6`

This "read → compute → CAS → retry on failure" loop is the basic pattern of lock-free data structures.

# Bounded Waiting with CAS

```
while (true) {
  waiting[i] = true;
  key = 1;
  while (waiting[i] && key == 1)
    key = compare_and_swap(&lock, 0, 1);
  waiting[i] = false;
  /* critical section */
  j = (i + 1) % n;
  while ((j != i) && !waiting[j])
    j = (j + 1) % n;
  if (j == i)
    lock = 0;            /* no one waiting */
  else
    waiting[j] = false;  /* hand over turn */
  /* remainder section */
}
```



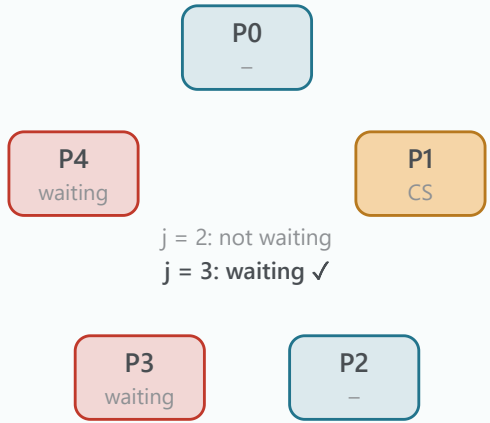

at most n − 1 = 4 turns of waiting

P1 is in its critical section; P3 and P4 wait with their `waiting` flags set.

Step 1 / 3

# Bounded Waiting with CAS

```
while (true) {
  waiting[i] = true;
  key = 1;
  while (waiting[i] && key == 1)
    key = compare_and_swap(&lock, 0, 1);
  waiting[i] = false;
  /* critical section */
  j = (i + 1) % n;
  while ((j != i) && !waiting[j])
    j = (j + 1) % n;
  if (j == i)
    lock = 0;            /* no one waiting */
  else
    waiting[j] = false;  /* hand over turn */
  /* remainder section */
}
```

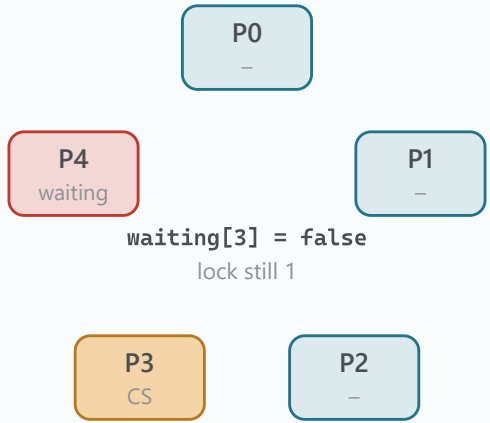



at most n − 1 = 4 turns of waiting

On exit P1 scans `j = 2, 3, …` in order; the first waiter is P3.

# Bounded Waiting with CAS

```
while (true) {
  waiting[i] = true;
  key = 1;
  while (waiting[i] && key == 1)
    key = compare_and_swap(&lock, 0, 1);
  waiting[i] = false;
  /* critical section */
  j = (i + 1) % n;
  while ((j != i) && !waiting[j])
    j = (j + 1) % n;
  if (j == i)
    lock = 0;            /* no one waiting */
  else
    waiting[j] = false;  /* hand over turn */
  /* remainder section */
}
```



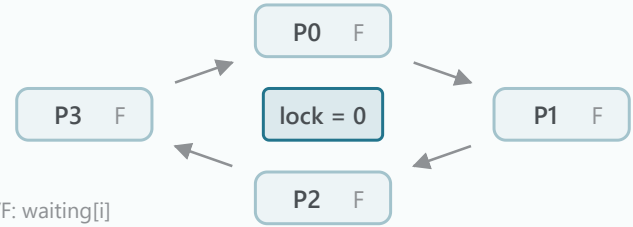

at most n − 1 = 4 turns of waiting

It does not release the lock; it sets `waiting[3] = false`: P3's loop ends and it enters. The cyclic scan ensures **bounded waiting**.

Step 3 / 3

# Bounded Waiting with CAS: Code Trace

```
1  /* P_i; n = 4 processes */
2  waiting[i] = true; key = 1;
3  while (waiting[i] && key == 1)
4    key = compare_and_swap(&lock, 0, 1);
5  waiting[i] = false;
6  /* critical section */
7  j = (i + 1) % n;
8  while ((j != i) && !waiting[j])
9    j = (j + 1) % n;
10 if (j == i)
11   lock = 0;            /* no one waiting */
12 else
13   waiting[j] = false;  /* hand over turn */
```




The lock is free ( `lock = 0` ) and no one is waiting. The four processes will be scanned in ring order.

Step 1 / 11

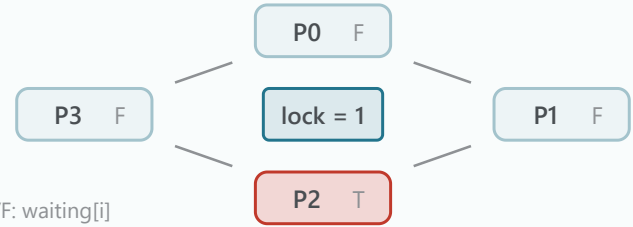
**VARIABLES**

`lock = 0` `key = –` `j = –`

`waiting = [F F F F]` `CS = –`

# Bounded Waiting with CAS: Code Trace

```
1  /* P_i; n = 4 processes */
2  waiting[i] = true; key = 1;
3  while (waiting[i] && key == 1)
4    key = compare_and_swap(&lock, 0, 1);
5  waiting[i] = false;
6  /* critical section */
7  j = (i + 1) % n;
8  while ((j != i) && !waiting[j])
9    j = (j + 1) % n;
10 if (j == i)
11   lock = 0;            /* no one waiting */
12 else
13   waiting[j] = false;  /* hand over turn */
```




P2 arrives: `waiting[2] = true` ; CAS finds `lock` at 0, writes 1, returns 0 → `key = 0` .

Step 2 / 11

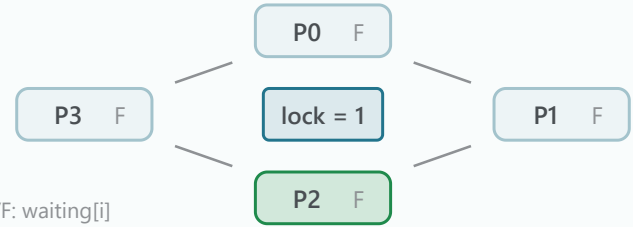
**VARIABLES**

`lock = 1` `key = 0` `j = –`

`waiting = [F F T F]` `CS = –`

# Bounded Waiting with CAS: Code Trace

```
1  /* P_i; n = 4 processes */
2  waiting[i] = true; key = 1;
3  while (waiting[i] && key == 1)
4    key = compare_and_swap(&lock, 0, 1);
▶  waiting[i] = false;
▶  /* critical section */
7  j = (i + 1) % n;
8  while ((j != i) && !waiting[j])
9    j = (j + 1) % n;
10 if (j == i)
11   lock = 0;            /* no one waiting */
12 else
13   waiting[j] = false;  /* hand over turn */
```




The loop ends; P2 sets `waiting[2] = false` and enters its critical section.

Step 3 / 11

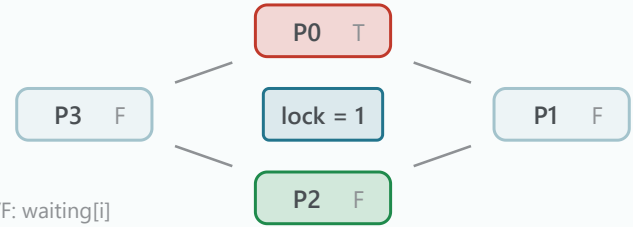
**VARIABLES**

`lock = 1` `key = 0` `j = –`

`waiting = [F F F F]` `CS = P2`

# Bounded Waiting with CAS: Code Trace

```
1  /* P_i; n = 4 processes */
2  waiting[i] = true; key = 1;
▶  while (waiting[i] && key == 1)
▶    key = compare_and_swap(&lock, 0, 1);
5  waiting[i] = false;
6  /* critical section */
7  j = (i + 1) % n;
8  while ((j != i) && !waiting[j])
9    j = (j + 1) % n;
10 if (j == i)
11   lock = 0;            /* no one waiting */
12 else
13   waiting[j] = false;  /* hand over turn */
```




P0 arrives: CAS returns 1 (lock held) → `key = 1`, it spins.

Step 4 / 11

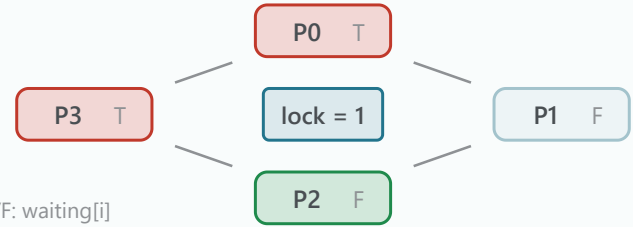
**VARIABLES**

`lock = 1` `key = 1` `j = –`

`waiting = [T F F F]` `CS = P2`

# Bounded Waiting with CAS: Code Trace

```
1  /* P_i; n = 4 processes */
2  waiting[i] = true; key = 1;
▶  while (waiting[i] && key == 1)
▶    key = compare_and_swap(&lock, 0, 1);
5  waiting[i] = false;
6  /* critical section */
7  j = (i + 1) % n;
8  while ((j != i) && !waiting[j])
9    j = (j + 1) % n;
10 if (j == i)
11   lock = 0;            /* no one waiting */
12 else
13   waiting[j] = false;  /* hand over turn */
```




P3 arrives and spins too. Waiting: P0 and P3.

Step 5 / 11

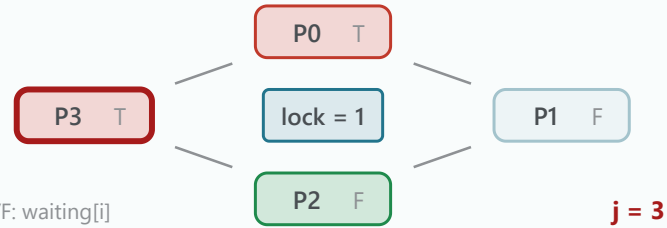
**VARIABLES**

`lock = 1` `key = 1` `j = –`

`waiting = [T F F T]` `CS = P2`

# Bounded Waiting with CAS: Code Trace

```
1  /* P_i; n = 4 processes */
2  waiting[i] = true; key = 1;
3  while (waiting[i] && key == 1)
4    key = compare_and_swap(&lock, 0, 1);
5  waiting[i] = false;
6  /* critical section */
▶  j = (i + 1) % n;
▶  while ((j != i) && !waiting[j])
9    j = (j + 1) % n;
10 if (j == i)
11   lock = 0;            /* no one waiting */
12 else
13   waiting[j] = false;  /* hand over turn */
```




On exit P2 starts scanning at the next index, **j = 3**; `waiting[3]` is true → the scan stops.

Step 6 / 11

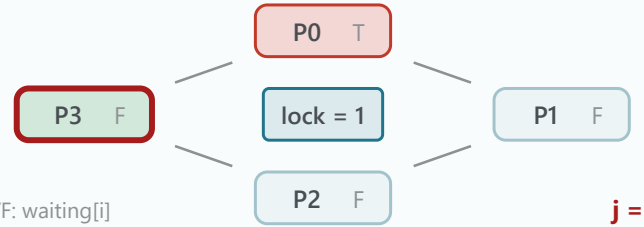
**VARIABLES**

`lock = 1` `key = –` `j = 3`

`waiting = [T F F T]` `CS = P2`

# Bounded Waiting with CAS: Code Trace

```
1  /* P_i; n = 4 processes */
2  waiting[i] = true; key = 1;
3  while (waiting[i] && key == 1)
4    key = compare_and_swap(&lock, 0, 1);
5  waiting[i] = false;
6  /* critical section */
7  j = (i + 1) % n;
8  while ((j != i) && !waiting[j])
9    j = (j + 1) % n;
10 if (j == i)
11   lock = 0;              /* no one waiting */
   else
     waiting[j] = false;  /* hand over turn */
```




P2 does **not release** the lock; it sets `waiting[3] = false` : P3's loop ends and P3 enters. `lock` is still 1.

Step 7 / 11

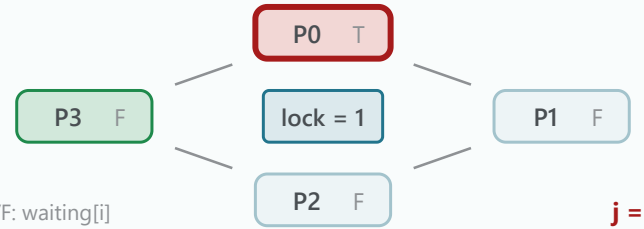
**VARIABLES**

`lock = 1` `key = –` `j = 3`

`waiting = [T F F F]` `CS = P3`

# Bounded Waiting with CAS: Code Trace

```
1  /* P_i; n = 4 processes */
2  waiting[i] = true; key = 1;
3  while (waiting[i] && key == 1)
4    key = compare_and_swap(&lock, 0, 1);
5  waiting[i] = false;
6  /* critical section */
▶  j = (i + 1) % n;
▶  while ((j != i) && !waiting[j])
9    j = (j + 1) % n;
10 if (j == i)
11   lock = 0;            /* no one waiting */
12 else
13   waiting[j] = false;  /* hand over turn */
```




P3 leaves: j = (3 + 1) % 4 = **0**; `waiting[0]` is true → the scan stops.

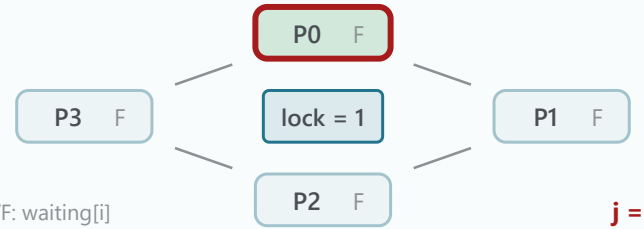
Step 8 / 11

**VARIABLES**

`lock = 1` `key = –` `j = 0` `waiting = [T F F F]` `CS = P3`

# Bounded Waiting with CAS: Code Trace

```
1  /* P_i; n = 4 processes */
2  waiting[i] = true; key = 1;
3  while (waiting[i] && key == 1)
4    key = compare_and_swap(&lock, 0, 1);
5  waiting[i] = false;
6  /* critical section */
7  j = (i + 1) % n;
8  while ((j != i) && !waiting[j])
9    j = (j + 1) % n;
10 if (j == i)
11   lock = 0;            /* no one waiting */
12 else
▶    waiting[j] = false;  /* hand over turn */
```




The turn passes to P0. P0 arrived before P3, but ring order chose P3 first; still, P0 waits for at most n − 1 = 3 entries.

Step 9 / 11

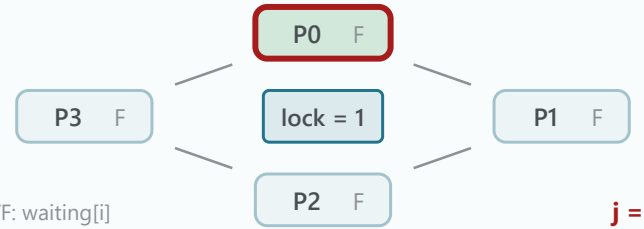
**VARIABLES**

`lock = 1` `key = –` `j = 0`

`waiting = [F F F F]` `CS = P0`

# Bounded Waiting with CAS: Code Trace

```
1  /* P_i; n = 4 processes */
2  waiting[i] = true; key = 1;
3  while (waiting[i] && key == 1)
4    key = compare_and_swap(&lock, 0, 1);
5  waiting[i] = false;
6  /* critical section */
▶  j = (i + 1) % n;
▶  while ((j != i) && !waiting[j])
▶    j = (j + 1) % n;
10 if (j == i)
11   lock = 0;            /* no one waiting */
12 else
13   waiting[j] = false;  /* hand over turn */
```




P0 leaves: none of j = 1, 2, 3 is waiting; j comes back to **0 = i**.

Step 10 / 11

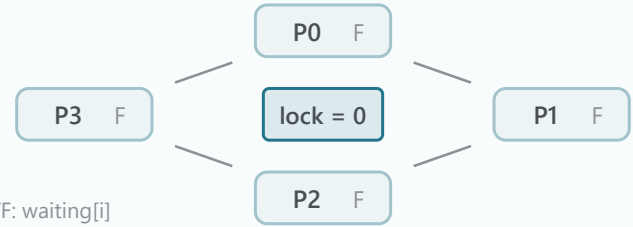
**VARIABLES**

`lock = 1` `key = –` `j = 0`

`waiting = [F F F F]` `CS = P0`

# Bounded Waiting with CAS: Code Trace

```
1  /* P_i; n = 4 processes */
2  waiting[i] = true; key = 1;
3  while (waiting[i] && key == 1)
4    key = compare_and_swap(&lock, 0, 1);
5  waiting[i] = false;
6  /* critical section */
7  j = (i + 1) % n;
8  while ((j != i) && !waiting[j])
9    j = (j + 1) % n;
▶ if (j == i)
▶   lock = 0;              /* no one waiting */
12 else
13   waiting[j] = false;  /* hand over turn */
```




No one waiting → `lock = 0`. Only now is the lock truly free.

Step 11 / 11

**VARIABLES**

`lock = 0` `key = –` `j = –`

`waiting = [F F F F]` `CS = –`

# Atomic Variables

## C11 `<stdatomic.h>`

```c
#include <stdatomic.h>
atomic_int counter = 0;   /* _Atomic int */

void *work(void *arg) {
  for (int n = 0; n < 100000; n++)
    atomic_fetch_add(&counter, 1);
  return NULL;
}
/* 2 threads + join → 200000 */
```

On an `_Atomic` type, `counter++` is atomic too (default `seq_cst`). It compiles to `lock xadd` on x86 and `ldadd` on ARMv8.1.

## Java `AtomicInteger`

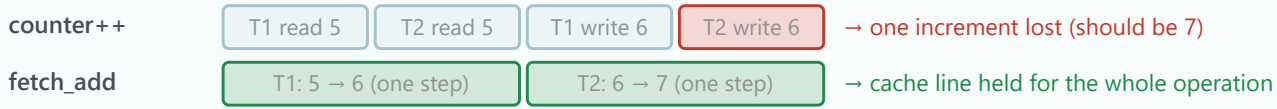
```java
static AtomicInteger atom = new AtomicInteger();
/* two threads, each 100_000 times: */
atom.incrementAndGet();

var v = new AtomicInteger(5);
v.compareAndSet(5, 6);   /* true,  v = 6 */
v.compareAndSet(5, 7);   /* false, v = 6 */
```

OUTPUT (JDK 21, SAME ON EVERY RUN)

```
AtomicInteger = 200000
synchronized  = 200000
true 6
false 6
```

An atomic variable protects updates to **a single variable**. Keeping several variables consistent together (e.g., a buffer array plus a count) still needs a lock or a monitor.



# Spinlocks and Mutexes



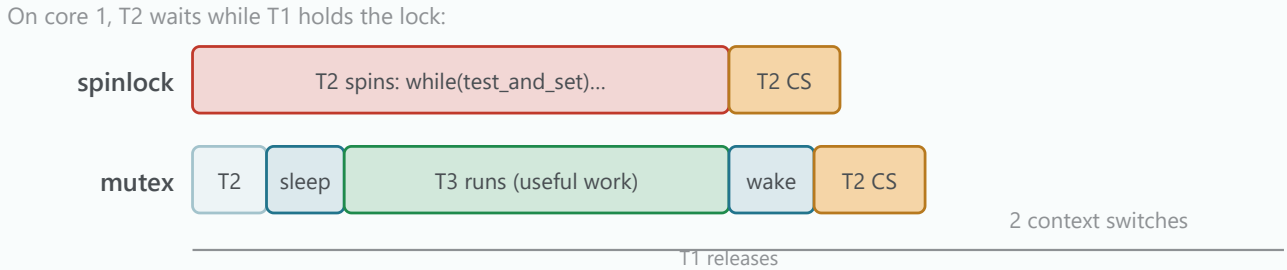

**Spinlock**: the waiter keeps the CPU; when the lock frees it enters with no context switch.

Step 1 / 2

| | **Spinlock** | **Mutex (sleeping)** |
|---|---|---|
| Best for | Multicore, CS shorter than two context switches | Long critical sections, or ones that may block |
| Single core | Pointless: the holder can't run, the waiter burns its slice | Right choice |
| Example | Linux `spin_lock`, `pthread_spin_lock` | `pthread_mutex_t`, Java `synchronized` |

Most mutex implementations spin briefly, then sleep.

# Spinlocks and Mutexes




**Sleeping mutex**: the waiter blocks and the CPU switches to T3; the cost is two context switches.

Step 2 / 2

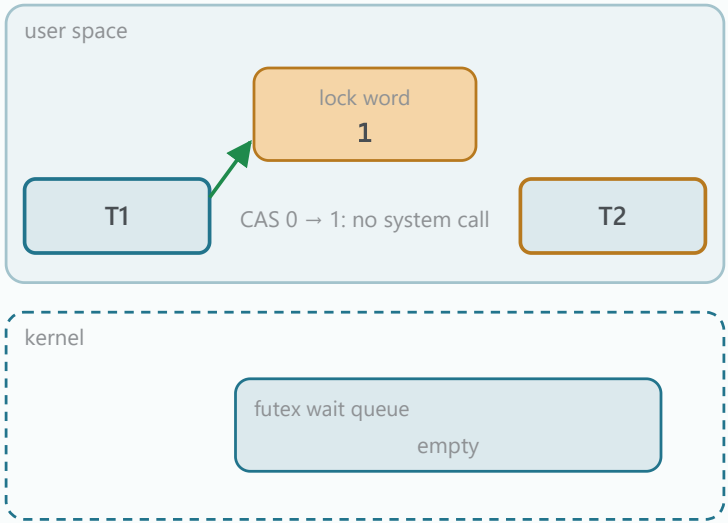
| | **Spinlock** | **Mutex (sleeping)** |
|---|---|---|
| Best for | Multicore, CS shorter than two context switches | Long critical sections, or ones that may block |
| Single core | Pointless: the holder can't run, the waiter burns its slice | Right choice |
| Example | Linux `spin_lock`, `pthread_spin_lock` | `pthread_mutex_t`, Java `synchronized` |

Most mutex implementations spin briefly, then sleep.

# Implementing a Mutex: yield and futex

```
mutex_lock:
  TSL REGISTER,MUTEX | get old, set 1
  CMP REGISTER,#0    | was it free?
  JZE ok             | yes: acquired
  CALL thread_yield  | no: give up CPU
  JMP mutex_lock     | then try again
ok: RET
mutex_unlock:
  MOVE MUTEX,#0
  RET
```

- Tanenbaum Figure 2-29: unlike `enter_region`, it calls `thread_yield` instead of looping, which avoids busy waiting with user-level threads.
- Linux **futex** (fast userspace mutex): no system call when there is no contention; glibc `pthread_mutex`, Java locks, and Go are built on it.

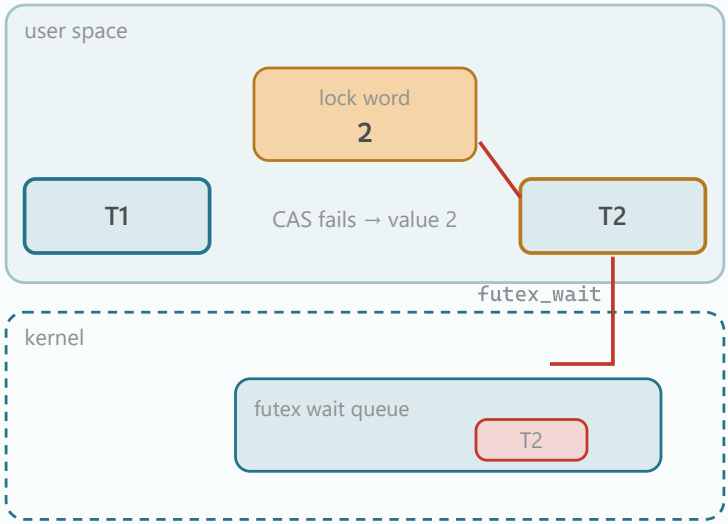



No contention: T1 flips the word 0 → 1 with one CAS; the kernel is never entered.

# Implementing a Mutex: yield and futex

```
mutex_lock:
  TSL REGISTER,MUTEX | get old, set 1
  CMP REGISTER,#0    | was it free?
  JZE ok             | yes: acquired
  CALL thread_yield  | no: give up CPU
  JMP mutex_lock     | then try again
ok: RET
mutex_unlock:
  MOVE MUTEX,#0
  RET
```

- Tanenbaum Figure 2-29: unlike `enter_region`, it calls `thread_yield` instead of looping, which avoids busy waiting with user-level threads.
- Linux **futex** (fast userspace mutex): no system call when there is no contention; glibc `pthread_mutex`, Java locks, and Go are built on it.

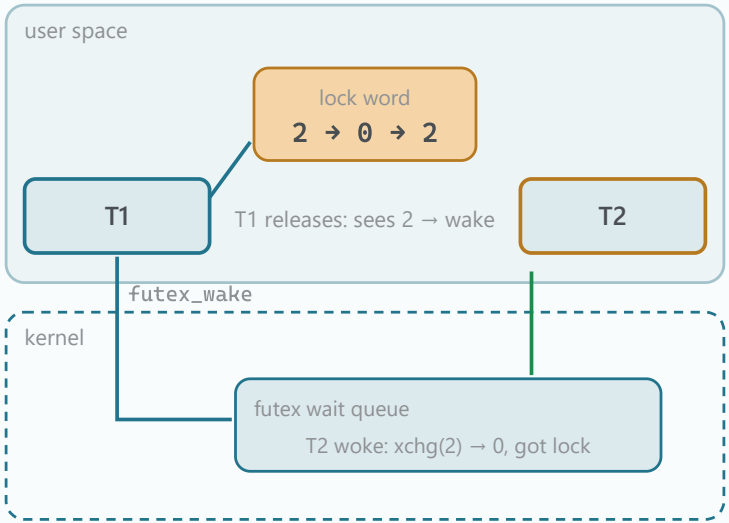



T2 finds the lock taken, sets the value to 2 ("there are waiters"), and sleeps in `futex_wait`.

# Implementing a Mutex: yield and futex

```
mutex_lock:
  TSL REGISTER,MUTEX | get old, set 1
  CMP REGISTER,#0    | was it free?
  JZE ok             | yes: acquired
  CALL thread_yield  | no: give up CPU
  JMP mutex_lock     | then try again
ok: RET
mutex_unlock:
  MOVE MUTEX,#0
  RET
```

- Tanenbaum Figure 2-29: unlike `enter_region`, it calls `thread_yield` instead of looping, which avoids busy waiting with user-level threads.
- Linux **futex** (fast userspace mutex): no system call when there is no contention; glibc `pthread_mutex`, Java locks, and Go are built on it.




On release T1 sees the old value 2, so it sets the word to 0 and calls `futex_wake`. T2 wakes and conservatively sets the word to 2 again; the old value was 0, so it holds the lock.

# A Scalable Counter: Per-CPU Counting

```
1  typedef struct {
▶    int global;      pthread_mutex_t glock;
▶    int local[NCPU]; pthread_mutex_t llock[NCPU];
▶    int S;                    /* transfer threshold */
5  } acount_t;
6
7  void add(acount_t *c, int cpu) {
8    pthread_mutex_lock(&c->llock[cpu]);
9    c->local[cpu]++;
10   if (c->local[cpu] >= c->S) {        /* transfer */
11     pthread_mutex_lock(&c->glock);
12     c->global += c->local[cpu];
13     pthread_mutex_unlock(&c->glock);
14     c->local[cpu] = 0;
15   }
16   pthread_mutex_unlock(&c->llock[cpu]);
17 }
18 int get(acount_t *c) {             /* approximate */
19   pthread_mutex_lock(&c->glock);
20   int v = c->global;
21   pthread_mutex_unlock(&c->glock);
22   return v;
23 }
```

| t | L1 | L2 | L3 | L4 | G | actual |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Four cores, threshold **S = 4**. Each core has its own counter and lock; the global counter G is updated only now and then.

Step 1 / 7

VARIABLES

`S = 4` `G = 0` `actual = 0`

With a single-lock counter every increment contends for one lock; adding cores makes it *slower*. Large S: rare trips to the global lock and good scaling, but `get()` lags further. Linux's `percpu_counter` uses the same idea.

# A Scalable Counter: Per-CPU Counting

```
1  typedef struct {
2    int global;      pthread_mutex_t glock;
3    int local[NCPU]; pthread_mutex_t llock[NCPU];
4    int S;                      /* transfer threshold */
5  } acount_t;
6
▶  void add(acount_t *c, int cpu) {
▶    pthread_mutex_lock(&c->llock[cpu]);
▶    c->local[cpu]++;
10   if (c->local[cpu] >= c->S) {        /* transfer */
11     pthread_mutex_lock(&c->glock);
12     c->global += c->local[cpu];
13     pthread_mutex_unlock(&c->glock);
14     c->local[cpu] = 0;
15   }
16   pthread_mutex_unlock(&c->llock[cpu]);
17 }
18 int get(acount_t *c) {              /* approximate */
19   pthread_mutex_lock(&c->glock);
20   int v = c->global;
21   pthread_mutex_unlock(&c->glock);
22   return v;
23 }
```

| t | L1 | L2 | L3 | L4 | G | actual |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 2 |

Cores 1 and 3 call `add` at the same time. Their locks differ, so they don't wait for each other.

Step 2 / 7

**VARIABLES**

`S = 4` `G = 0` `actual = 2`

With a single-lock counter every increment contends for one lock; adding cores makes it *slower*. Large S: rare trips to the global lock and good scaling, but `get()` lags further. Linux's `percpu_counter` uses the same idea.

# A Scalable Counter: Per-CPU Counting

```
1  typedef struct {
2    int global;      pthread_mutex_t glock;
3    int local[NCPU]; pthread_mutex_t llock[NCPU];
4    int S;                     /* transfer threshold */
5  } acount_t;
6
7  void add(acount_t *c, int cpu) {
▶    pthread_mutex_lock(&c->llock[cpu]);
▶    c->local[cpu]++;
10   if (c->local[cpu] >= c->S) {        /* transfer */
11     pthread_mutex_lock(&c->glock);
12     c->global += c->local[cpu];
13     pthread_mutex_unlock(&c->glock);
14     c->local[cpu] = 0;
15   }
16   pthread_mutex_unlock(&c->llock[cpu]);
17 }
18 int get(acount_t *c) {             /* approximate */
19   pthread_mutex_lock(&c->glock);
20   int v = c->global;
21   pthread_mutex_unlock(&c->glock);
22   return v;
23 }
```

| t | L1 | L2 | L3 | L4 | G | actual |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 2 |
| 2 | 2 | 1 | 2 | 0 | 0 | 5 |

Three more cores increment. The global lock still hasn't been touched.

Step 3 / 7

**VARIABLES**

`S = 4` `G = 0` `actual = 5`

With a single-lock counter every increment contends for one lock; adding cores makes it *slower*. Large S: rare trips to the global lock and good scaling, but `get()` lags further. Linux's `percpu_counter` uses the same idea.

# A Scalable Counter: Per-CPU Counting

```
 1 typedef struct {
 2   int global;      pthread_mutex_t glock;
 3   int local[NCPU]; pthread_mutex_t llock[NCPU];
 4   int S;                     /* transfer threshold */
 5 } acount_t;
 6
 7 void add(acount_t *c, int cpu) {
▶  pthread_mutex_lock(&c->llock[cpu]);
▶  c->local[cpu]++;
10   if (c->local[cpu] >= c->S) {        /* transfer */
11     pthread_mutex_lock(&c->glock);
12     c->global += c->local[cpu];
13     pthread_mutex_unlock(&c->glock);
14     c->local[cpu] = 0;
15   }
16   pthread_mutex_unlock(&c->llock[cpu]);
17 }
18 int get(acount_t *c) {             /* approximate */
19   pthread_mutex_lock(&c->glock);
20   int v = c->global;
21   pthread_mutex_unlock(&c->glock);
22   return v;
23 }
```

| t | L1 | L2 | L3 | L4 | G | actual |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 2 |
| 2 | 2 | 1 | 2 | 0 | 0 | 5 |
| 3 | 3 | 1 | 3 | 1 | 0 | 8 |

The actual value is 8, but G is still 0: the global counter lags behind.

Step 4 / 7

**VARIABLES**

`S = 4` `G = 0` `actual = 8`

With a single-lock counter every increment contends for one lock; adding cores makes it *slower*. Large S: rare trips to the global lock and good scaling, but `get()` lags further. Linux's `percpu_counter` uses the same idea.

# A Scalable Counter: Per-CPU Counting

```
1  typedef struct {
2    int global;      pthread_mutex_t glock;
3    int local[NCPU]; pthread_mutex_t llock[NCPU];
4    int S;                      /* transfer threshold */
5  } acount_t;
6
7  void add(acount_t *c, int cpu) {
8    pthread_mutex_lock(&c->llock[cpu]);
9    c->local[cpu]++;
▶    if (c->local[cpu] >= c->S) {        /* transfer */
▶      pthread_mutex_lock(&c->glock);
▶      c->global += c->local[cpu];
▶      pthread_mutex_lock(&c->glock);
▶      c->local[cpu] = 0;
15   }
16   pthread_mutex_unlock(&c->llock[cpu]);
17 }
18 int get(acount_t *c) {              /* approximate */
19   pthread_mutex_lock(&c->glock);
20   int v = c->global;
21   pthread_mutex_unlock(&c->glock);
22   return v;
23 }
```

| t | L1 | L2 | L3 | L4 | G | actual |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 2 |
| 2 | 2 | 1 | 2 | 0 | 0 | 5 |
| 3 | 3 | 1 | 3 | 1 | 0 | 8 |
| 4 | 4→0 | 2 | 3 | 1 | 4 | 10 |

L1 hits the threshold: core 1 takes the global lock, adds 4 to G, and resets its local counter.

Step 5 / 7

**VARIABLES**

`S = 4` `G = 4` `actual = 10`

With a single-lock counter every increment contends for one lock; adding cores makes it *slower*. Large S: rare trips to the global lock and good scaling, but `get()` lags further. Linux's `percpu_counter` uses the same idea.

# A Scalable Counter: Per-CPU Counting

```
1  typedef struct {
2    int global;      pthread_mutex_t glock;
3    int local[NCPU]; pthread_mutex_t llock[NCPU];
4    int S;                      /* transfer threshold */
5  } acount_t;
6
7  void add(acount_t *c, int cpu) {
8    pthread_mutex_lock(&c->llock[cpu]);
9    c->local[cpu]++;
▶    if (c->local[cpu] >= c->S) {        /* transfer */
▶      pthread_mutex_lock(&c->glock);
▶      c->global += c->local[cpu];
▶      pthread_mutex_unlock(&c->glock);
▶      c->local[cpu] = 0;
15   }
16   pthread_mutex_unlock(&c->llock[cpu]);
17 }
18 int get(acount_t *c) {              /* approximate */
19   pthread_mutex_lock(&c->glock);
20   int v = c->global;
21   pthread_mutex_unlock(&c->glock);
22   return v;
23 }
```

| t | L1 | L2 | L3 | L4 | G | actual |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 2 |
| 2 | 2 | 1 | 2 | 0 | 0 | 5 |
| 3 | 3 | 1 | 3 | 1 | 0 | 8 |
| 4 | 0 | 2 | 3 | 1 | 4 | 10 |
| 5 | 0 | 2 | 4→0 | 2 | 8 | 12 |

L3 also hits the threshold and transfers: G = 8.

Step 6 / 7

**VARIABLES**

S = 4 | G = 8 | actual = 12

With a single-lock counter every increment contends for one lock; adding cores makes it *slower*. Large S: rare trips to the global lock and good scaling, but `get()` lags further. Linux's `percpu_counter` uses the same idea.

# A Scalable Counter: Per-CPU Counting

```
1  typedef struct {
2    int global;      pthread_mutex_t glock;
3    int local[NCPU]; pthread_mutex_t llock[NCPU];
4    int S;                      /* transfer threshold */
5  } acount_t;
6
7  void add(acount_t *c, int cpu) {
8    pthread_mutex_lock(&c->llock[cpu]);
9    c->local[cpu]++;
10   if (c->local[cpu] >= c->S) {        /* transfer */
11     pthread_mutex_lock(&c->glock);
12     c->global += c->local[cpu];
13     pthread_mutex_unlock(&c->glock);
14     c->local[cpu] = 0;
15   }
16   pthread_mutex_unlock(&c->llock[cpu]);
17 }
▶ int get(acount_t *c) {             /* approximate */
▶   pthread_mutex_lock(&c->glock);
▶   int v = c->global;
▶   pthread_mutex_unlock(&c->glock);
▶   return v;
23 }
```

| t | L1 | L2 | L3 | L4 | G | actual |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 2 |
| 2 | 2 | 1 | 2 | 0 | 0 | 5 |
| 3 | 3 | 1 | 3 | 1 | 0 | 8 |
| 4 | 0 | 2 | 3 | 1 | 4 | 10 |
| 5 | 0 | 2 | 0 | 2 | 8 | 12 |

**get() = 8, actual = 12: the rest sits in locals**

`get()` returns 8; the actual value is 12. In the worst case the gap can reach NCPU × (S − 1) = 4 × 3 = 12.

Step 7 / 7

VARIABLES

`S = 4` `G = 8` `actual = 12`

With a single-lock counter every increment contends for one lock; adding cores makes it *slower*. Large S: rare trips to the global lock and good scaling, but `get()` lags further. Linux's `percpu_counter` uses the same idea.

# Locking Strategies for Concurrent Data Structures

| Structure | Locking | Result |
|---|---|---|
| Linked list | Single mutex | Simple. Call `malloc` outside the lock; don't forget `unlock` on every return path. |
| Hand-over-hand locking | A lock per node: grab the next, then release the previous | Traversals proceed in parallel, but the per-node locking cost often eats the gain. |
| Queue | Separate head and tail locks, a dummy node | Enqueue and dequeue can run concurrently. |
| Hash table | A locked list per bucket | Operations on different buckets don't wait for each other: scales very well. |



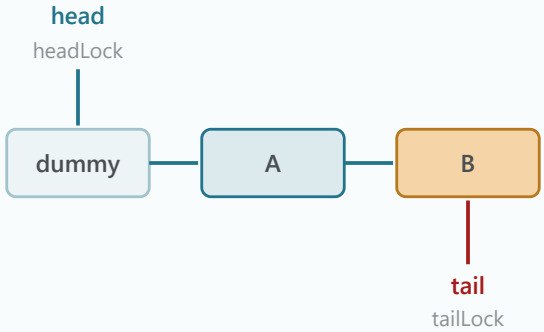

Initially there is a single **dummy** node; head and tail both point to it.

Step 1 / 3

Write the code correctly with one lock first. Split the lock only if measurement shows it is the bottleneck. More locks do not always mean faster.

# Locking Strategies for Concurrent Data Structures

| Structure | Locking | Result |
|---|---|---|
| Linked list | Single mutex | Simple. Call `malloc` outside the lock; don't forget `unlock` on every return path. |
| Hand-over-hand locking | A lock per node: grab the next, then release the previous | Traversals proceed in parallel, but the per-node locking cost often eats the gain. |
| Queue | Separate head and tail locks, a dummy node | Enqueue and dequeue can run concurrently. |
| Hash table | A locked list per bucket | Operations on different buckets don't wait for each other: scales very well. |



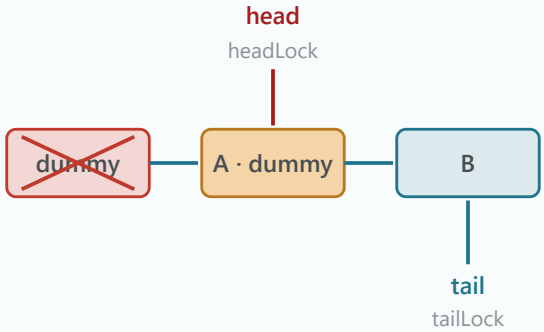

`enqueue(A)`, `enqueue(B)`: only **tailLock** is taken; the tail end advances.

Step 2 / 3

Write the code correctly with one lock first. Split the lock only if measurement shows it is the bottleneck. More locks do not always mean faster.

# Locking Strategies for Concurrent Data Structures

| Structure | Locking | Result |
|---|---|---|
| Linked list | Single mutex | Simple. Call `malloc` outside the lock; don't forget `unlock` on every return path. |
| Hand-over-hand locking | A lock per node: grab the next, then release the previous | Traversals proceed in parallel, but the per-node locking cost often eats the gain. |
| Queue | Separate head and tail locks, a dummy node | Enqueue and dequeue can run concurrently. |
| Hash table | A locked list per bucket | Operations on different buckets don't wait for each other: scales very well. |




`dequeue()` : only **headLock**. The old dummy is freed, A's value is returned, and A becomes the new dummy.

Step 3 / 3

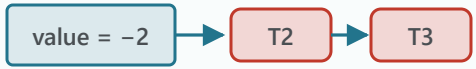
Write the code correctly with one lock first. Split the lock only if measurement shows it is the bottleneck. More locks do not always mean faster.

# Semaphore

- Dijkstra (1965): an integer accessed only through two **atomic** operations.
  - `wait(S)` = **P** = `down`: if the value is 0 the **process blocks** (sleeps); otherwise it is decremented and the process continues.
  - `signal(S)` = **V** = `up`: increment the value; if anyone is waiting, wake one.
- Think of the value as the **number of available resources**.
- In the Silberschatz implementation the value can go negative: `−k` means **k processes are waiting**.
- `wait` / `signal` are themselves short critical sections: protected by disabling interrupts on one core, or by a spinlock on multicore. Busy waiting does not vanish, but shrinks to a few instructions.

```
typedef struct {
  int value;
  struct process *list;
} semaphore;
```

A process that calls `down` on a zero value **sleeps** in Dijkstra's semaphore and in real systems; the busy-waiting version is only the first conceptual definition.




k = 2 waiting

list: sleeping processes (FIFO)

# Semaphore Waiting Queue

```
1  void wait(semaphore *S) {      /* P / down */
2    S->value--;
3    if (S->value < 0) {
4      add(S->list, this_proc);
5      sleep();                    /* block */
6    }
7  }
8  void signal(semaphore *S) {    /* V / up */
9    S->value++;
10   if (S->value <= 0) {
11     P = remove(S->list);        /* FIFO */
12     wakeup(P);
13   }
14 }
```

S.value
**1**

S.list (waiting)

empty

critical section

One resource: `S` starts at 1 (a binary semaphore, like a mutex).

Step 1 / 7

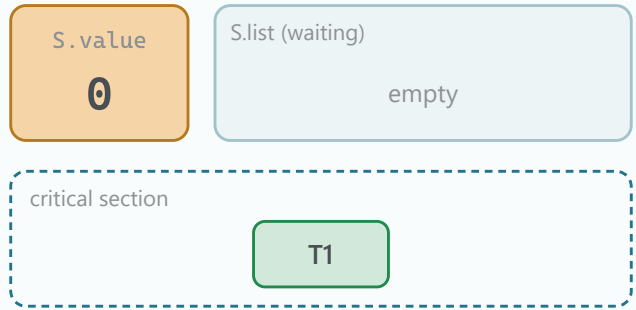
**VARIABLES**

`S.value = 1`

Silberschatz 10e §6.6.2. The definition does not fix the queue order; without FIFO, starvation is possible.

# Semaphore Waiting Queue

```
▶ void wait(semaphore *S) {      /* P / down */
▶   S->value--;
▶   if (S->value < 0) {
4     add(S->list, this_proc);
5     sleep();                    /* block */
6   }
7 }
8 void signal(semaphore *S) {    /* V / up */
9   S->value++;
10  if (S->value <= 0) {
11    P = remove(S->list);        /* FIFO */
12    wakeup(P);
13  }
14 }
```




T1 `wait` : the value becomes 0, not negative → T1 continues.

Step 2 / 7

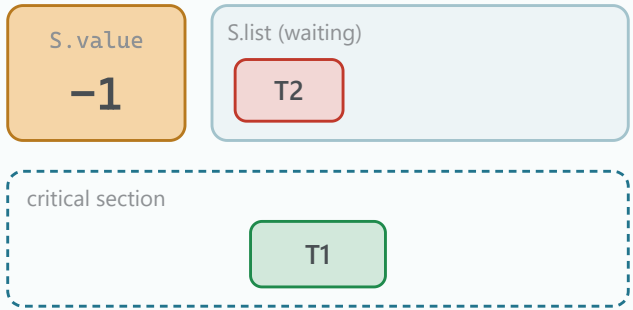
**VARIABLES**

`S.value = 0`

Silberschatz 10e §6.6.2. The definition does not fix the queue order; without FIFO, starvation is possible.

# Semaphore Waiting Queue

```
1  void wait(semaphore *S) {      /* P / down */
▶    S->value--;
▶    if (S->value < 0) {
▶      add(S->list, this_proc);
▶      sleep();                    /* block */
6    }
7  }
8  void signal(semaphore *S) {    /* V / up */
9    S->value++;
10   if (S->value <= 0) {
11     P = remove(S->list);        /* FIFO */
12     wakeup(P);
13   }
14 }
```




T2 `wait`: the value is −1 → T2 joins the queue and sleeps.

Step 3 / 7

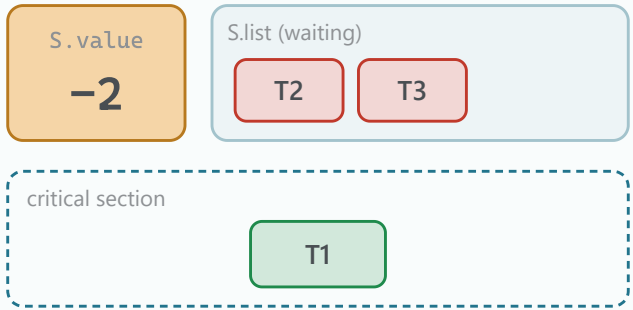
**VARIABLES**

`S.value = −1`

Silberschatz 10e §6.6.2. The definition does not fix the queue order; without FIFO, starvation is possible.

# Semaphore Waiting Queue

```
1  void wait(semaphore *S) {      /* P / down */
▶    S->value--;
▶    if (S->value < 0) {
▶      add(S->list, this_proc);
▶      sleep();                    /* block */
6    }
7  }
8  void signal(semaphore *S) {    /* V / up */
9    S->value++;
10   if (S->value <= 0) {
11     P = remove(S->list);        /* FIFO */
12     wakeup(P);
13   }
14 }
```




T3 sleeps too. `−2` : two processes are waiting.

Step 4 / 7

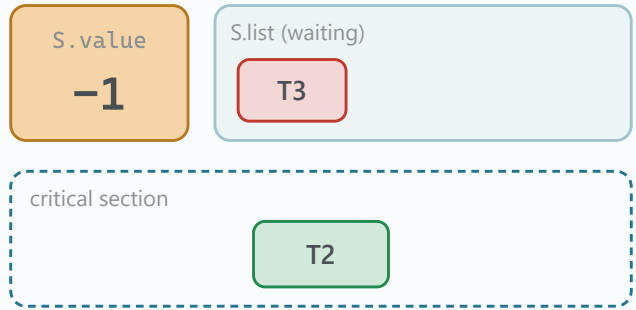
**VARIABLES**

`S.value = −2`

Silberschatz 10e §6.6.2. The definition does not fix the queue order; without FIFO, starvation is possible.

# Semaphore Waiting Queue

```
1  void wait(semaphore *S) {      /* P / down */
2    S->value--;
3    if (S->value < 0) {
4      add(S->list, this_proc);
5      sleep();                    /* block */
6    }
7  }
8  void signal(semaphore *S) {    /* V / up */
▶    S->value++;
▶    if (S->value <= 0) {
▶      P = remove(S->list);       /* FIFO */
▶      wakeup(P);
13   }
14 }
```




T1 `signal` : value −1, still ≤ 0 → T2 at the head of the queue is woken and enters.

Step 5 / 7

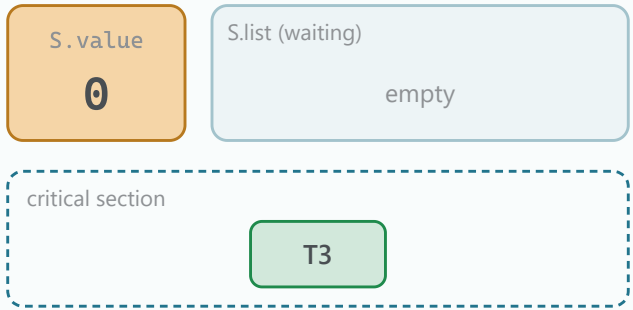
**VARIABLES**

`S.value = −1`

Silberschatz 10e §6.6.2. The definition does not fix the queue order; without FIFO, starvation is possible.

# Semaphore Waiting Queue

```
1  void wait(semaphore *S) {      /* P / down */
2    S->value--;
3    if (S->value < 0) {
4      add(S->list, this_proc);
5      sleep();                    /* block */
6    }
7  }
8  void signal(semaphore *S) {    /* V / up */
▶    S->value++;
▶    if (S->value <= 0) {
▶      P = remove(S->list);       /* FIFO */
▶      wakeup(P);
13   }
14 }
```




T2 `signal` : value 0 → T3 is woken.

Step 6 / 7

**VARIABLES**

`S.value = 0`

Silberschatz 10e §6.6.2. The definition does not fix the queue order; without FIFO, starvation is possible.

# Semaphore Waiting Queue

```
1  void wait(semaphore *S) {      /* P / down */
2    S->value--;
3    if (S->value < 0) {
4      add(S->list, this_proc);
5      sleep();                    /* block */
6    }
7  }
8  void signal(semaphore *S) {    /* V / up */
▶    S->value++;
▶    if (S->value <= 0) {
11     P = remove(S->list);        /* FIFO */
12     wakeup(P);
13   }
14 }
```

S.value
**1**

S.list (waiting)

empty

critical section

T3 `signal`: value 1, no one waiting. The queue is FIFO, so waiting is bounded.

Step 7 / 7

**VARIABLES**

`S.value = 1`

Silberschatz 10e §6.6.2. The definition does not fix the queue order; without FIFO, starvation is possible.

# Semaphore Usage Patterns

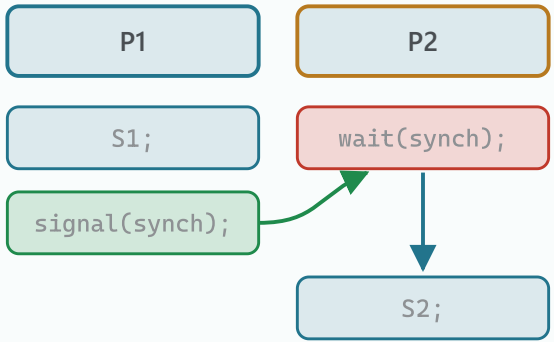
**Binary (mutex-like)**
Initial value **1**: `wait` … CS … `signal`.

**Counting**
Initial value **N**: N identical resources (a connection pool, buffer slots).

**Signaling / ordering**
Initial value **0**: wait until an event happens (figure on the right).

## Common mistakes

| | |
|---|---|
| `signal(m)` … CS … `wait(m)` | Reversed: several processes in the CS |
| `wait(m)` … CS … `wait(m)` | Waits on itself: blocked forever |
| A forgotten `wait` or `signal` | Broken exclusion, or deadlock |
| Two semaphores taken in different orders | Deadlock |

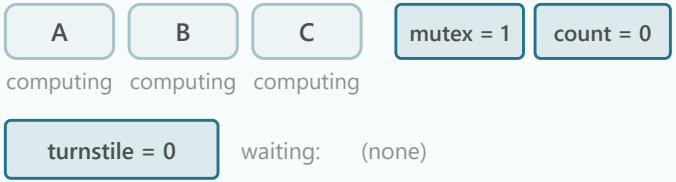



synch starts at 0

Semaphores are powerful but *unstructured*: correctness depends on every `wait` / `signal` pair. That is why monitors were developed.

# Barrier with Semaphores: Code Trace

```
1  #define N 3
▶  semaphore mutex = 1, turnstile = 0;
▶  int count = 0;
4  void barrier(void) {
5    down(&mutex);
6    count = count + 1;
7    if (count == N) up(&turnstile); /* last */
8    up(&mutex);
9    down(&turnstile);  /* pass through */
10   up(&turnstile);    /* let next one in */
11 }
```




Three threads compute a phase. The turnstile is closed (0); `count` counts arrivals.

Step 1 / 10

**VARIABLES**

`mutex = 1` `turnstile = 0` `count = 0` `queue = []`

Tanenbaum semaphore: if someone is sleeping, `up` wakes one without incrementing. This barrier is single-use; a reusable one needs two turnstiles.

# Barrier with Semaphores: Code Trace

```
1  #define N 3
2  semaphore mutex = 1, turnstile = 0;
3  int count = 0;
4  void barrier(void) {
▶    down(&mutex);
▶    count = count + 1;
7    if (count == N) up(&turnstile); /* last */
8    up(&mutex);
9    down(&turnstile);  /* pass through */
10   up(&turnstile);    /* let next one in */
11 }
```

A
active

B
computing

C
computing

mutex = 0

count = 1

turnstile = 0

waiting: (none)

A is done: it takes the mutex, `count = 1`.

Step 2 / 10

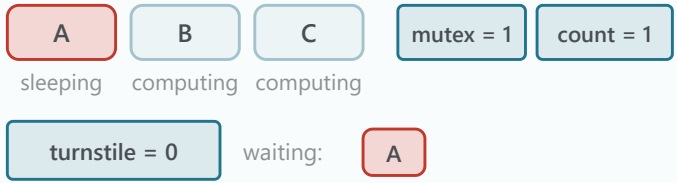
**VARIABLES**

`mutex = 0` `turnstile = 0` `count = 1` `queue = []`

Tanenbaum semaphore: if someone is sleeping, `up` wakes one without incrementing. This barrier is single-use; a reusable one needs two turnstiles.

# Barrier with Semaphores: Code Trace

```
1  #define N 3
2  semaphore mutex = 1, turnstile = 0;
3  int count = 0;
4  void barrier(void) {
5    down(&mutex);
6    count = count + 1;
7    if (count == N) up(&turnstile); /* last */
▶    up(&mutex);
▶    down(&turnstile);  /* pass through */
10   up(&turnstile);    /* let next one in */
11 }
```




1 ≠ 3: A releases the mutex and **sleeps** in `down(&turnstile)`.

Step 3 / 10

**VARIABLES**

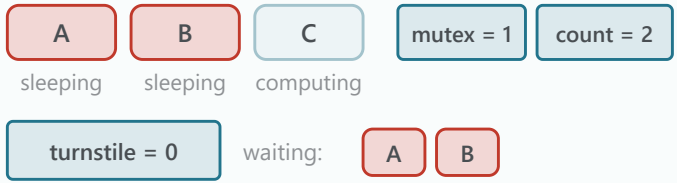
`mutex = 1` `turnstile = 0` `count = 1`

`queue = [A]`

Tanenbaum semaphore: if someone is sleeping, `up` wakes one without incrementing. This barrier is single-use; a reusable one needs two turnstiles.

# Barrier with Semaphores: Code Trace

```
1  #define N 3
2  semaphore mutex = 1, turnstile = 0;
3  int count = 0;
4  void barrier(void) {
▶    down(&mutex);
6    count = count + 1;
7    if (count == N) up(&turnstile); /* last */
▶    up(&mutex);
▶    down(&turnstile);  /* pass through */
10   up(&turnstile);    /* let next one in */
11 }
```




B arrives too: `count = 2`, sleeps at the turnstile.

Step 4 / 10

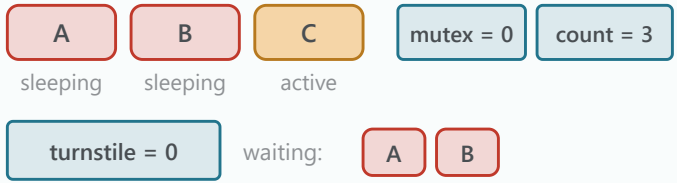
**VARIABLES**

`mutex = 1` `turnstile = 0` `count = 2` `queue = [A, B]`

Tanenbaum semaphore: if someone is sleeping, `up` wakes one without incrementing. This barrier is single-use; a reusable one needs two turnstiles.

# Barrier with Semaphores: Code Trace

```
1  #define N 3
2  semaphore mutex = 1, turnstile = 0;
3  int count = 0;
4  void barrier(void) {
▶    down(&mutex);
▶    count = count + 1;
7    if (count == N) up(&turnstile); /* last */
8    up(&mutex);
9    down(&turnstile);  /* pass through */
10   up(&turnstile);    /* let next one in */
11 }
```




The slowest, C, arrives: `count = 3 = N`.

Step 5 / 10

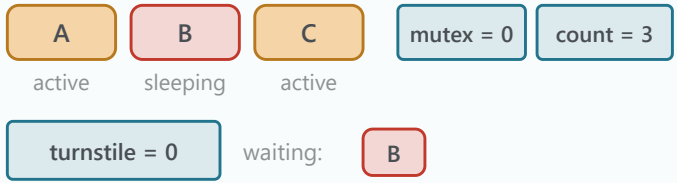
**VARIABLES**

`mutex = 0` `turnstile = 0` `count = 3` `queue = [A, B]`

Tanenbaum semaphore: if someone is sleeping, `up` wakes one without incrementing. This barrier is single-use; a reusable one needs two turnstiles.

# Barrier with Semaphores: Code Trace

```
1  #define N 3
2  semaphore mutex = 1, turnstile = 0;
3  int count = 0;
4  void barrier(void) {
5    down(&mutex);
6    count = count + 1;
▶    if (count == N) up(&turnstile); /* last */
8    up(&mutex);
9    down(&turnstile);  /* pass through */
10   up(&turnstile);    /* let next one in */
11 }
```




C, the last to arrive, calls `up(&turnstile)` : since someone is waiting the value stays put and **A is woken**.

Step 6 / 10

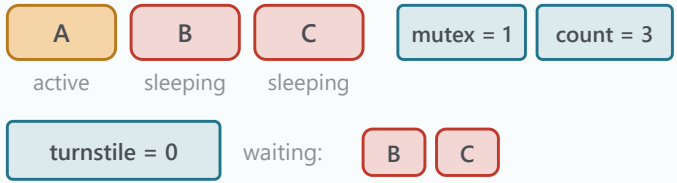
**VARIABLES**

`mutex = 0` `turnstile = 0` `count = 3` `queue = [B]`

Tanenbaum semaphore: if someone is sleeping, `up` wakes one without incrementing. This barrier is single-use; a reusable one needs two turnstiles.

# Barrier with Semaphores: Code Trace

```
1  #define N 3
2  semaphore mutex = 1, turnstile = 0;
3  int count = 0;
4  void barrier(void) {
5    down(&mutex);
6    count = count + 1;
7    if (count == N) up(&turnstile); /* last */
▶    up(&mutex);
▶    down(&turnstile);  /* pass through */
10   up(&turnstile);    /* let next one in */
11 }
```




C releases the mutex and sleeps at the turnstile: it can pass only if someone lets it through.

Step 7 / 10

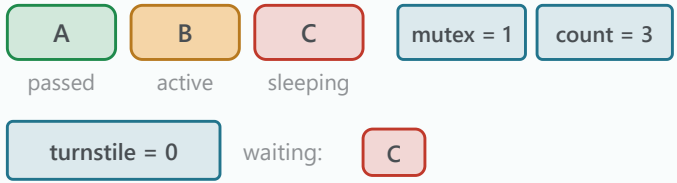
**VARIABLES**

`mutex = 1` `turnstile = 0` `count = 3` `queue = [B, C]`

Tanenbaum semaphore: if someone is sleeping, `up` wakes one without incrementing. This barrier is single-use; a reusable one needs two turnstiles.

# Barrier with Semaphores: Code Trace

```
1  #define N 3
2  semaphore mutex = 1, turnstile = 0;
3  int count = 0;
4  void barrier(void) {
5    down(&mutex);
6    count = count + 1;
7    if (count == N) up(&turnstile); /* last */
8    up(&mutex);
9    down(&turnstile);  /* pass through */
▶    up(&turnstile);    /* let next one in */
11 }
```




A is through; with `up(&turnstile)` it wakes **B** behind it.

Step 8 / 10

**VARIABLES**

`mutex = 1` `turnstile = 0` `count = 3` `queue = [C]`

Tanenbaum semaphore: if someone is sleeping, `up` wakes one without incrementing. This barrier is single-use; a reusable one needs two turnstiles.

# Barrier with Semaphores: Code Trace

```
1  #define N 3
2  semaphore mutex = 1, turnstile = 0;
3  int count = 0;
4  void barrier(void) {
5    down(&mutex);
6    count = count + 1;
7    if (count == N) up(&turnstile); /* last */
8    up(&mutex);
9    down(&turnstile);  /* pass through */
▶    up(&turnstile);    /* let next one in */
11 }
```

A (passed) | B (passed) | C (active) | mutex = 1 | count = 3

turnstile = 0 | waiting: (none)

B passes and wakes C: the turnstile works like a one-person revolving door.

Step 9 / 10

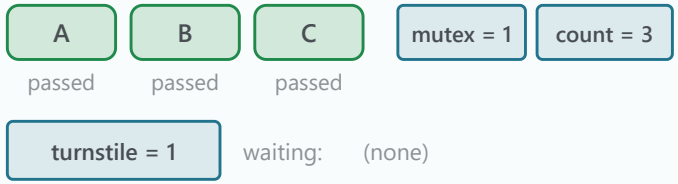
**VARIABLES**

`mutex = 1` `turnstile = 0` `count = 3` `queue = []`

Tanenbaum semaphore: if someone is sleeping, `up` wakes one without incrementing. This barrier is single-use; a reusable one needs two turnstiles.

# Barrier with Semaphores: Code Trace

```
1  #define N 3
2  semaphore mutex = 1, turnstile = 0;
3  int count = 0;
4  void barrier(void) {
5    down(&mutex);
6    count = count + 1;
7    if (count == N) up(&turnstile); /* last */
8    up(&mutex);
9    down(&turnstile);  /* pass through */
▶    up(&turnstile);    /* let next one in */
11 }
```




C passes; with no one left waiting, `turnstile = 1`. All have moved to the next phase; to reuse the barrier the turnstile must be closed again.

Step 10 / 10

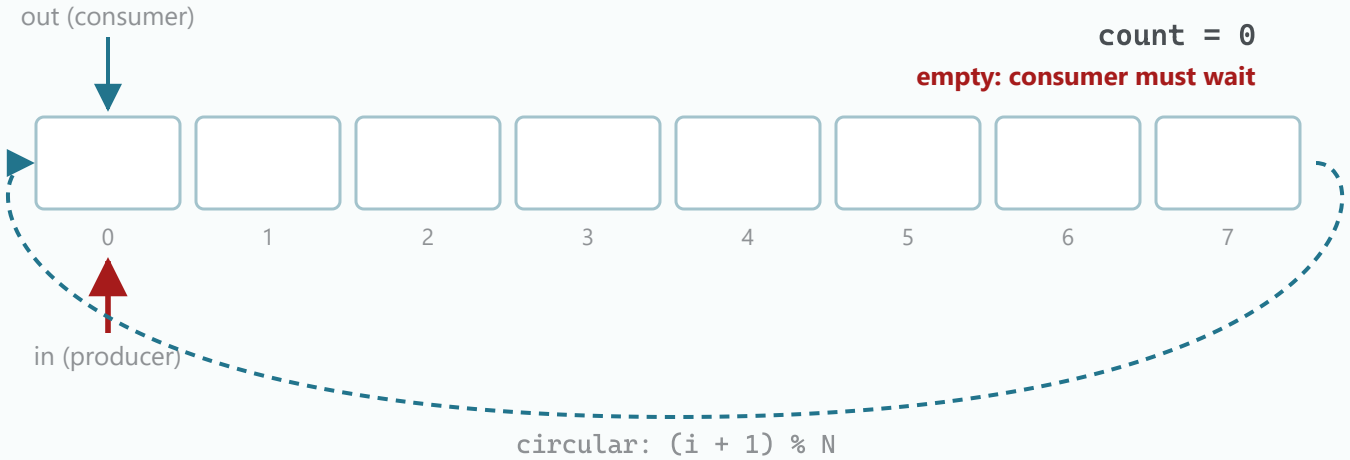
**VARIABLES**

`mutex = 1` `turnstile = 1` `count = 3` `queue = []`

Tanenbaum semaphore: if someone is sleeping, `up` wakes one without incrementing. This barrier is single-use; a reusable one needs two turnstiles.

# The Producer–Consumer (Bounded-Buffer) Problem

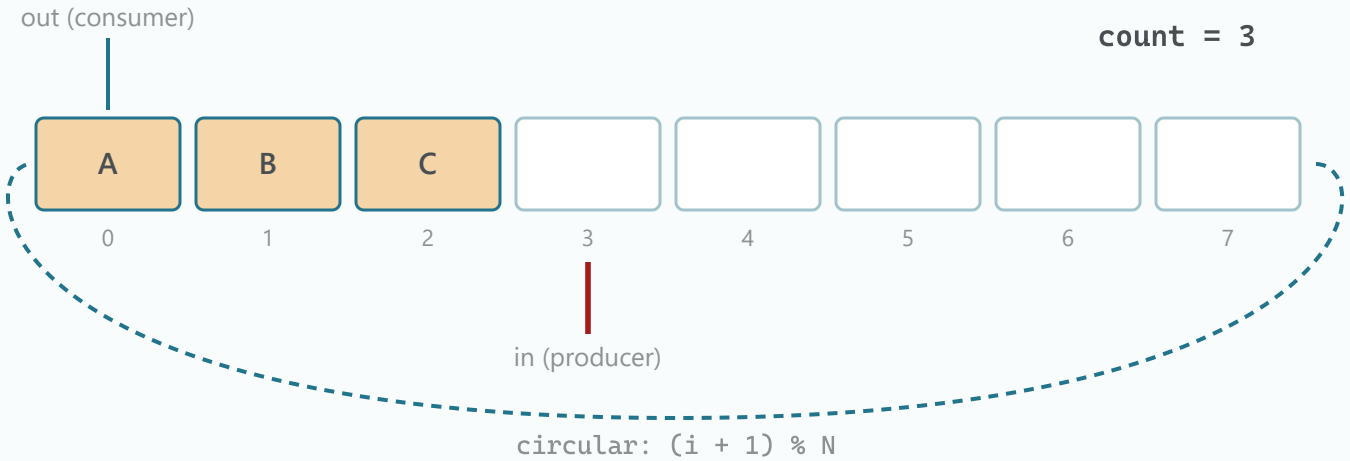
- Two processes share a fixed-size buffer of N slots: the **producer** writes, the **consumer** reads.
- **Bounded buffer:** the producer waits when it is full, the consumer when it is empty. **Unbounded buffer:** only the consumer waits.




The buffer is empty (count = 0): the consumer cannot read and must wait.

# The Producer–Consumer (Bounded-Buffer) Problem

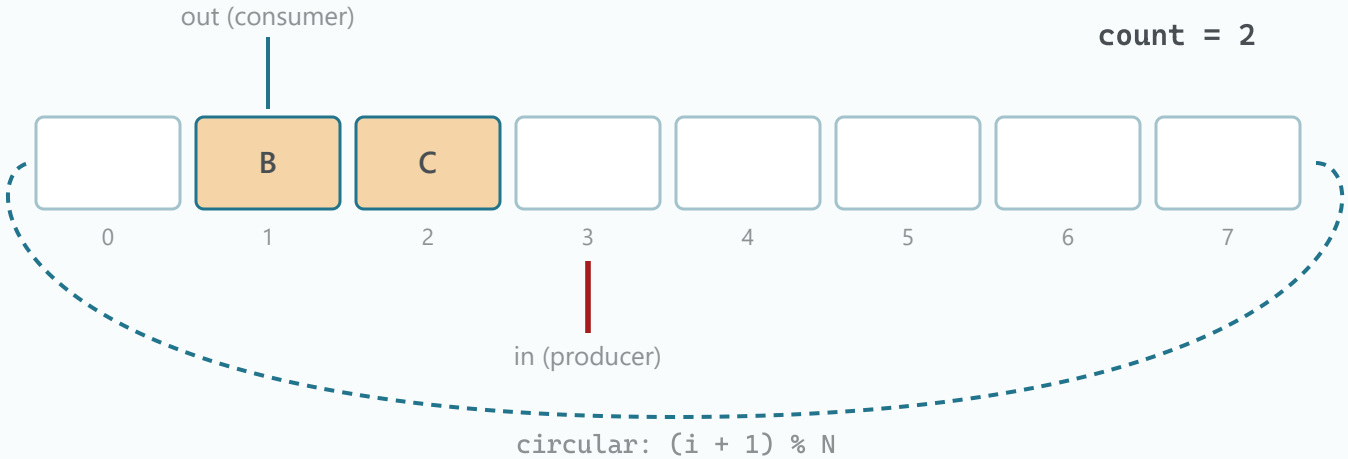
- Two processes share a fixed-size buffer of N slots: the **producer** writes, the **consumer** reads.
- **Bounded buffer:** the producer waits when it is full, the consumer when it is empty. **Unbounded buffer:** only the consumer waits.




The producer adds A, B, C: each insert does `in = (in + 1) % N`.

# The Producer–Consumer (Bounded-Buffer) Problem

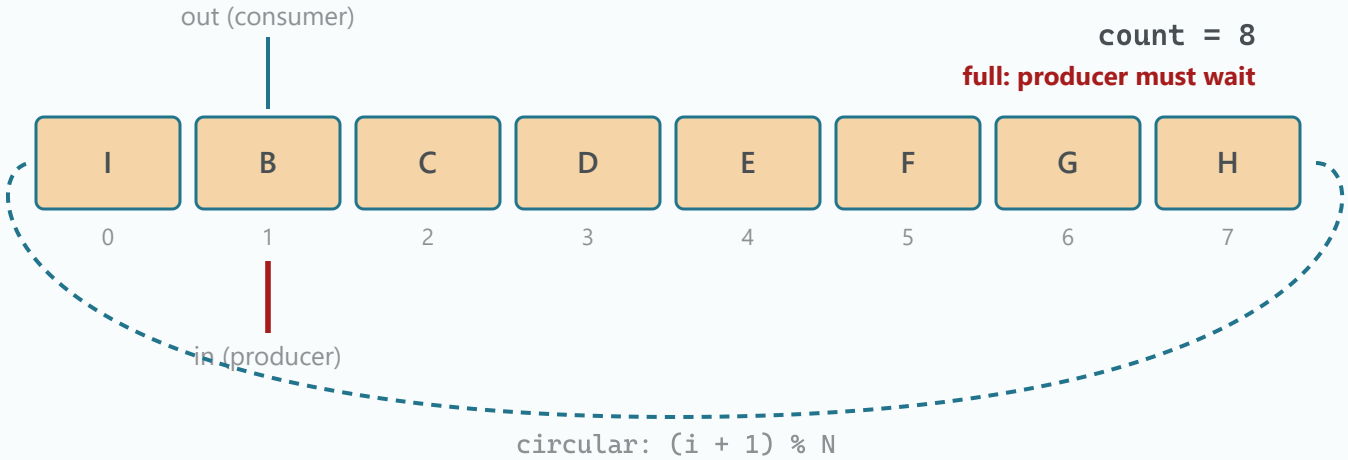
- Two processes share a fixed-size buffer of N slots: the **producer** writes, the **consumer** reads.
- **Bounded buffer:** the producer waits when it is full, the consumer when it is empty. **Unbounded buffer:** only the consumer waits.




The consumer takes A: `out = 1`, count = 2.

# The Producer–Consumer (Bounded-Buffer) Problem

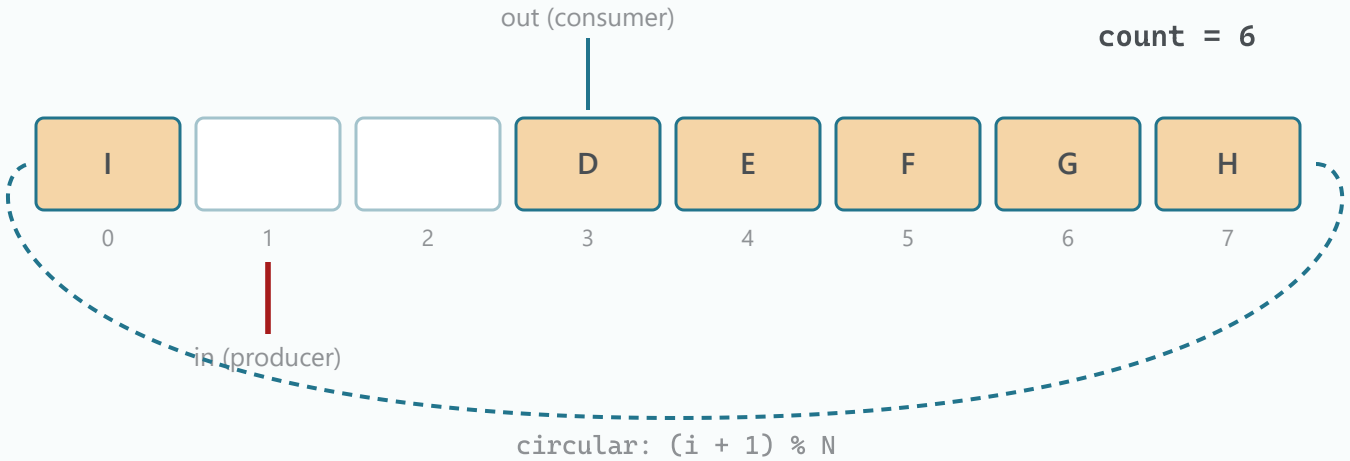
- Two processes share a fixed-size buffer of N slots: the **producer** writes, the **consumer** reads.
- **Bounded buffer:** the producer waits when it is full, the consumer when it is empty. **Unbounded buffer:** only the consumer waits.




The producer adds D…I; `in` wraps around (1). count = N = 8: the buffer is full and the producer must wait.

# The Producer–Consumer (Bounded-Buffer) Problem

- Two processes share a fixed-size buffer of N slots: the **producer** writes, the **consumer** reads.
- **Bounded buffer:** the producer waits when it is full, the consumer when it is empty. **Unbounded buffer:** only the consumer waits.




The consumer takes B and C: `out = 3`, count = 6; the producer can write again.

# A Fatal Race: The Lost Wakeup

```
1  #define N 100
2  int count = 0;
3  void producer(void) {
4      while (TRUE) {
5          item = produce_item();
6          if (count == N) sleep();
7          insert_item(item);
8          count = count + 1;
9          if (count == 1) wakeup(consumer);
10     }
11 }
12 void consumer(void) {
13     while (TRUE) {
▶          if (count == 0) sleep();
15         item = remove_item();
16         count = count - 1;
17         if (count == N - 1) wakeup(producer);
18         consume_item(item);
19     }
20 }
```

**VARIABLES**

`count = 0` `consumer = about to sleep` `producer = ready`




**`count = 0`**

The buffer is empty. The consumer reads `count`: 0. Just before calling `sleep()` it is interrupted.

Step 1 / 7

Tanenbaum MOS Figure 2-27. `sleep()` blocks the caller (no busy waiting); `wakeup(p)` makes a sleeping `p` ready. A semaphore stores wakeups as a count, so no signal is lost.

# A Fatal Race: The Lost Wakeup

```
1  #define N 100
2  int count = 0;
3  void producer(void) {
4      while (TRUE) {
▶          item = produce_item();
6          if (count == N) sleep();
▶          insert_item(item);
8          count = count + 1;
9          if (count == 1) wakeup(consumer);
10     }
11 }
12 void consumer(void) {
13     while (TRUE) {
14         if (count == 0) sleep();
15         item = remove_item();
16         count = count - 1;
17         if (count == N - 1) wakeup(producer);
18         consume_item(item);
19     }
20 }
```

VARIABLES

`count = 0` `consumer = about to sleep` `producer = running`




**`count = 0`**

The producer produces an item and puts it in the buffer.

Step 2 / 7

Tanenbaum MOS Figure 2-27. `sleep()` blocks the caller (no busy waiting); `wakeup(p)` makes a sleeping `p` ready. A semaphore stores wakeups as a count, so no signal is lost.

# A Fatal Race: The Lost Wakeup

```
1  #define N 100
2  int count = 0;
3  void producer(void) {
4      while (TRUE) {
5          item = produce_item();
6          if (count == N) sleep();
7          insert_item(item);
▶          count = count + 1;
9          if (count == 1) wakeup(consumer);
10     }
11 }
12 void consumer(void) {
13     while (TRUE) {
14         if (count == 0) sleep();
15         item = remove_item();
16         count = count - 1;
17         if (count == N - 1) wakeup(producer);
18         consume_item(item);
19     }
20 }
```

VARIABLES

`count = 1` `consumer = about to sleep` `producer = running`




`count = 1`

`count` becomes 1.

Step 3 / 7

Tanenbaum MOS Figure 2-27. `sleep()` blocks the caller (no busy waiting); `wakeup(p)` makes a sleeping `p` ready. A semaphore stores wakeups as a count, so no signal is lost.

# A Fatal Race: The Lost Wakeup

```
1  #define N 100
2  int count = 0;
3  void producer(void) {
4      while (TRUE) {
5          item = produce_item();
6          if (count == N) sleep();
7          insert_item(item);
8          count = count + 1;
▶          if (count == 1) wakeup(consumer);
10     }
11 }
12 void consumer(void) {
13     while (TRUE) {
14         if (count == 0) sleep();
15         item = remove_item();
16         count = count - 1;
17         if (count == N - 1) wakeup(producer);
18         consume_item(item);
19     }
20 }
```

VARIABLES

`count = 1` `consumer = about to sleep` `producer = running` `wakeup = lost`

**producer**
wakeup → lost

**consumer**
interrupted

**`count = 1`**

count == 1, so the producer assumes the consumer is asleep and sends `wakeup(consumer)`. But the consumer isn't asleep yet: the signal is **lost**.

Step 4 / 7

Tanenbaum MOS Figure 2-27. `sleep()` blocks the caller (no busy waiting); `wakeup(p)` makes a sleeping `p` ready. A semaphore stores wakeups as a count, so no signal is lost.

# A Fatal Race: The Lost Wakeup

```
1  #define N 100
2  int count = 0;
3  void producer(void) {
4      while (TRUE) {
5          item = produce_item();
6          if (count == N) sleep();
7          insert_item(item);
8          count = count + 1;
9          if (count == 1) wakeup(consumer);
10     }
11 }
12 void consumer(void) {
13     while (TRUE) {
▶          if (count == 0) sleep();
15         item = remove_item();
16         count = count - 1;
17         if (count == N - 1) wakeup(producer);
18         consume_item(item);
19     }
20 }
```

VARIABLES

`count = 1` `consumer = sleeping` `producer = ready`




**`count = 1`**

The consumer resumes: acting on its stale decision, it calls `sleep()` and goes to sleep.

Step 5 / 7

Tanenbaum MOS Figure 2-27. `sleep()` blocks the caller (no busy waiting); `wakeup(p)` makes a sleeping `p` ready. A semaphore stores wakeups as a count, so no signal is lost.

# A Fatal Race: The Lost Wakeup

```
1  #define N 100
2  int count = 0;
3  void producer(void) {
4      while (TRUE) {
5          item = produce_item();
6          if (count == N) sleep();
7          insert_item(item);
▶          count = count + 1;
9          if (count == 1) wakeup(consumer);
10     }
11 }
12 void consumer(void) {
13     while (TRUE) {
14         if (count == 0) sleep();
15         item = remove_item();
16         count = count - 1;
17         if (count == N - 1) wakeup(producer);
18         consume_item(item);
19     }
20 }
```

VARIABLES

`count = 100` `consumer = sleeping` `producer = running`




`count = 100`

The producer fills the buffer; the consumer is asleep, so no one reads.

Step 6 / 7

Tanenbaum MOS Figure 2-27. `sleep()` blocks the caller (no busy waiting); `wakeup(p)` makes a sleeping `p` ready. A semaphore stores wakeups as a count, so no signal is lost.

# A Fatal Race: The Lost Wakeup

```
1  #define N 100
2  int count = 0;
3  void producer(void) {
4      while (TRUE) {
5          item = produce_item();
▶          if (count == N) sleep();
7          insert_item(item);
8          count = count + 1;
9          if (count == 1) wakeup(consumer);
10     }
11 }
12 void consumer(void) {
13     while (TRUE) {
14         if (count == 0) sleep();
15         item = remove_item();
16         count = count - 1;
17         if (count == N - 1) wakeup(producer);
18         consume_item(item);
19     }
20 }
```

VARIABLES

`count = 100` `consumer = sleeping` `producer = sleeping`

| producer | consumer |
|---|---|
| sleeping | sleeping |

`count = 100`

count == N → the producer sleeps too. **Both sleep forever.** The problem: testing `count` and calling `sleep` are not one indivisible (atomic) step.

Step 7 / 7

Tanenbaum MOS Figure 2-27. `sleep()` blocks the caller (no busy waiting); `wakeup(p)` makes a sleeping `p` ready. A semaphore stores wakeups as a count, so no signal is lost.

# Bounded Buffer with Semaphores

```
#define N 100
semaphore mutex = 1;  /* buffer access      */
semaphore empty = N;  /* empty slots        */
semaphore full  = 0;  /* full slots         */

void producer(void) {
  while (true) {
    item = produce_item();
    down(&empty);    /* wait for a slot  */
    down(&mutex);    /* enter CS         */
    insert_item(item);
    up(&mutex);      /* leave CS         */
    up(&full);       /* one more full    */
  }
}
```

```
void consumer(void) {
  while (true) {
    down(&full);     /* wait for an item */
    down(&mutex);
    item = remove_item();
    up(&mutex);
    up(&empty);      /* one more empty   */
    consume_item(item);
  }
}
```

**mutex = 1**

Mutual exclusion. (Starting at 0, no one could ever enter.)

**empty = N**

The producer blocks when the buffer is full.

**full = 0**

The consumer blocks when the buffer is empty: an *ordering* (synchronization) role.

# Bounded Buffer, Step by Step

```
▶ semaphore mutex = 1, empty = 2, full = 0;
2
3  void producer(void) {
4    while (true) {
5      item = produce();
6      wait(&empty);
7      wait(&mutex);
8      insert(item);
9      signal(&mutex);
10     signal(&full);
11   }
12 }
13 void consumer(void) {
14   while (true) {
15     wait(&full);
16     wait(&mutex);
17     item = remove();
18     signal(&mutex);
19     signal(&empty);
20     consume(item);
21   }
22 }
```

P — starting

C — starting

buffer (N = 2)



The buffer is empty: 2 empty slots, 0 full.

Step 1 / 8

**VARIABLES**

`mutex = 1` `empty = 2` `full = 0`

# Bounded Buffer, Step by Step

```
1  semaphore mutex = 1, empty = 2, full = 0;
2
3  void producer(void) {
4    while (true) {
5      item = produce();
6      wait(&empty);
7      wait(&mutex);
8      insert(item);
9      signal(&mutex);
10     signal(&full);
11   }
12 }
13 void consumer(void) {
14   while (true) {
▶      wait(&full);
16     wait(&mutex);
17     item = remove();
18     signal(&mutex);
19     signal(&empty);
20     consume(item);
21   }
22 }
```

P — ready

C — wait(full) → blocked

buffer (N = 2)



The consumer runs first: `full = 0` → it **sleeps**. No busy waiting.

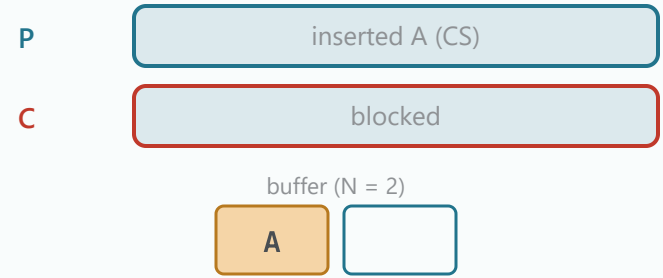
Step 2 / 8

**VARIABLES**

`mutex = 1` `empty = 2` `full = 0`

# Bounded Buffer, Step by Step

```
1  semaphore mutex = 1, empty = 2, full = 0;
2
3  void producer(void) {
4    while (true) {
5      item = produce();
▶      wait(&empty);
▶      wait(&mutex);
▶      insert(item);
9      signal(&mutex);
10     signal(&full);
11   }
12 }
13 void consumer(void) {
14   while (true) {
15     wait(&full);
16     wait(&mutex);
17     item = remove();
18     signal(&mutex);
19     signal(&empty);
20     consume(item);
21   }
22 }
```




Producer: `empty` 2→1, `mutex` 1→0, inserts A.

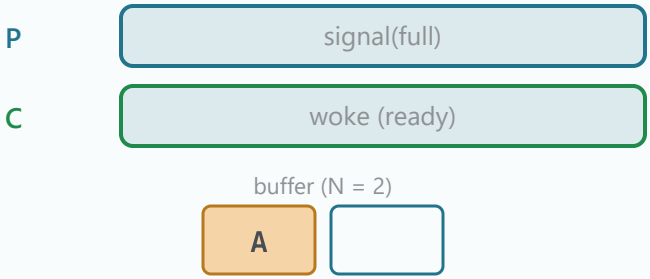
Step 3 / 8

**VARIABLES**

`mutex = 0` `empty = 1` `full = 0`

# Bounded Buffer, Step by Step

```
1  semaphore mutex = 1, empty = 2, full = 0;
2
3  void producer(void) {
4    while (true) {
5      item = produce();
6      wait(&empty);
7      wait(&mutex);
8      insert(item);
▶      signal(&mutex);
▶      signal(&full);
11   }
12 }
13 void consumer(void) {
14   while (true) {
15     wait(&full);
16     wait(&mutex);
17     item = remove();
18     signal(&mutex);
19     signal(&empty);
20     consume(item);
21   }
22 }
```




`signal(full)` wakes the waiting consumer; the woken process uses up that "1", so `full` stays 0.

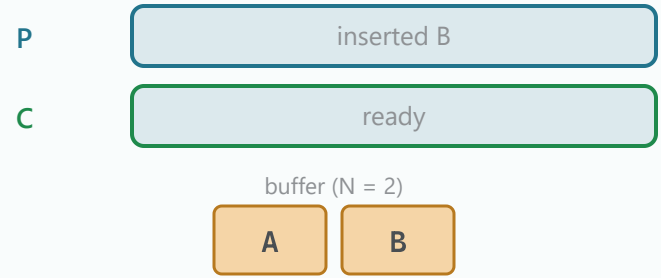
Step 4 / 8

**VARIABLES**

`mutex = 1` `empty = 1` `full = 0`

# Bounded Buffer, Step by Step

```
1  semaphore mutex = 1, empty = 2, full = 0;
2
3  void producer(void) {
4    while (true) {
5      item = produce();
▶      wait(&empty);
▶      wait(&mutex);
▶      insert(item);
▶      signal(&mutex);
▶      signal(&full);
11   }
12 }
13 void consumer(void) {
14   while (true) {
15     wait(&full);
16     wait(&mutex);
17     item = remove();
18     signal(&mutex);
19     signal(&empty);
20     consume(item);
21   }
22 }
```




The consumer hasn't run yet; the producer inserts B too: `empty = 0` , `full = 1` .

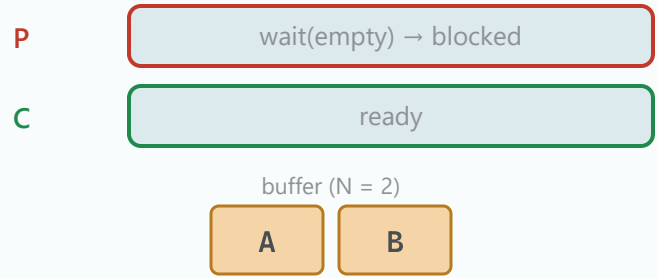
Step 5 / 8

**VARIABLES**

`mutex = 1` `empty = 0` `full = 1`

# Bounded Buffer, Step by Step

```
1  semaphore mutex = 1, empty = 2, full = 0;
2
3  void producer(void) {
4    while (true) {
5      item = produce();
▶      wait(&empty);
7      wait(&mutex);
8      insert(item);
9      signal(&mutex);
10     signal(&full);
11   }
12 }
13 void consumer(void) {
14   while (true) {
15     wait(&full);
16     wait(&mutex);
17     item = remove();
18     signal(&mutex);
19     signal(&empty);
20     consume(item);
21   }
22 }
```




No empty slot for C: the producer **sleeps**. The buffer does not overflow.

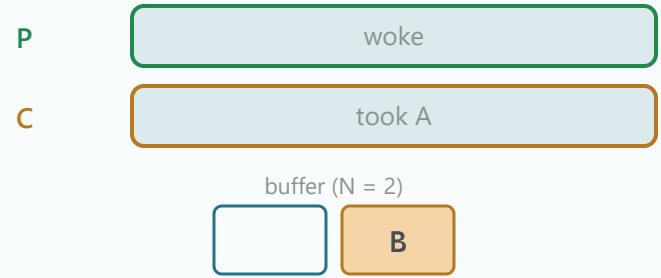
Step 6 / 8

**VARIABLES**

`mutex = 1` `empty = 0` `full = 1`

# Bounded Buffer, Step by Step

```
1  semaphore mutex = 1, empty = 2, full = 0;
2
3  void producer(void) {
4    while (true) {
5      item = produce();
6      wait(&empty);
7      wait(&mutex);
8      insert(item);
9      signal(&mutex);
10     signal(&full);
11   }
12 }
13 void consumer(void) {
14   while (true) {
15     wait(&full);
▶      wait(&mutex);
▶      item = remove();
▶      signal(&mutex);
▶      signal(&empty);
20     consume(item);
21   }
22 }
```




The consumer takes A; `signal(empty)` wakes the sleeping producer.

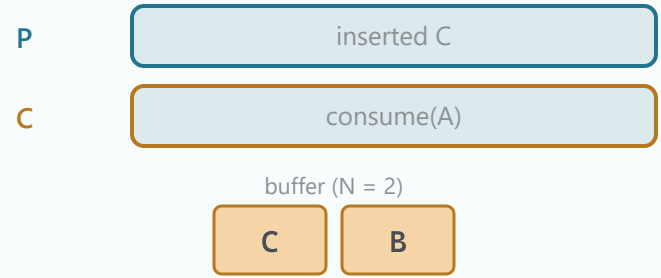
Step 7 / 8

**VARIABLES**

`mutex = 1` `empty = 0` `full = 1`

# Bounded Buffer, Step by Step

```
1  semaphore mutex = 1, empty = 2, full = 0;
2
3  void producer(void) {
4    while (true) {
5      item = produce();
6      wait(&empty);
▶      wait(&mutex);
▶      insert(item);
▶      signal(&mutex);
▶      signal(&full);
11   }
12 }
13 void consumer(void) {
14   while (true) {
15     wait(&full);
16     wait(&mutex);
17     item = remove();
18     signal(&mutex);
19     signal(&empty);
20     consume(item);
21   }
22 }
```




The producer puts C in the freed slot. At all times: `empty + full` + operations in progress = N.

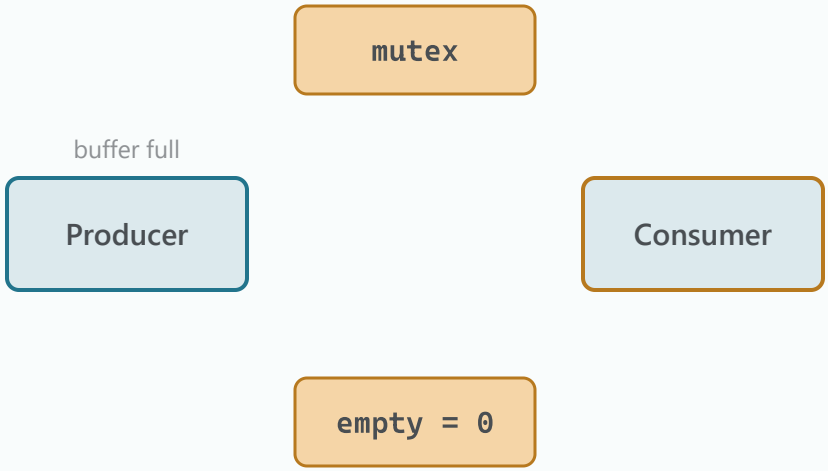
Step 8 / 8

**VARIABLES**

`mutex = 1` `empty = 0` `full = 2`

# If the downs Are Swapped

```
void producer(void) {
  while (true) {
    item = produce_item();
    down(&mutex);  /* ← first */
    down(&empty);  /* ← then  */
    insert_item(item);
    up(&mutex);
    up(&full);
  }
}
```

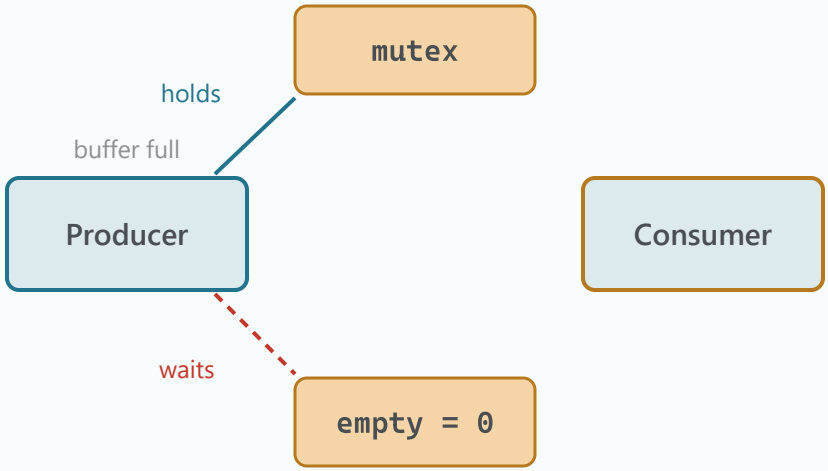



The producer's two `down` calls are swapped. The buffer is completely full: `empty = 0`.

Step 1 / 4

# If the downs Are Swapped

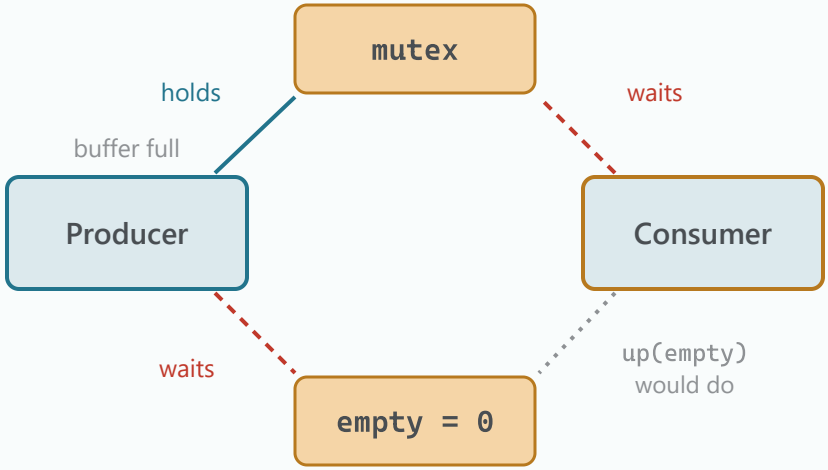
```
void producer(void) {
  while (true) {
    item = produce_item();
    down(&mutex);  /* ← first */
    down(&empty);  /* ← then  */
    insert_item(item);
    up(&mutex);
    up(&full);
  }
}
```




The producer takes `mutex`, then sleeps in `down(&empty)`, **still holding the mutex**.

# If the downs Are Swapped

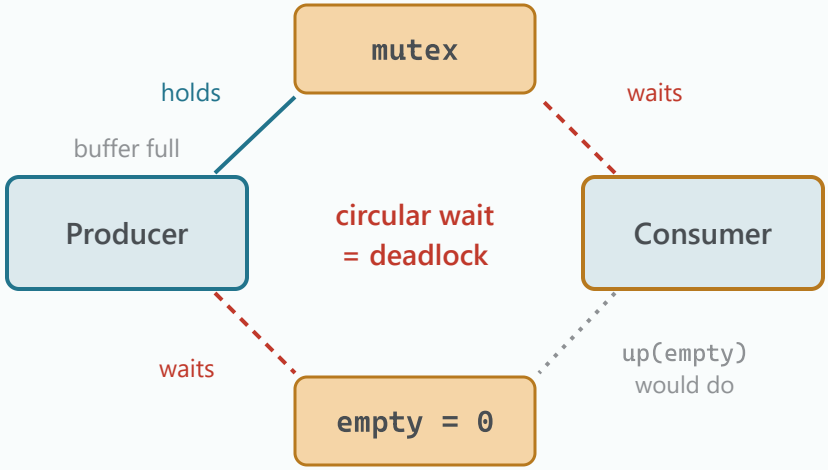
```
void producer(void) {
  while (true) {
    item = produce_item();
    down(&mutex);  /* ← first */
    down(&empty);  /* ← then  */
    insert_item(item);
    up(&mutex);
    up(&full);
  }
}
```




The consumer passes `down(&full)` but sleeps in `down(&mutex)`. It was the one that would call `up(&empty)`.

# If the downs Are Swapped

```
void producer(void) {
  while (true) {
    item = produce_item();
    down(&mutex);  /* ← first */
    down(&empty);  /* ← then  */
    insert_item(item);
    up(&mutex);
    up(&full);
  }
}
```



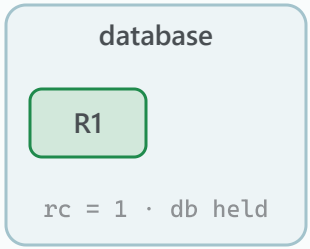

Each waits for the other: **deadlock**. Rule: wait on the counting semaphore first, then the mutex; never sleep while holding the mutex.

# The Readers–Writers Problem

```
semaphore mutex = 1;  /* guards rc          */
semaphore db = 1;     /* guards database    */
int rc = 0;           /* reader count       */

void writer(void) {
  while (true) {
    think_up_data();
    down(&db);  write_data_base();  up(&db);
  }
}
void reader(void) {
  while (true) {
    down(&mutex);
    if (++rc == 1) down(&db);  /* first reader */
    up(&mutex);
    read_data_base();
    down(&mutex);
    if (--rc == 0) up(&db);    /* last reader  */
    up(&mutex);
    use_data_read();
  }
}
```

first reader locks db



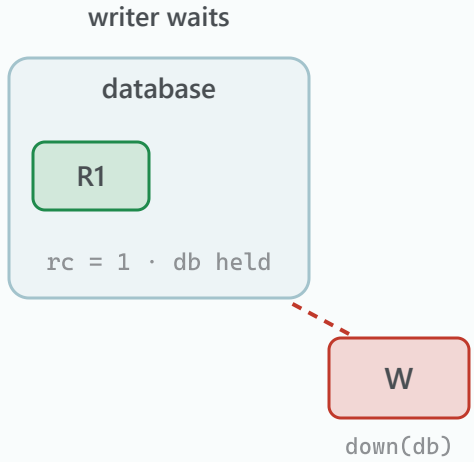

Readers don't exclude each other; only the **first** reader takes `db`.

Step 1 / 5

# The Readers–Writers Problem

```
semaphore mutex = 1;  /* guards rc           */
semaphore db = 1;     /* guards database     */
int rc = 0;           /* reader count        */

void writer(void) {
  while (true) {
    think_up_data();
    down(&db);  write_data_base();  up(&db);
  }
}
void reader(void) {
  while (true) {
    down(&mutex);
    if (++rc == 1) down(&db);  /* first reader */
    up(&mutex);
    read_data_base();
    down(&mutex);
    if (--rc == 0) up(&db);    /* last reader  */
    up(&mutex);
    use_data_read();
  }
}
```

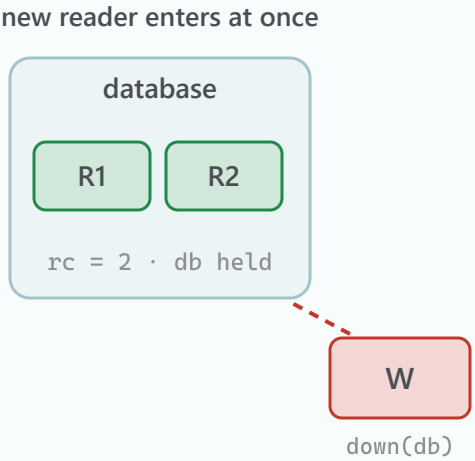



A writer arrives and waits in `down(&db)`.

# The Readers–Writers Problem

```
semaphore mutex = 1;  /* guards rc          */
semaphore db = 1;     /* guards database    */
int rc = 0;           /* reader count       */

void writer(void) {
  while (true) {
    think_up_data();
    down(&db);  write_data_base();  up(&db);
  }
}
void reader(void) {
  while (true) {
    down(&mutex);
    if (++rc == 1) down(&db);  /* first reader */
    up(&mutex);
    read_data_base();
    down(&mutex);
    if (--rc == 0) up(&db);    /* last reader  */
    up(&mutex);
    use_data_read();
  }
}
```



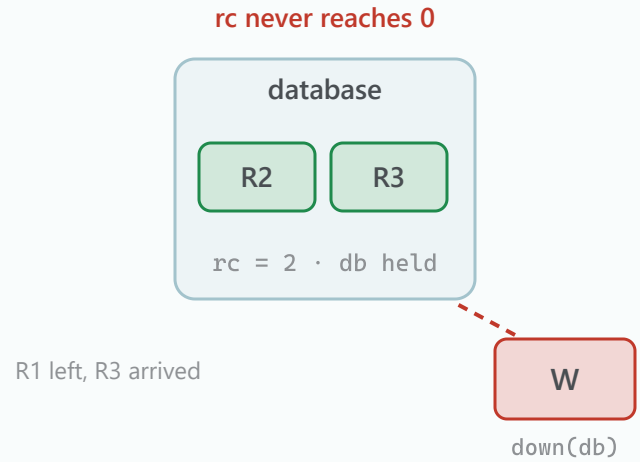

A new reader, R2, enters without touching `db` : `rc = 2` .

Step 3 / 5

# The Readers–Writers Problem

```
semaphore mutex = 1;  /* guards rc          */
semaphore db = 1;     /* guards database    */
int rc = 0;           /* reader count       */

void writer(void) {
  while (true) {
    think_up_data();
    down(&db);  write_data_base();  up(&db);
  }
}
void reader(void) {
  while (true) {
    down(&mutex);
    if (++rc == 1) down(&db);  /* first reader */
    up(&mutex);
    read_data_base();
    down(&mutex);
    if (--rc == 0) up(&db);    /* last reader  */
    up(&mutex);
    use_data_read();
  }
}
```

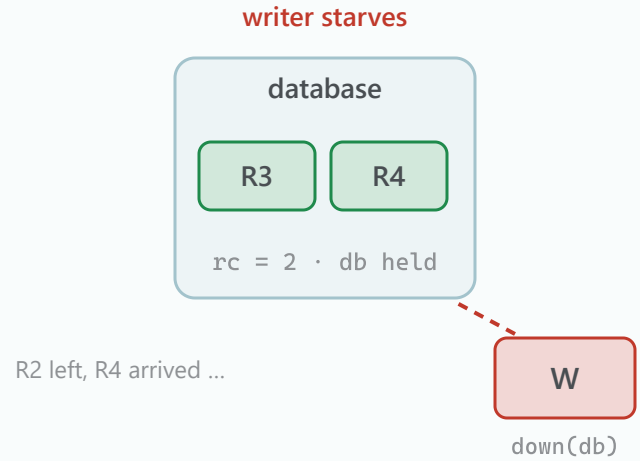



R3 arrives as R1 leaves: `rc` drops to 1 and climbs back to 2; it never hits 0.

# The Readers–Writers Problem

```
semaphore mutex = 1;  /* guards rc          */
semaphore db = 1;     /* guards database    */
int rc = 0;           /* reader count       */

void writer(void) {
  while (true) {
    think_up_data();
    down(&db);  write_data_base();  up(&db);
  }
}
void reader(void) {
  while (true) {
    down(&mutex);
    if (++rc == 1) down(&db);  /* first reader */
    up(&mutex);
    read_data_base();
    down(&mutex);
    if (--rc == 0) up(&db);    /* last reader  */
    up(&mutex);
    use_data_read();
  }
}
```




As long as readers keep coming the writer **waits indefinitely**: this solution favors readers and is prone to **writer starvation**.

# Readers–Writers: Variants and APIs

| Variant | Behavior |
|---|---|
| First readers–writers (reader priority) | A writer enters only when there are no readers; **writers may starve**. |
| Second readers–writers (writer priority) | No new reader may enter while a writer waits; **readers may starve**. |
| Fair (turnstile) | Everyone passes through the same turnstile first; a waiting writer holds the turnstile. |
| `pthread_rwlock_t` | The glibc default favors readers; writer preference is configurable. |
| Java `ReentrantReadWriteLock` | `new ReentrantReadWriteLock(true)` gives fair (near-FIFO) ordering. |
| Linux seqlock, RCU | Read-mostly kernel structures in which readers take no lock at all. |

```
/* Downey: starvation-free */
/* writer */
wait(turnstile);
wait(roomEmpty);
  write();
signal(turnstile);
signal(roomEmpty);

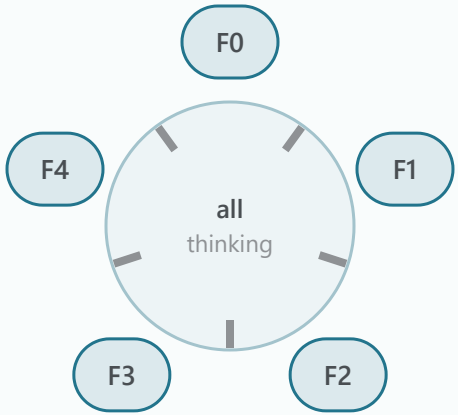
/* reader */
wait(turnstile); signal(turnstile);
/* first reader takes roomEmpty,
   last reader releases it (via rc) */
  read();
```

# Dining Philosophers: The Naive Solution

```
semaphore fork[5];          /* all 1 */

void philosopher(int i) {
  while (true) {
    think();
    wait(&fork[i]);               /* left */
    wait(&fork[(i + 1) % 5]);     /* right */
    eat();
    signal(&fork[(i + 1) % 5]);
    signal(&fork[i]);
  }
}
```

- Dijkstra (1965): each philosopher thinks, and when hungry picks up the **two** neighboring forks and eats.
- A model for sharing limited resources among processes **without deadlock or starvation**.



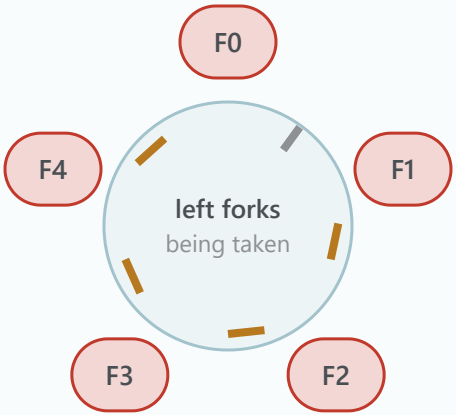

Forks are free; everyone is thinking.

# Dining Philosophers: The Naive Solution

```
semaphore fork[5];          /* all 1 */

void philosopher(int i) {
  while (true) {
    think();
    wait(&fork[i]);               /* left */
    wait(&fork[(i + 1) % 5]);     /* right */
    eat();
    signal(&fork[(i + 1) % 5]);
    signal(&fork[i]);
  }
}
```

- Dijkstra (1965): each philosopher thinks, and when hungry picks up the **two** neighboring forks and eats.
- A model for sharing limited resources among processes **without deadlock or starvation**.



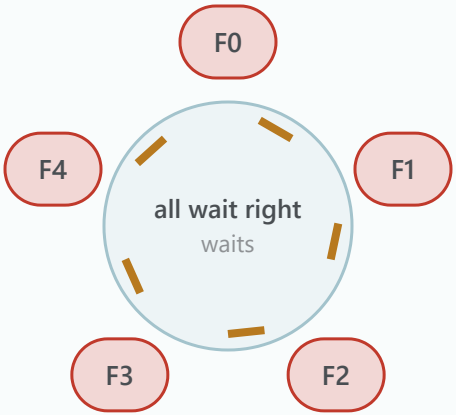

All get hungry at once and pick up their **left** fork…

# Dining Philosophers: The Naive Solution

```
semaphore fork[5];          /* all 1 */

void philosopher(int i) {
  while (true) {
    think();
    wait(&fork[i]);               /* left */
    wait(&fork[(i + 1) % 5]);     /* right */
    eat();
    signal(&fork[(i + 1) % 5]);
    signal(&fork[i]);
  }
}
```

- Dijkstra (1965): each philosopher thinks, and when hungry picks up the **two** neighboring forks and eats.
- A model for sharing limited resources among processes **without deadlock or starvation**.



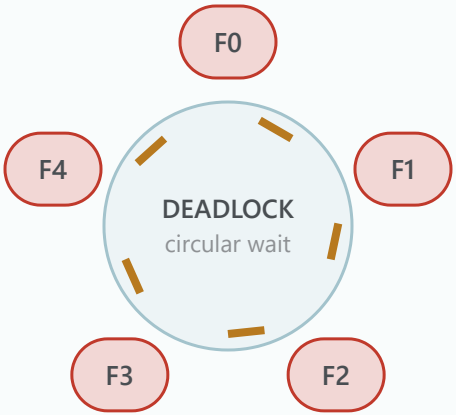

...the fifth does too. Now everyone waits for the **right** fork.

# Dining Philosophers: The Naive Solution

```
semaphore fork[5];          /* all 1 */

void philosopher(int i) {
  while (true) {
    think();
    wait(&fork[i]);               /* left */
    wait(&fork[(i + 1) % 5]);     /* right */
    eat();
    signal(&fork[(i + 1) % 5]);
    signal(&fork[i]);
  }
}
```

- Dijkstra (1965): each philosopher thinks, and when hungry picks up the **two** neighboring forks and eats.
- A model for sharing limited resources among processes **without deadlock or starvation**.




No one lets go: **deadlock**. The solution is wrong even though every fork is guarded by a semaphore.

CLASSIC PROBLEMS

# Philosophers: Ways Out

### At most four at the table

A `room` semaphore initialized to 4: at least one philosopher gets both forks. Circular wait is broken.

### Resource ordering (asymmetry)

Each philosopher picks up the *lower-numbered* fork first; F4 asks for fork 0 first. General rule: always acquire locks in one global order.

### Pick up both forks together

Tanenbaum: a `state[]` array and a semaphore per philosopher; start eating only if neither neighbor is eating.

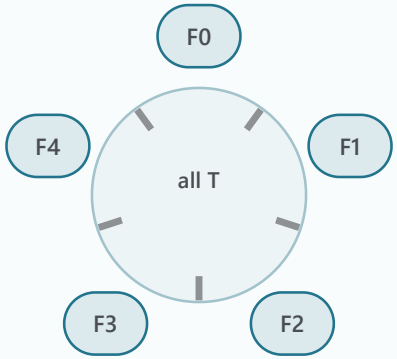
### Monitor

Silberschatz: `pickup(i)` / `putdown(i)` and a condition variable `self[i]` per philosopher; the same `test()` logic.

Deadlock-free does not mean **starvation-free**: a philosopher whose two neighbors take turns eating may wait forever. Preventing starvation needs an extra rule (ticket numbers, aging).

# Tanenbaum's Solution, Step by Step

```
1  void take_forks(int i) {
2    down(&mutex);
3    state[i] = HUNGRY;
4    test(i);
5    up(&mutex);
6    down(&s[i]);    /* block if no forks */
7  }
8  void put_forks(int i) {
9    down(&mutex);
10   state[i] = THINKING;
11   test(LEFT);     /* (i+4) % 5 */
12   test(RIGHT);    /* (i+1) % 5 */
13   up(&mutex);
14 }
15 void test(int i) {
16   if (state[i] == HUNGRY && state[LEFT] != EATING
17                          && state[RIGHT] != EATING) {
18     state[i] = EATING;
19     up(&s[i]);
20   }
21 }
```




Initially everyone is `THINKING` (T). The `s[i]` semaphores are 0.

Step 1 / 6

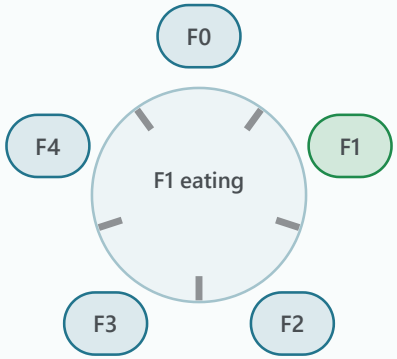
**VARIABLES**

`state` = **TTTTT**

Tanenbaum MOS Figure 2-47. There is no deadlock, and at most ⌊5/2⌋ = 2 of the five eat at once; but if F1 and F3 keep taking turns, F2 could in theory starve.

# Tanenbaum's Solution, Step by Step

```
 1 void take_forks(int i) {
 2   down(&mutex);
▶    state[i] = HUNGRY;
▶    test(i);
 5   up(&mutex);
▶    down(&s[i]);    /* block if no forks */
 7 }
 8 void put_forks(int i) {
 9   down(&mutex);
10   state[i] = THINKING;
11   test(LEFT);     /* (i+4) % 5 */
12   test(RIGHT);    /* (i+1) % 5 */
13   up(&mutex);
14 }
15 void test(int i) {
16   if (state[i] == HUNGRY && state[LEFT] != EATING
17                          && state[RIGHT] != EATING) {
▶      state[i] = EATING;
▶      up(&s[i]);
20   }
21 }
```




F1 gets hungry; `test(1)` : neighbors F0 and F2 aren't eating → `EATING` , `up(&s[1])` ; then `down(&s[1])` passes at once.

Step 2 / 6

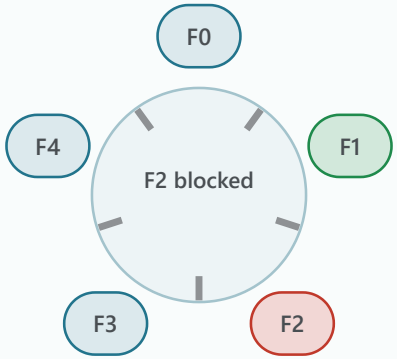
**VARIABLES**

`state = TETTT` `s[1] = 0`

Tanenbaum MOS Figure 2-47. There is no deadlock, and at most ⌊5/2⌋ = 2 of the five eat at once; but if F1 and F3 keep taking turns, F2 could in theory starve.

# Tanenbaum's Solution, Step by Step

```
1  void take_forks(int i) {
2    down(&mutex);
▶    state[i] = HUNGRY;
▶    test(i);
5    up(&mutex);
▶    down(&s[i]);    /* block if no forks */
7  }
8  void put_forks(int i) {
9    down(&mutex);
10   state[i] = THINKING;
11   test(LEFT);     /* (i+4) % 5 */
12   test(RIGHT);    /* (i+1) % 5 */
13   up(&mutex);
14 }
15 void test(int i) {
16   if (state[i] == HUNGRY && state[LEFT] != EATING
17                          && state[RIGHT] != EATING) {
18     state[i] = EATING;
19     up(&s[i]);
20   }
21 }
```




F2 gets hungry; its left neighbor F1 is eating → the test fails and it sleeps in `down(&s[2])`.

Step 3 / 6

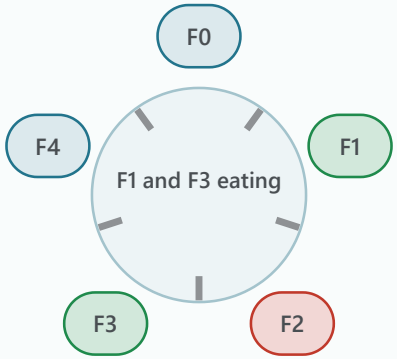
VARIABLES

`state = TEHTT` `s[2] = 0`

Tanenbaum MOS Figure 2-47. There is no deadlock, and at most ⌊5/2⌋ = 2 of the five eat at once; but if F1 and F3 keep taking turns, F2 could in theory starve.

# Tanenbaum's Solution, Step by Step

```
 1 void take_forks(int i) {
 2   down(&mutex);
▶    state[i] = HUNGRY;
▶    test(i);
 5   up(&mutex);
▶    down(&s[i]);    /* block if no forks */
 7 }
 8 void put_forks(int i) {
 9   down(&mutex);
10   state[i] = THINKING;
11   test(LEFT);     /* (i+4) % 5 */
12   test(RIGHT);    /* (i+1) % 5 */
13   up(&mutex);
14 }
15 void test(int i) {
16   if (state[i] == HUNGRY && state[LEFT] != EATING
17                          && state[RIGHT] != EATING) {
▶      state[i] = EATING;
▶      up(&s[i]);
20   }
21 }
```




F3 gets hungry; F2 is hungry but not eating, F4 is thinking → F3 eats. Two philosophers can eat at once.

Step 4 / 6

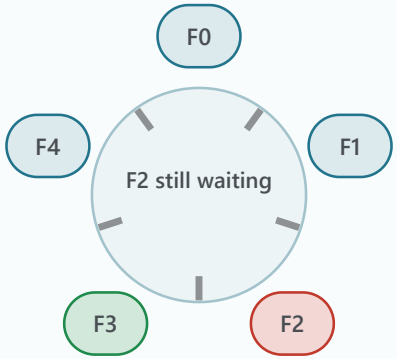
VARIABLES

`state = TEHET` `s[3] = 0`

Tanenbaum MOS Figure 2-47. There is no deadlock, and at most ⌊5/2⌋ = 2 of the five eat at once; but if F1 and F3 keep taking turns, F2 could in theory starve.

# Tanenbaum's Solution, Step by Step

```
1  void take_forks(int i) {
2    down(&mutex);
3    state[i] = HUNGRY;
4    test(i);
5    up(&mutex);
6    down(&s[i]);    /* block if no forks */
7  }
8  void put_forks(int i) {
9    down(&mutex);
▶    state[i] = THINKING;
▶    test(LEFT);     /* (i+4) % 5 */
▶    test(RIGHT);    /* (i+1) % 5 */
13   up(&mutex);
14 }
15 void test(int i) {
16   if (state[i] == HUNGRY && state[LEFT] != EATING
17                          && state[RIGHT] != EATING) {
18     state[i] = EATING;
19     up(&s[i]);
20   }
21 }
```




F1 puts down its forks and tests its neighbors: `test(2)` fails because F3 on the right is still eating.

Step 5 / 6

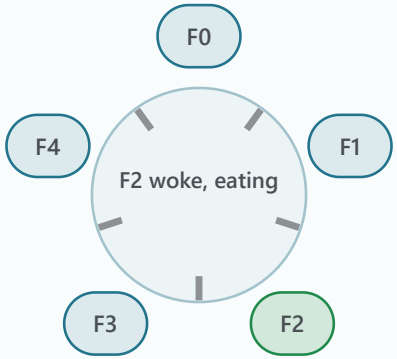
**VARIABLES**

`state = TTHET`

Tanenbaum MOS Figure 2-47. There is no deadlock, and at most ⌊5/2⌋ = 2 of the five eat at once; but if F1 and F3 keep taking turns, F2 could in theory starve.

# Tanenbaum's Solution, Step by Step

```
 1 void take_forks(int i) {
 2   down(&mutex);
 3   state[i] = HUNGRY;
 4   test(i);
 5   up(&mutex);
 6   down(&s[i]);    /* block if no forks */
 7 }
 8 void put_forks(int i) {
 9   down(&mutex);
▶   state[i] = THINKING;
▶   test(LEFT);     /* (i+4) % 5 */
12   test(RIGHT);    /* (i+1) % 5 */
13   up(&mutex);
14 }
15 void test(int i) {
16   if (state[i] == HUNGRY && state[LEFT] != EATING
17                          && state[RIGHT] != EATING) {
▶     state[i] = EATING;
▶     up(&s[i]);
20   }
21 }
```




When F3 puts its forks down, `test(2)` succeeds: F2 becomes `EATING` and `up(&s[2])` wakes it.

Step 6 / 6

**VARIABLES**

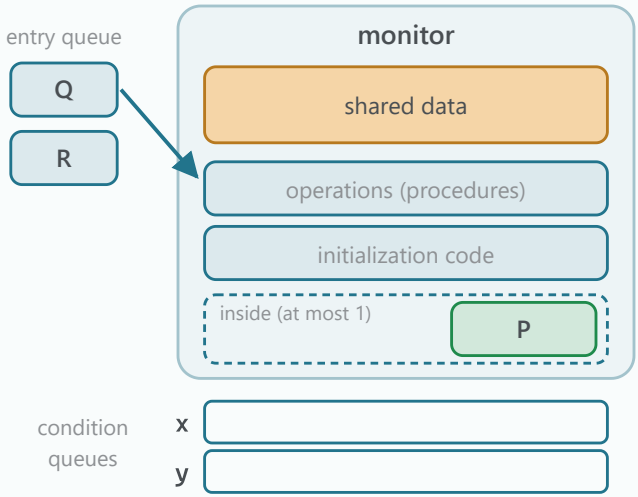
`state = TTETT` `s[2] = 0`

Tanenbaum MOS Figure 2-47. There is no deadlock, and at most ⌊5/2⌋ = 2 of the five eat at once; but if F1 and F3 keep taking turns, F2 could in theory starve.

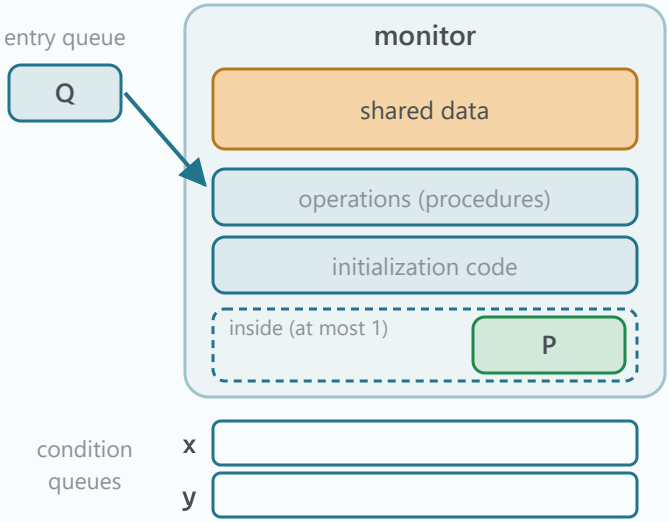
# Monitor

- Hoare (1974) and Brinch Hansen (1975): a **language-level** abstract data type that bundles data, procedures, and initialization code.
- Processes may call a monitor's procedures but **cannot access** its internal data directly.
- **Only one process** may be active inside the monitor at a time; the **compiler** inserts the entry lock, so the programmer cannot forget it.
- If the monitor is busy, the caller waits in the entry queue. To wait for a condition, processes use **condition variables**.
- Getting the `down` order wrong with semaphores caused deadlock; monitors shrink this class of bugs.

```
monitor example
  integer i;
  condition c;
  procedure producer(); … end;
  procedure consumer(); … end;
end monitor;
```



# Condition Variables



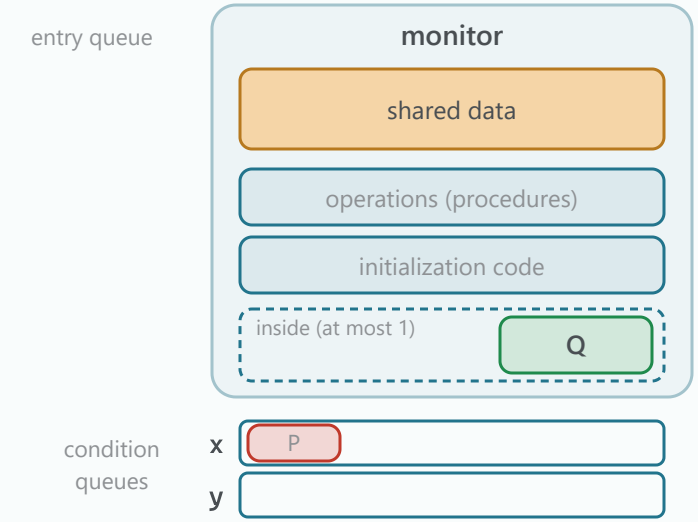

P is in the monitor but cannot continue because the buffer is empty. Q is in the entry queue.



| Operation | Meaning |
|---|---|
| `x.wait()` | Suspend the caller and release the monitor. |
| `x.signal()` | Resume one process waiting on x (if any). |
| `broadcast` / `notifyAll` | Wake every process waiting on x. |

A condition variable **has no value**; it is just a waiting queue. The condition being waited for (e.g., `count > 0`) is always kept separately in shared data.

# Condition Variables

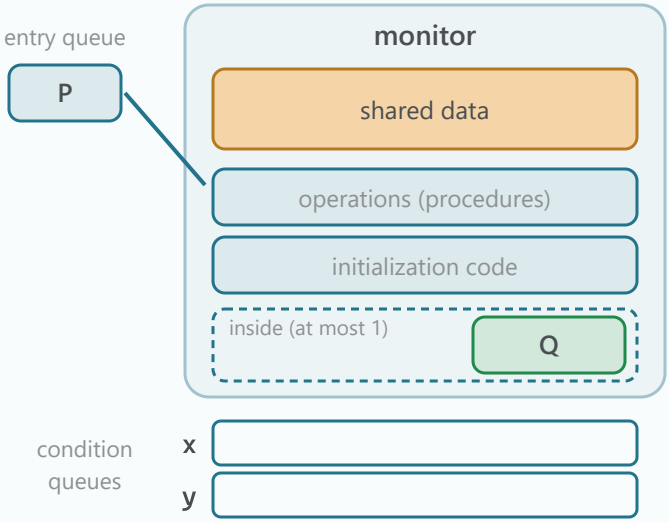



P calls `x.wait()` : it **releases the monitor** and sleeps on x's queue, so Q can enter.

| Operation | Meaning |
|---|---|
| `x.wait()` | Suspend the caller and release the monitor. |
| `x.signal()` | Resume one process waiting on x (if any). |
| `broadcast` / `notifyAll` | Wake every process waiting on x. |

A condition variable **has no value**; it is just a waiting queue. The condition being waited for (e.g., `count > 0`) is always kept separately in shared data.

# Condition Variables

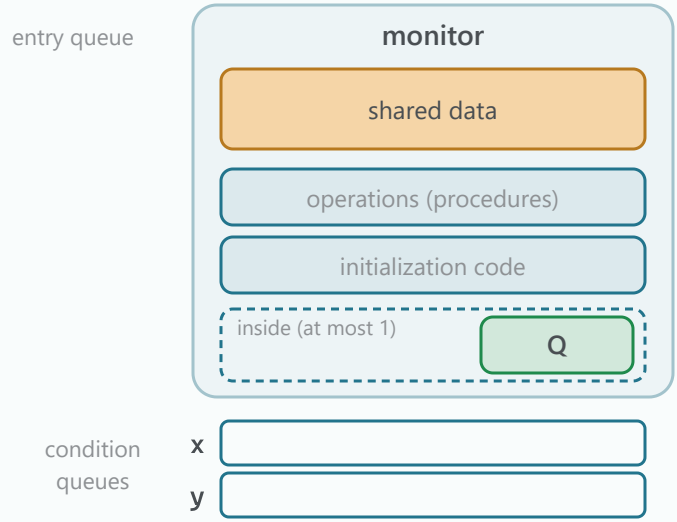



Q makes the condition true and calls `x.signal()` : **one** process on x's queue wakes. Who runs next is a semantic choice.

| Operation | Meaning |
|---|---|
| `x.wait()` | Suspend the caller and release the monitor. |
| `x.signal()` | Resume one process waiting on x (if any). |
| `broadcast` / `notifyAll` | Wake every process waiting on x. |

A condition variable **has no value**; it is just a waiting queue. The condition being waited for (e.g., `count > 0`) is always kept separately in shared data.

# Condition Variables

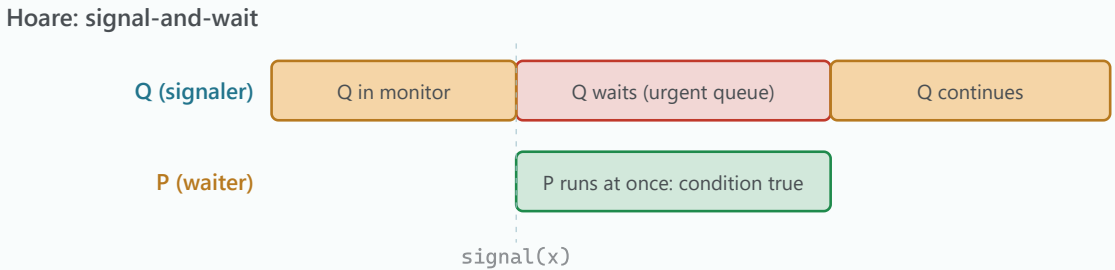



When no one waits on x, `x.signal()` **has no effect** and leaves no trace. A semaphore's `signal`, in contrast, increments the value and is remembered.

| Operation | Meaning |
|---|---|
| `x.wait()` | Suspend the caller and release the monitor. |
| `x.signal()` | Resume one process waiting on x (if any). |
| `broadcast` / `notifyAll` | Wake every process waiting on x. |

A condition variable **has no value**; it is just a waiting queue. The condition being waited for (e.g., `count > 0`) is always kept separately in shared data.

# Hoare and Mesa Semantics

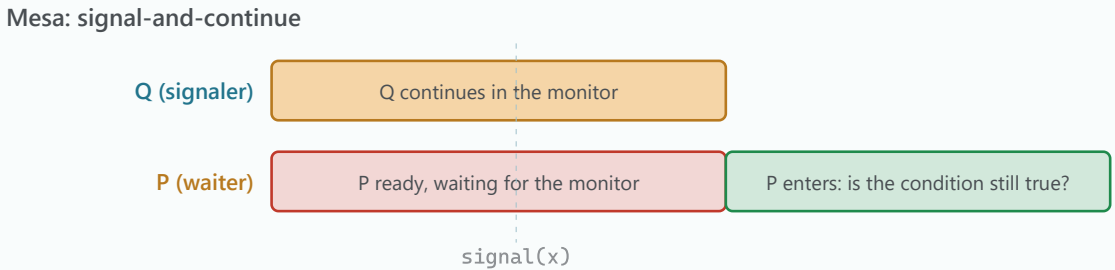



**Signal-and-wait** (Hoare): the awakened P runs **immediately** and Q waits. P can be sure the condition holds: an `if` suffices.

Step 1 / 2

| Semantics | Who runs next? | Used by |
|---|---|---|
| Signal-and-wait (Hoare) | The awakened process | Textbook monitors |
| Signal-and-exit (Brinch Hansen) | The awakened process; `signal` must be the procedure's last statement | Concurrent Pascal |
| Signal-and-continue (Mesa) | The signaler | Pthreads, Java, C#, Go `sync.Cond`, the Linux kernel |

# Hoare and Mesa Semantics




**Signal-and-continue** (Mesa, 1980): Q continues; P merely becomes ready. Someone else may falsify the condition before P runs: **it must recheck**.

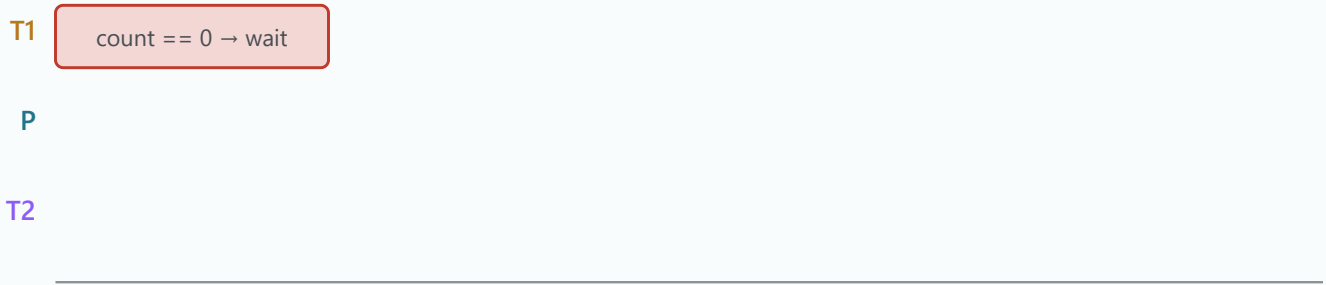
Step 2 / 2

| Semantics | Who runs next? | Used by |
|---|---|---|
| Signal-and-wait (Hoare) | The awakened process | Textbook monitors |
| Signal-and-exit (Brinch Hansen) | The awakened process; `signal` must be the procedure's last statement | Concurrent Pascal |
| Signal-and-continue (Mesa) | The signaler | Pthreads, Java, C#, Go `sync.Cond`, the Linux kernel |

# Why while, Not if?

```
/* WRONG (under Mesa) */
lock(&m);
if (count == 0)
  cond_wait(&full, &m);
x = take();      /* may be empty */
unlock(&m);
```

```
/* RIGHT */
lock(&m);
while (count == 0)
  cond_wait(&full, &m);
x = take();      /* count > 0 for sure */
unlock(&m);
```

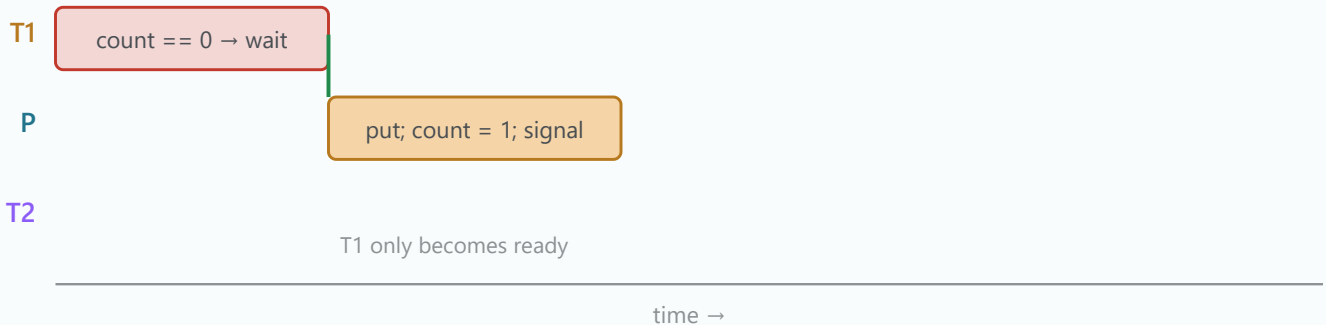



The buffer is empty: consumer T1 waits on the condition variable.

# Why while, Not if?

```c
/* WRONG (under Mesa) */
lock(&m);
if (count == 0)
  cond_wait(&full, &m);
x = take();      /* may be empty */
unlock(&m);
```

```c
/* RIGHT */
lock(&m);
while (count == 0)
  cond_wait(&full, &m);
x = take();      /* count > 0 for sure */
unlock(&m);
```

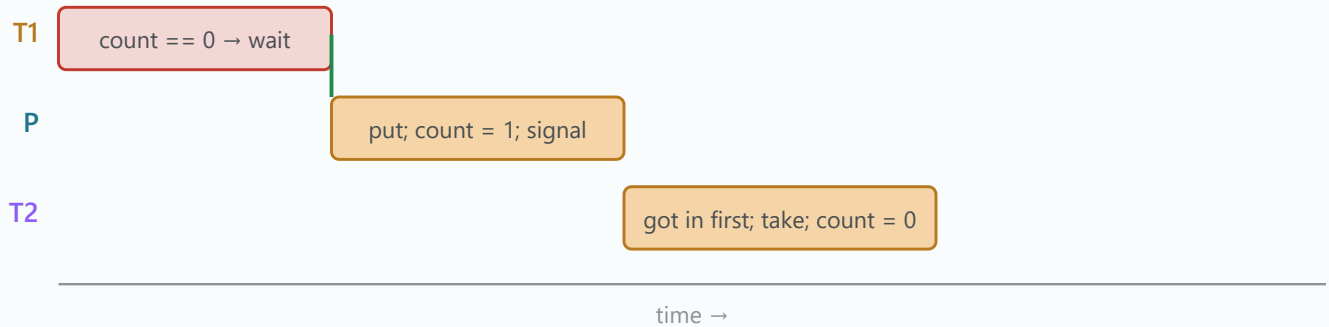



The producer puts an item and calls `signal`. T1 wakes, but must first reacquire the lock.

# Why while, Not if?

```
/* WRONG (under Mesa) */
lock(&m);
if (count == 0)
  cond_wait(&full, &m);
x = take();      /* may be empty */
unlock(&m);
```

```
/* RIGHT */
lock(&m);
while (count == 0)
  cond_wait(&full, &m);
x = take();      /* count > 0 for sure */
unlock(&m);
```

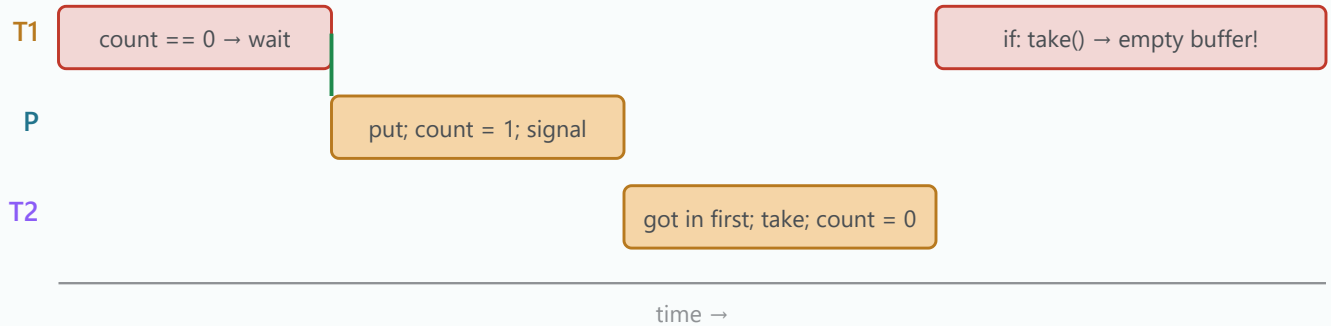



Meanwhile a newcomer, T2, grabs the lock **first** (barging) and takes the item: `count = 0`.

# Why while, Not if?

```
/* WRONG (under Mesa) */
lock(&m);
if (count == 0)
  cond_wait(&full, &m);
x = take();      /* may be empty */
unlock(&m);
```

```
/* RIGHT */
lock(&m);
while (count == 0)
  cond_wait(&full, &m);
x = take();      /* count > 0 for sure */
unlock(&m);
```

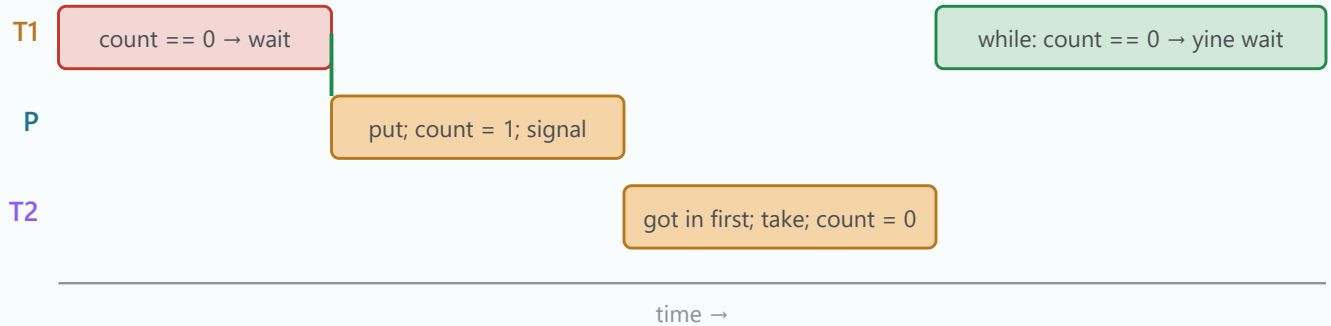



T1 finally runs. With `if` it doesn't retest the condition and reads from an empty buffer: a **bug**.

# Why while, Not if?

```c
/* WRONG (under Mesa) */
lock(&m);
if (count == 0)
  cond_wait(&full, &m);
x = take();      /* may be empty */
unlock(&m);
```

```c
/* RIGHT */
lock(&m);
while (count == 0)
  cond_wait(&full, &m);
x = take();      /* count > 0 for sure */
unlock(&m);
```




With `while`, T1 retests the condition and waits again. POSIX also allows **spurious wakeups**: `while` is required in every case.

# One Condition Variable Is Not Enough

```
/* WRONG: one condition c */
C: while (count == 0) wait(&c, &m);
   x = get(); signal(&c);
P: while (count == 1) wait(&c, &m);
   put(x);    signal(&c);
```

```
/* RIGHT: two conditions */
C: while (count == 0) wait(&full, &m);
   x = get(); signal(&empty);
P: while (count == 1) wait(&empty, &m);
   put(x);    signal(&full);
```




The buffer (size 1) is empty: both consumers sleep on the same condition variable `c`.

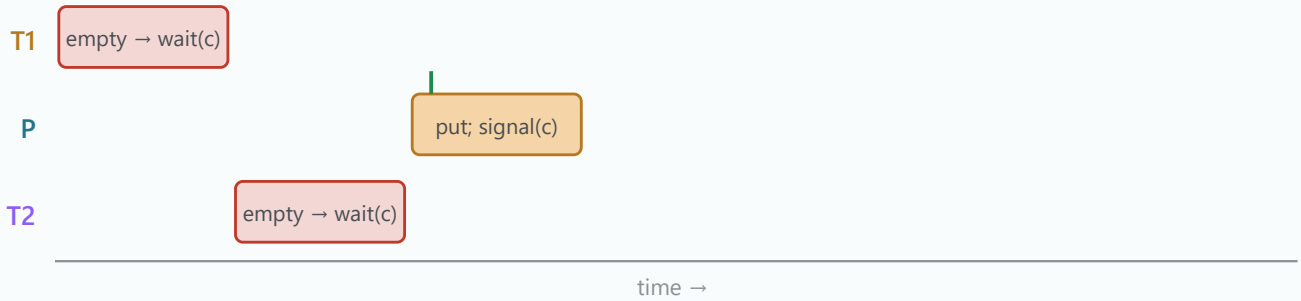
Step 1 / 5

Both versions hold lock `m` and use `while`. The bug: all threads wait on the **same** queue, yet a consumer should wake only the producer, and vice versa.

# One Condition Variable Is Not Enough

```
/* WRONG: one condition c */
C: while (count == 0) wait(&c, &m);
   x = get(); signal(&c);
P: while (count == 1) wait(&c, &m);
   put(x);    signal(&c);
```

```
/* RIGHT: two conditions */
C: while (count == 0) wait(&full, &m);
   x = get(); signal(&empty);
P: while (count == 1) wait(&empty, &m);
   put(x);    signal(&full);
```




The producer puts an item and wakes T1 with `signal(c)`.

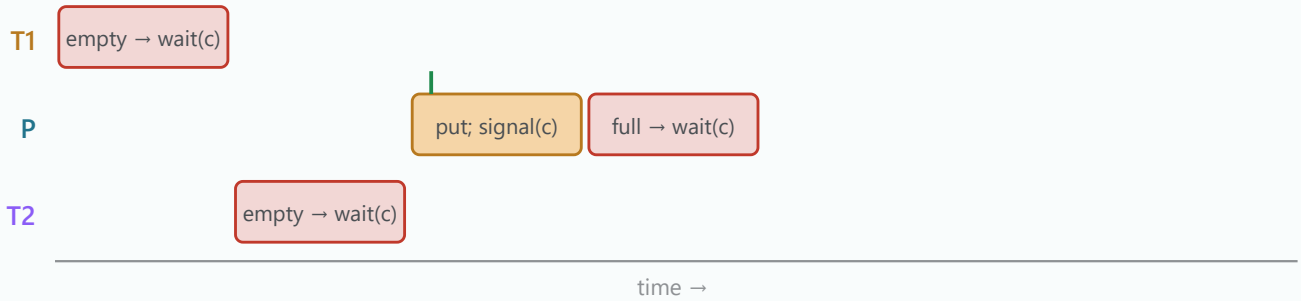


Both versions hold lock `m` and use `while`. The bug: all threads wait on the **same** queue, yet a consumer should wake only the producer, and vice versa.

# One Condition Variable Is Not Enough

```
/* WRONG: one condition c */
C: while (count == 0) wait(&c, &m);
   x = get(); signal(&c);
P: while (count == 1) wait(&c, &m);
   put(x);    signal(&c);
```

```
/* RIGHT: two conditions */
C: while (count == 0) wait(&full, &m);
   x = get(); signal(&empty);
P: while (count == 1) wait(&empty, &m);
   put(x);    signal(&full);
```




The producer retries; the buffer is full, so it also sleeps on `c`. Now T2 and P wait on `c`.

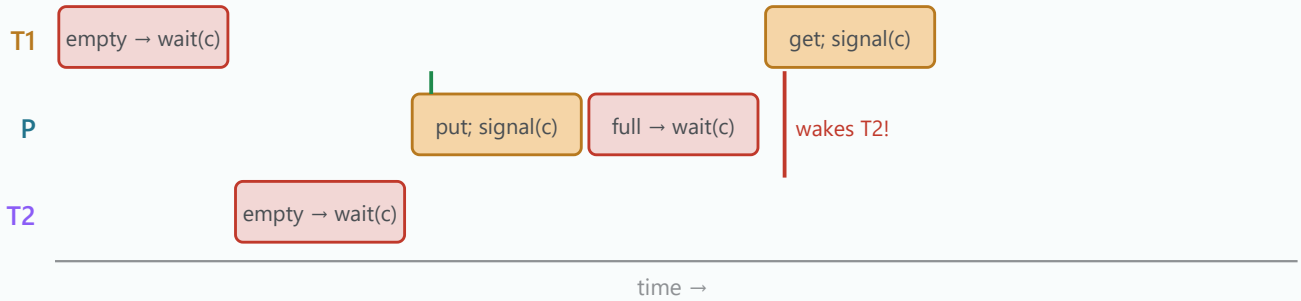


Both versions hold lock `m` and use `while`. The bug: all threads wait on the **same** queue, yet a consumer should wake only the producer, and vice versa.

# One Condition Variable Is Not Enough

```
/* WRONG: one condition c */
C: while (count == 0) wait(&c, &m);
   x = get(); signal(&c);
P: while (count == 1) wait(&c, &m);
   put(x);    signal(&c);
```

```
/* RIGHT: two conditions */
C: while (count == 0) wait(&full, &m);
   x = get(); signal(&empty);
P: while (count == 1) wait(&empty, &m);
   put(x);    signal(&full);
```




T1 takes the item and calls `signal(c)`. Which waiter wakes is unspecified: here, **T2**.

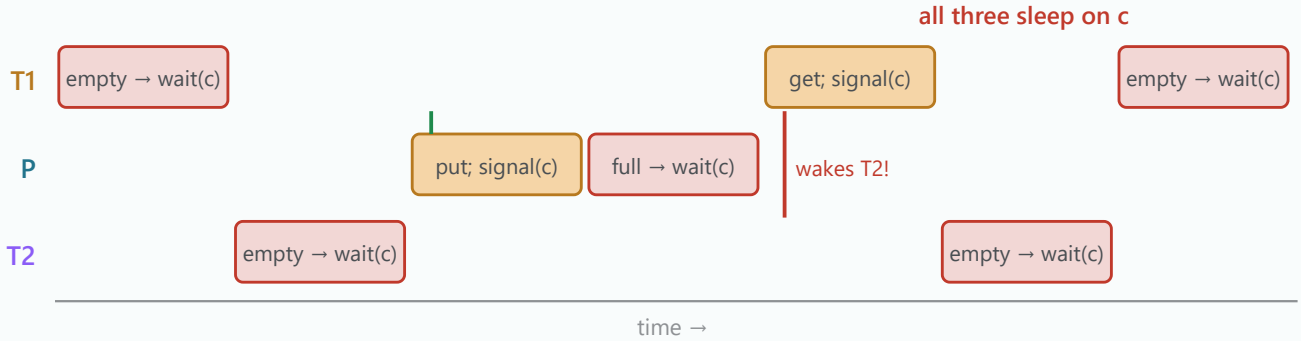


Both versions hold lock `m` and use `while`. The bug: all threads wait on the **same** queue, yet a consumer should wake only the producer, and vice versa.

# One Condition Variable Is Not Enough

```
/* WRONG: one condition c */
C: while (count == 0) wait(&c, &m);
   x = get(); signal(&c);
P: while (count == 1) wait(&c, &m);
   put(x);    signal(&c);
```

```
/* RIGHT: two conditions */
C: while (count == 0) wait(&full, &m);
   x = get(); signal(&empty);
P: while (count == 1) wait(&empty, &m);
   put(x);    signal(&full);
```




T2 finds the buffer empty and sleeps, as does T1 next round. No one can wake the producer: the program hangs.

Step 5 / 5

Both versions hold lock `m` and use `while`. The bug: all threads wait on the **same** queue, yet a consumer should wake only the producer, and vice versa.

# Whom to Wake? signal and broadcast

```
int avail = 0;          /* free bytes */

void mem_alloc(int n) {
  lock(&m);
  while (avail < n)
    wait(&c, &m);
  avail -= n;  /* ... allocate */
  unlock(&m);
}
void mem_free(int n) {
  lock(&m);
  avail += n;
  broadcast(&c); /* signal won't do */
  unlock(&m);
}
```

**avail = 0** initially

**Ta** wants 100 bytes
waits

**Tb** wants 10 bytes
waits

Both wait on c.

c queue: Ta, Tb

Memory is exhausted: Ta wants 100 bytes, Tb wants 10; both sleep in their `while` loops.

Step 1 / 3

**Covering condition:** when you can't tell which waiter can proceed, use `pthread_cond_broadcast` (Java: `notifyAll`). If waiters can be split by condition, `signal` is cheaper. Either way, wait inside a `while`.

# Whom to Wake? signal and broadcast

```
int avail = 0;          /* free bytes */

void mem_alloc(int n) {
  lock(&m);
  while (avail < n)
    wait(&c, &m);
  avail -= n;  /* ... allocate */
  unlock(&m);
}
void mem_free(int n) {
  lock(&m);
  avail += n;
  broadcast(&c); /* signal won't do */
  unlock(&m);
}
```

**avail = 50** free(50) + signal

**Ta** wants 100 bytes
woke: 50 < 100 → yine bekler

**Tb** wants 10 bytes
not woken, still waiting

**Tb could proceed but is asleep.**

c queue: Ta, Tb

Tc frees 50 bytes. `signal` wakes only one; if that is Ta, it sleeps again and Tb's chance is wasted.



**Covering condition:** when you can't tell which waiter can proceed, use `pthread_cond_broadcast` (Java: `notifyAll`). If waiters can be split by condition, `signal` is cheaper. Either way, wait inside a `while`.

# Whom to Wake? signal and broadcast

```
int avail = 0;          /* free bytes */

void mem_alloc(int n) {
  lock(&m);
  while (avail < n)
    wait(&c, &m);
  avail -= n;  /* ... allocate */
  unlock(&m);
}
void mem_free(int n) {
  lock(&m);
  avail += n;
  broadcast(&c); /* signal won't do */
  unlock(&m);
}
```

avail = 40 free(50) + broadcast

**Ta** wants 100 bytes
woke: 40 < 100 → yine bekler

**Tb** wants 10 bytes
woke: 50 ≥ 10 → takes it

**Tb proceeded; Ta woke for nothing.**

c queue: Ta

`broadcast` wakes everyone, and each retests its condition. The cost is needless wakeups.



**Covering condition:** when you can't tell which waiter can proceed, use `pthread_cond_broadcast` (Java: `notifyAll`). If waiters can be split by condition, `signal` is cheaper. Either way, wait inside a `while`.

# Producer–Consumer with a Monitor

```
monitor ProducerConsumer
  condition full, empty;
  integer count;

  procedure insert(item: integer);
  begin
    if count = N then wait(full);
    insert_item(item);
    count := count + 1;
    if count = 1 then signal(empty)
  end;

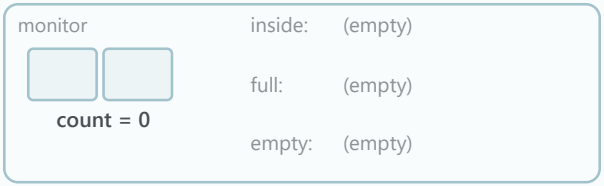
  function remove: integer;
  begin
    if count = 0 then wait(empty);
    remove := remove_item;
    count := count - 1;
    if count = N - 1 then signal(full)
  end;

  count := 0;
end monitor;
```

- Tanenbaum Figure 2-34 (Pidgin Pascal). The monitor already guarantees mutual exclusion on `count`.
- If the buffer is **full, the producer** blocks in `wait(full)` and releases the monitor; once a consumer enters and takes an item, `signal(full)` wakes the producer.
- It is `wait` that blocks the producer; `signal` only wakes it.
- An `if` suffices here because Tanenbaum assumes Brinch Hansen semantics (signal is the last statement). In Pthreads/Java the same code must use `while`.
- No lost wakeup: the test and the `wait` happen inside the same monitor, without interruption.

# Producer–Consumer with a Monitor: Code Trace

```
1  monitor ProducerConsumer      { N = 2 }
▶    condition full, empty; integer count;
3    procedure insert(item: integer);
4    begin
5      if count = N then wait(full);
6      insert_item(item); count := count + 1;
7      if count = 1 then signal(empty)
8    end;
9    function remove: integer;
10   begin
11     if count = 0 then wait(empty);
12     remove := remove_item; count := count - 1;
13     if count = N - 1 then signal(full)
14   end;
```




The buffer is empty ( `count = 0` ), N = 2. Producer P and consumer C enter the monitor in turn.

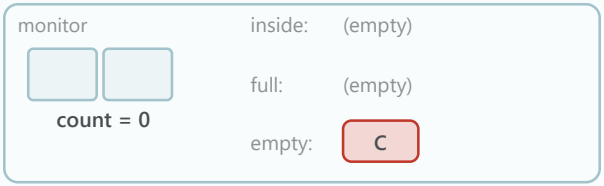
Step 1 / 10

**VARIABLES**

`count = 0` `buffer = []` `full = []` `empty = []` `C got = –`

# Producer–Consumer with a Monitor: Code Trace

```
1  monitor ProducerConsumer      { N = 2 }
2    condition full, empty; integer count;
3    procedure insert(item: integer);
4    begin
5      if count = N then wait(full);
6      insert_item(item); count := count + 1;
7      if count = 1 then signal(empty)
8    end;
9    function remove: integer;
10   begin
▶      if count = 0 then wait(empty);
12     remove := remove_item; count := count - 1;
13     if count = N - 1 then signal(full)
14   end;
```




C calls `remove` : count = 0 → `wait(empty)` .
C sleeps and **releases the monitor**.

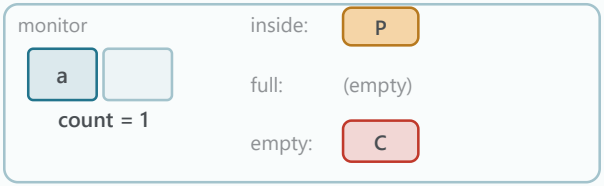
Step 2 / 10

**VARIABLES**

`count = 0` `buffer = []` `full = []`
`empty = [C]` `C got = –`

# Producer–Consumer with a Monitor: Code Trace

```
1  monitor ProducerConsumer      { N = 2 }
2    condition full, empty; integer count;
3    procedure insert(item: integer);
4    begin
5      if count = N then wait(full);
▶      insert_item(item); count := count + 1;
7      if count = 1 then signal(empty)
8    end;
9    function remove: integer;
10   begin
11     if count = 0 then wait(empty);
12     remove := remove_item; count := count - 1;
13     if count = N - 1 then signal(full)
14   end;
```




P calls `insert(a)`: the buffer isn't full, so it inserts the item; count = 1.

Step 3 / 10

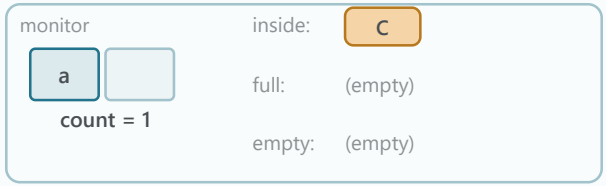
**VARIABLES**

`count = 1` `buffer = [a]`

`full = []` `empty = [C]` `C got = –`

# Producer–Consumer with a Monitor: Code Trace

```
1  monitor ProducerConsumer      { N = 2 }
2    condition full, empty; integer count;
3    procedure insert(item: integer);
4    begin
5      if count = N then wait(full);
6      insert_item(item); count := count + 1;
▶      if count = 1 then signal(empty)
8    end;
9    function remove: integer;
10   begin
11     if count = 0 then wait(empty);
12     remove := remove_item; count := count - 1;
13     if count = N - 1 then signal(full)
14   end;
```




count = 1 → `signal(empty)` : C wakes. Since `signal` is the last statement, P leaves at once; there are never two processes in the monitor.

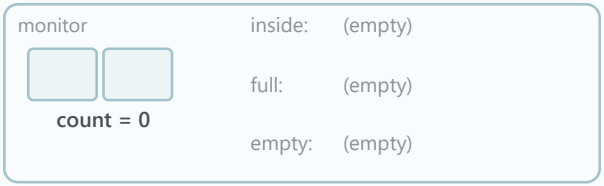
Step 4 / 10

**VARIABLES**

`count = 1` `buffer = [a]` `full = []` `empty = []` `C got = –`

# Producer–Consumer with a Monitor: Code Trace

```
1  monitor ProducerConsumer      { N = 2 }
2    condition full, empty; integer count;
3    procedure insert(item: integer);
4    begin
5      if count = N then wait(full);
6      insert_item(item); count := count + 1;
7      if count = 1 then signal(empty)
8    end;
9    function remove: integer;
10   begin
11     if count = 0 then wait(empty);
▶      remove := remove_item; count := count - 1;
▶      if count = N - 1 then signal(full)
14   end;
```




C resumes: takes `a`, count = 0. 0 ≠ N − 1, so no signal.

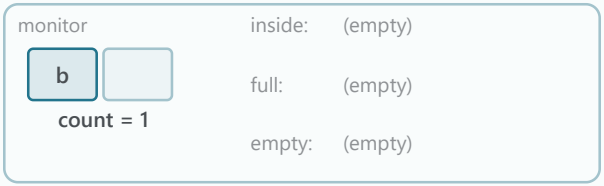
Step 5 / 10

**VARIABLES**

`count = 0` `buffer = []` `full = []` `empty = []` `C got = a`

# Producer–Consumer with a Monitor: Code Trace

```
1  monitor ProducerConsumer      { N = 2 }
2    condition full, empty; integer count;
3    procedure insert(item: integer);
4    begin
5      if count = N then wait(full);
▶      insert_item(item); count := count + 1;
▶      if count = 1 then signal(empty)
8    end;
9    function remove: integer;
10   begin
11     if count = 0 then wait(empty);
12     remove := remove_item; count := count - 1;
13     if count = N - 1 then signal(full)
14   end;
```




P calls `insert(b)`: count = 1 → `signal(empty)`, but no one is waiting: the signal is **lost**, harmlessly.

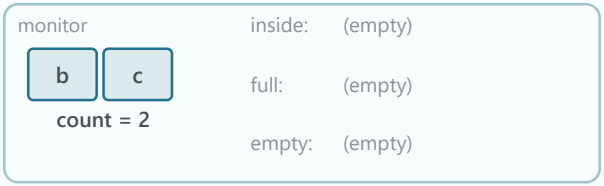
Step 6 / 10

**VARIABLES**

`count = 1` `buffer = [b]` `full = []` `empty = []` `C got = a`

# Producer–Consumer with a Monitor: Code Trace

```
1  monitor ProducerConsumer      { N = 2 }
2    condition full, empty; integer count;
3    procedure insert(item: integer);
4    begin
5      if count = N then wait(full);
▶      insert_item(item); count := count + 1;
7      if count = 1 then signal(empty)
8    end;
9    function remove: integer;
10   begin
11     if count = 0 then wait(empty);
12     remove := remove_item; count := count - 1;
13     if count = N - 1 then signal(full)
14   end;
```




P calls `insert(c)`: count = 2. The buffer is full.

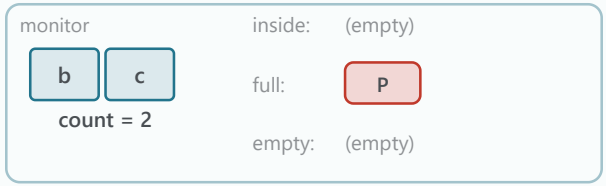
Step 7 / 10

**VARIABLES**

`count = 2` `buffer = [b c]` `full = []` `empty = []` `C got = a`

# Producer–Consumer with a Monitor: Code Trace

```
1  monitor ProducerConsumer      { N = 2 }
2    condition full, empty; integer count;
3    procedure insert(item: integer);
4    begin
▶      if count = N then wait(full);
6      insert_item(item); count := count + 1;
7      if count = 1 then signal(empty)
8    end;
9    function remove: integer;
10   begin
11     if count = 0 then wait(empty);
12     remove := remove_item; count := count - 1;
13     if count = N - 1 then signal(full)
14   end;
```




P calls `insert(d)`: count = N → `wait(full)`. P sleeps and the monitor is free.

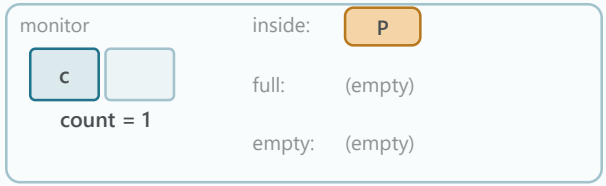
Step 8 / 10

**VARIABLES**

`count = 2` `buffer = [b c]` `full = [P]` `empty = []` `C got = a`

# Producer–Consumer with a Monitor: Code Trace

```
1  monitor ProducerConsumer      { N = 2 }
2    condition full, empty; integer count;
3    procedure insert(item: integer);
4    begin
5      if count = N then wait(full);
6      insert_item(item); count := count + 1;
7      if count = 1 then signal(empty)
8    end;
9    function remove: integer;
10   begin
11     if count = 0 then wait(empty);
▶      remove := remove_item; count := count - 1;
▶      if count = N - 1 then signal(full)
14   end;
```




C calls `remove` : takes `b` , count = 1 = N − 1 → `signal(full)` wakes P.

Step 9 / 10

**VARIABLES**

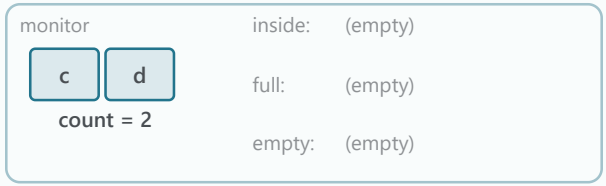
`count = 1` `buffer = [c]`

`full = []` `empty = []`

`C got = a b`

# Producer–Consumer with a Monitor: Code Trace

```
1  monitor ProducerConsumer      { N = 2 }
2    condition full, empty; integer count;
3    procedure insert(item: integer);
4    begin
5      if count = N then wait(full);
▶      insert_item(item); count := count + 1;
7      if count = 1 then signal(empty)
8    end;
9    function remove: integer;
10   begin
11     if count = 0 then wait(empty);
12     remove := remove_item; count := count - 1;
13     if count = N - 1 then signal(full)
14   end;
```




P returns from `wait` and inserts `d`: count = 2. No data was lost and no process slept forever.

Step 10 / 10

**VARIABLES**

`count = 2` `buffer = [c d]` `full = []` `empty = []` `C got = a b`

# Pthreads: Mutexes and Condition Variables

| Mutex call | Description | Condition call | Description |
|---|---|---|---|
| `pthread_mutex_init` | Creates a mutex | `pthread_cond_init` | Creates a condition variable |
| `pthread_mutex_destroy` | Destroys it | `pthread_cond_destroy` | Destroys it |
| `pthread_mutex_lock` | Acquires the lock or blocks | `pthread_cond_wait` | **Atomically** unlocks and sleeps; relocks on return |
| `pthread_mutex_trylock` | Acquires or returns `EBUSY` at once | `pthread_cond_signal` | Wakes **at least one** waiter |
| `pthread_mutex_unlock` | Releases it (owner only) | `pthread_cond_broadcast` | Wakes **all** waiters |

### Rule

Call `cond_wait` only while holding the mutex, inside a `while` loop.

### Mutex types

`NORMAL`, `ERRORCHECK`, `RECURSIVE`; `PTHREAD_PRIO_INHERIT` for priority inheritance.

### More APIs

`pthread_rwlock_*`, `pthread_spin_*`, `pthread_barrier_*`, C11 `mtx_t` / `cnd_t`.

Static init: `PTHREAD_MUTEX_INITIALIZER`, `PTHREAD_COND_INITIALIZER`. `cond_signal` affects only current *waiters*; with none, the signal is lost.

# Pthreads Producer–Consumer, Step by Step

```
1  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
2  pthread_cond_t condc = PTHREAD_COND_INITIALIZER,
3                 condp = PTHREAD_COND_INITIALIZER;
4  int buffer = 0;                    /* 0 = empty */
5
6  void *producer(void *p) {
7    for (int i = 1; i <= MAX; i++) {  /* MAX = 2 */
8      pthread_mutex_lock(&m);
9      while (buffer != 0) pthread_cond_wait(&condp, &m);
10     buffer = i;
11     pthread_cond_signal(&condc);
12     pthread_mutex_unlock(&m);
13   } return NULL;
14 }
15 void *consumer(void *p) {
16   for (int i = 1; i <= MAX; i++) {
▶      pthread_mutex_lock(&m);
18     while (buffer == 0) pthread_cond_wait(&condc, &m);
19     printf("%d\n", buffer); buffer = 0;
20     pthread_cond_signal(&condp);
21     pthread_mutex_unlock(&m);
22   } return NULL;
23 }
```

Adapted from Tanenbaum & Bos, Figure 2-32.

P — not started

C — lock(m)

mutex m: C

The consumer runs first and takes the mutex.

Step 1 / 8

**VARIABLES**

`buffer = 0`

**OUTPUT**

# Pthreads Producer–Consumer, Step by Step

```
1  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
2  pthread_cond_t condc = PTHREAD_COND_INITIALIZER,
3                 condp = PTHREAD_COND_INITIALIZER;
4  int buffer = 0;                    /* 0 = empty */
5
6  void *producer(void *p) {
7    for (int i = 1; i <= MAX; i++) {  /* MAX = 2 */
8      pthread_mutex_lock(&m);
9      while (buffer != 0) pthread_cond_wait(&condp, &m);
10     buffer = i;
11     pthread_cond_signal(&condc);
12     pthread_mutex_unlock(&m);
13   } return NULL;
14 }
15 void *consumer(void *p) {
16   for (int i = 1; i <= MAX; i++) {
17     pthread_mutex_lock(&m);
▶    while (buffer == 0) pthread_cond_wait(&condc, &m);
19     printf("%d\n", buffer); buffer = 0;
20     pthread_cond_signal(&condp);
21     pthread_mutex_unlock(&m);
22   } return NULL;
23 }
```

Adapted from Tanenbaum & Bos, Figure 2-32.

P — ready

C — cond_wait(condc) → sleeps

mutex m: free

The buffer is empty: `cond_wait` **atomically releases** the mutex and sleeps. There is no lost-wakeup window.

Step 2 / 8

**VARIABLES**

`buffer = 0`

**OUTPUT**

# Pthreads Producer–Consumer, Step by Step

```
1  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
2  pthread_cond_t condc = PTHREAD_COND_INITIALIZER,
3                 condp = PTHREAD_COND_INITIALIZER;
4  int buffer = 0;                    /* 0 = empty */
5
6  void *producer(void *p) {
7    for (int i = 1; i <= MAX; i++) {  /* MAX = 2 */
▶      pthread_mutex_lock(&m);
▶      while (buffer != 0) pthread_cond_wait(&condp, &m);
▶      buffer = i;
11     pthread_cond_signal(&condc);
12     pthread_mutex_unlock(&m);
13   } return NULL;
14 }
15 void *consumer(void *p) {
16   for (int i = 1; i <= MAX; i++) {
17     pthread_mutex_lock(&m);
18     while (buffer == 0) pthread_cond_wait(&condc, &m);
19     printf("%d\n", buffer); buffer = 0;
20     pthread_cond_signal(&condp);
21     pthread_mutex_unlock(&m);
22   } return NULL;
23 }
```

Adapted from Tanenbaum & Bos, Figure 2-32.

P — lock, buffer = 1

C — sleeping

mutex m: P

The producer takes the mutex; the buffer is empty, so it skips the loop and writes 1.

Step 3 / 8

**VARIABLES**

`buffer = 1`

**OUTPUT**

# Pthreads Producer–Consumer, Step by Step

```
1  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
2  pthread_cond_t condc = PTHREAD_COND_INITIALIZER,
3                 condp = PTHREAD_COND_INITIALIZER;
4  int buffer = 0;                    /* 0 = empty */
5
6  void *producer(void *p) {
7    for (int i = 1; i <= MAX; i++) {  /* MAX = 2 */
8      pthread_mutex_lock(&m);
9      while (buffer != 0) pthread_cond_wait(&condp, &m);
10     buffer = i;
▶      pthread_cond_signal(&condc);
▶      pthread_mutex_unlock(&m);
13   } return NULL;
14 }
15 void *consumer(void *p) {
16   for (int i = 1; i <= MAX; i++) {
17     pthread_mutex_lock(&m);
18     while (buffer == 0) pthread_cond_wait(&condc, &m);
19     printf("%d\n", buffer); buffer = 0;
20     pthread_cond_signal(&condp);
21     pthread_mutex_unlock(&m);
22   } return NULL;
23 }
```

Adapted from Tanenbaum & Bos, Figure 2-32.

| | |
|---|---|
| P | signal(condc), unlock |
| C | woke, waiting for mutex |

mutex m: free

`signal` wakes the consumer, but it cannot return **until it reacquires the mutex** (Mesa).

Step 4 / 8

**VARIABLES**

`buffer = 1`

**OUTPUT**

# Pthreads Producer–Consumer, Step by Step

```
1  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
2  pthread_cond_t condc = PTHREAD_COND_INITIALIZER,
3                 condp = PTHREAD_COND_INITIALIZER;
4  int buffer = 0;                    /* 0 = empty */
5
6  void *producer(void *p) {
7    for (int i = 1; i <= MAX; i++) {  /* MAX = 2 */
▶      pthread_mutex_lock(&m);
▶      while (buffer != 0) pthread_cond_wait(&condp, &m);
10     buffer = i;
11     pthread_cond_signal(&condc);
12     pthread_mutex_unlock(&m);
13   } return NULL;
14 }
15 void *consumer(void *p) {
16   for (int i = 1; i <= MAX; i++) {
17     pthread_mutex_lock(&m);
18     while (buffer == 0) pthread_cond_wait(&condc, &m);
19     printf("%d\n", buffer); buffer = 0;
20     pthread_cond_signal(&condp);
21     pthread_mutex_unlock(&m);
22   } return NULL;
23 }
```

Adapted from Tanenbaum & Bos, Figure 2-32.

P — i = 2: cond_wait(condp)

C — waiting for mutex

mutex m: P

The producer may win the race and retake the mutex: the buffer is full → it waits and releases the mutex.

Step 5 / 8

**VARIABLES**

`buffer = 1`

**OUTPUT**

# Pthreads Producer–Consumer, Step by Step

```
1  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
2  pthread_cond_t condc = PTHREAD_COND_INITIALIZER,
3                 condp = PTHREAD_COND_INITIALIZER;
4  int buffer = 0;                    /* 0 = empty */
5
6  void *producer(void *p) {
7    for (int i = 1; i <= MAX; i++) {  /* MAX = 2 */
8      pthread_mutex_lock(&m);
9      while (buffer != 0) pthread_cond_wait(&condp, &m);
10     buffer = i;
11     pthread_cond_signal(&condc);
12     pthread_mutex_unlock(&m);
13   } return NULL;
14 }
15 void *consumer(void *p) {
16   for (int i = 1; i <= MAX; i++) {
17     pthread_mutex_lock(&m);
▶      while (buffer == 0) pthread_cond_wait(&condc, &m);
▶      printf("%d\n", buffer); buffer = 0;
20     pthread_cond_signal(&condp);
21     pthread_mutex_unlock(&m);
22   } return NULL;
23 }
```

Adapted from Tanenbaum & Bos, Figure 2-32.

P — sleeping

C — passed while, printed, buffer = 0

mutex m: C

The consumer takes the mutex, retests the `while` condition (1 ≠ 0), and consumes the item.

Step 6 / 8

**VARIABLES**

`buffer = 0`

**OUTPUT**

```
1
```

# Pthreads Producer–Consumer, Step by Step

```
1  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
2  pthread_cond_t condc = PTHREAD_COND_INITIALIZER,
3                 condp = PTHREAD_COND_INITIALIZER;
4  int buffer = 0;                    /* 0 = empty */
5
6  void *producer(void *p) {
7    for (int i = 1; i <= MAX; i++) {  /* MAX = 2 */
8      pthread_mutex_lock(&m);
▶      while (buffer != 0) pthread_cond_wait(&condp, &m);
▶      buffer = i;
11     pthread_cond_signal(&condc);
12     pthread_mutex_unlock(&m);
13   } return NULL;
14 }
15 void *consumer(void *p) {
16   for (int i = 1; i <= MAX; i++) {
17     pthread_mutex_lock(&m);
18     while (buffer == 0) pthread_cond_wait(&condc, &m);
19     printf("%d\n", buffer); buffer = 0;
▶      pthread_cond_signal(&condp);
▶      pthread_mutex_unlock(&m);
22   } return NULL;
23 }
```

Adapted from Tanenbaum & Bos, Figure 2-32.

P — woke, buffer = 2

C — signal(condp), unlock

mutex m: P

The consumer signals `condp`; the producer wakes, finds the buffer empty, and writes 2.

Step 7 / 8

**VARIABLES**

`buffer = 2`

**OUTPUT**

```
1
```

# Pthreads Producer–Consumer, Step by Step

```
1  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
2  pthread_cond_t condc = PTHREAD_COND_INITIALIZER,
3                 condp = PTHREAD_COND_INITIALIZER;
4  int buffer = 0;                    /* 0 = empty */
5
6  void *producer(void *p) {
7    for (int i = 1; i <= MAX; i++) {  /* MAX = 2 */
8      pthread_mutex_lock(&m);
9      while (buffer != 0) pthread_cond_wait(&condp, &m);
10     buffer = i;
11     pthread_cond_signal(&condc);
12     pthread_mutex_unlock(&m);
13   } return NULL;
14 }
15 void *consumer(void *p) {
16   for (int i = 1; i <= MAX; i++) {
▶      pthread_mutex_lock(&m);
▶      while (buffer == 0) pthread_cond_wait(&condc, &m);
▶      printf("%d\n", buffer); buffer = 0;
20     pthread_cond_signal(&condp);
21     pthread_mutex_unlock(&m);
22   } return NULL;
23 }
```

Adapted from Tanenbaum & Bos, Figure 2-32.

P: done

C: printed 2

mutex m: C

Second round: the consumer takes 2. The output is the same in any order.

Step 8 / 8

**VARIABLES**

`buffer = 0`

**OUTPUT**

```
1
2
```

# POSIX Semaphores

```c
#include <semaphore.h>

/* unnamed: threads of one process */
sem_t empty, full;
sem_init(&empty, 0, N);  /* 0: in-process */
sem_init(&full,  0, 0);
sem_wait(&empty); /* … put … */  sem_post(&full);
sem_destroy(&empty);

/* named: between unrelated processes */
sem_t *s = sem_open("/os06", O_CREAT, 0644, 1);
sem_wait(s);  /* CS */  sem_post(s);
sem_close(s);
sem_unlink("/os06");
```

| | |
|---|---|
| `sem_wait` | P / down: blocks if 0 |
| `sem_trywait`, `sem_timedwait` | Try without waiting or with a timeout |
| `sem_post` | V / up; safe in a signal handler |
| `sem_getvalue` | Current value (informational only) |

- If the second argument of `sem_init` is 1, the semaphore must be in shared memory.
- macOS lacks unnamed semaphores (`sem_init` is deprecated); use a named one or `dispatch_semaphore`.
- Java: `java.util.concurrent.Semaphore`: `acquire()`, `release()`, optional fair mode.

# Java: synchronized, wait, and notify

```
public class OneSlot {
  private int slot = 0;             /* 0 = empty */

  synchronized void put(int x) throws InterruptedException {
    while (slot != 0) wait();       /* releases lock */
    slot = x;
    notifyAll();
  }
  synchronized int take() throws InterruptedException {
    while (slot == 0) wait();
    int x = slot;
    slot = 0;
    notifyAll();
    return x;
  }
  /* main: a producer puts 10, 20, 30, 40;
     main calls take() four times and prints */
}
```

- Every Java object is a **monitor**: an intrinsic lock plus a single wait set.
- A `synchronized` method or block takes the lock and releases it on exit (even on an exception); the lock is **reentrant**.
- `wait()` releases the lock and sleeps; `notify()` wakes one, `notifyAll()` wakes all. The semantics are **Mesa**: `while` is required.
- Producer and consumer share one wait set, so `notifyAll` is safer.

OUTPUT (JDK 21, SAME ON EVERY RUN)

```
took 10
took 20
took 30
took 40
```

# Java: ReentrantLock and Condition

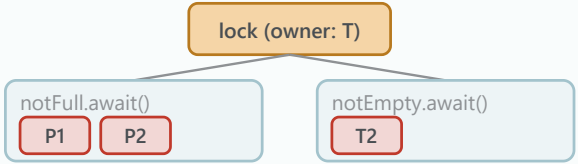
```
class Buffer {
  private final int[] buf = new int[2];
  private int head = 0, tail = 0, count = 0;
  private final ReentrantLock lock = new ReentrantLock();
  private final Condition notFull  = lock.newCondition();
  private final Condition notEmpty = lock.newCondition();

  void put(int x) throws InterruptedException {
    lock.lock();
    try {
      while (count == buf.length) notFull.await();
      buf[tail] = x; tail = (tail + 1) % buf.length; count++;
      notEmpty.signal();
    } finally { lock.unlock(); }
  }
  /* take(): while (count == 0) notEmpty.await(); …
             notFull.signal();                      */
}
/* producer puts 1..5; main calls take() 5 times */
```

- `java.util.concurrent.locks`: one lock can have **several conditions**; `signal` wakes the right group.
- `tryLock()`, timed and interruptible locking, and an optional **fair** mode.
- Always call `unlock()` in a `finally` block.
- Java 21 virtual threads blocking inside `synchronized` "pinned" their carrier thread; JDK 24 (JEP 491) fixed this.

OUTPUT (JDK 21, THREE RUNS)

```
1 2 3 4 5 | sum = 15
1 2 3 4 5 | sum = 15
1 2 3 4 5 | sum = 15
```



# Priority Inversion

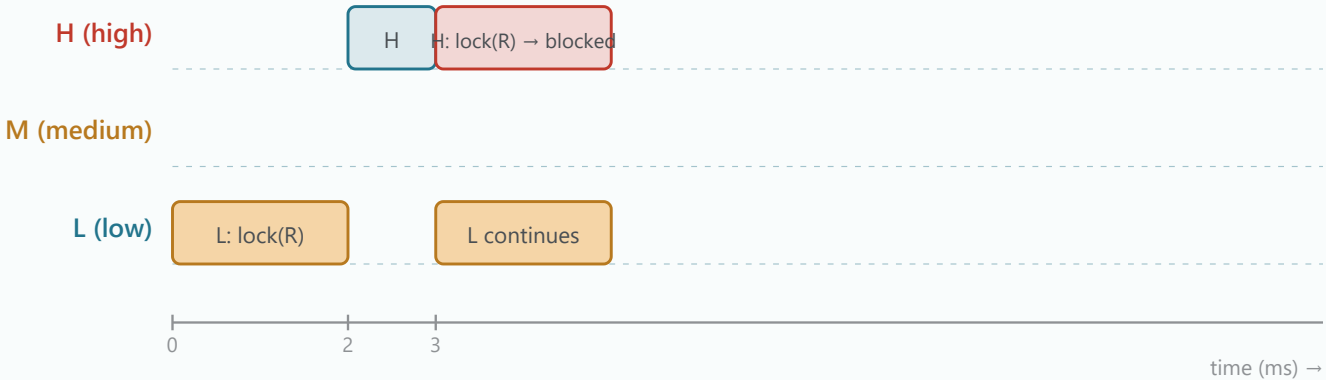



t = 0: low-priority L acquires the lock on shared resource R.

Step 1 / 4

Priority inversion is **the problem**, not a solution; the remedies are priority **inheritance** and the priority **ceiling**. On a single core with strict preemptive priority scheduling, an H spinning on a spinlock never yields the CPU; lower-priority L never runs and never releases the lock: the system hangs.

# Priority Inversion

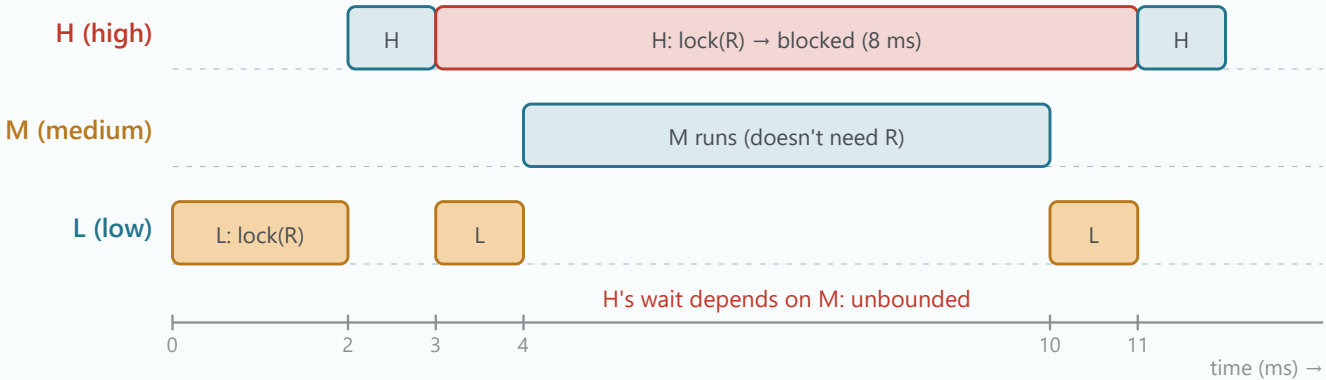



t = 2: high-priority H arrives and preempts L; at t = 3 it requests R and **blocks**. A short wait is expected: when L leaves, H enters.



Priority inversion is **the problem**, not a solution; the remedies are priority **inheritance** and the priority **ceiling**. On a single core with strict preemptive priority scheduling, an H spinning on a spinlock never yields the CPU; lower-priority L never runs and never releases the lock: the system hangs.

# Priority Inversion

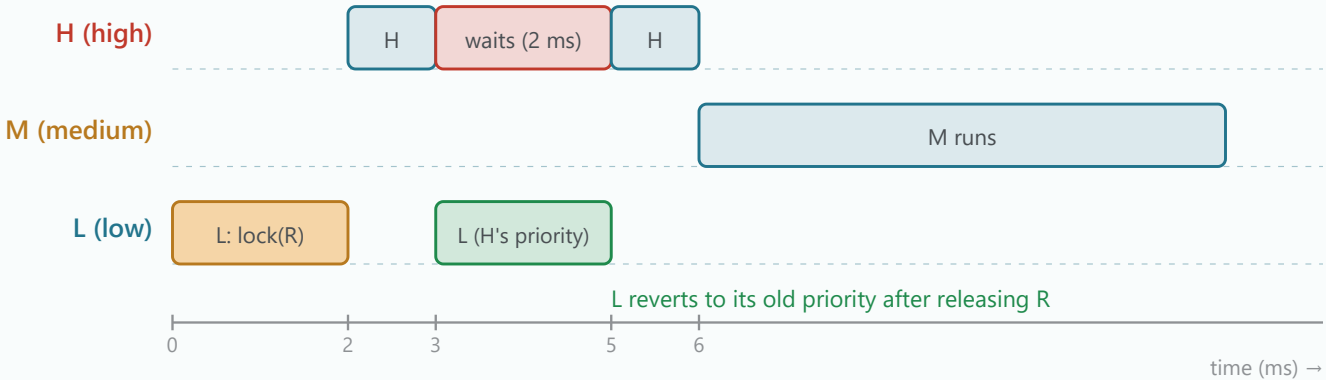



But medium-priority M, which has nothing to do with R, preempts L and runs for a long time. H is indirectly **waiting for M**: **priority inversion**. If M runs from t = 4 to 10, L releases R only at t = 11: H waits **8 ms**, and the longer M runs, the longer H waits, without bound.

Step 3 / 4

Priority inversion is **the problem**, not a solution; the remedies are priority **inheritance** and the priority **ceiling**. On a single core with strict preemptive priority scheduling, an H spinning on a spinlock never yields the CPU; lower-priority L never runs and never releases the lock: the system hangs.

# Priority Inversion



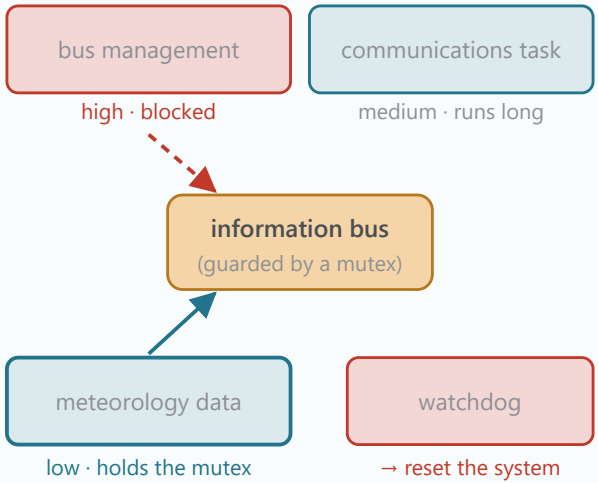

**Priority inheritance**: while H waits for R, L temporarily inherits H's priority, so M cannot cut in. L leaves and H runs at once: in the same example H waits only **2 ms**.



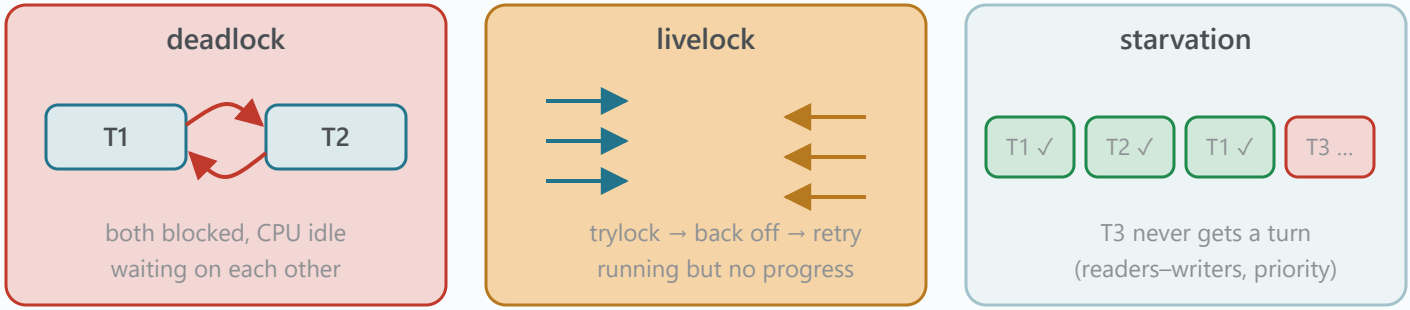
Priority inversion is **the problem**, not a solution; the remedies are priority **inheritance** and the priority **ceiling**. On a single core with strict preemptive priority scheduling, an H spinning on a spinlock never yields the CPU; lower-priority L never runs and never releases the lock: the system hangs.

# Mars Pathfinder (1997)

- July 1997: on the surface of Mars, Pathfinder, running the VxWorks real-time OS, **kept resetting itself**.
- While a low-priority meteorology task held the bus mutex, the high-priority bus management task blocked; a medium-priority communications task got in between.
- When the management task missed its deadline, a **watchdog timer** concluded the system had hung and forced a full reset.
- JPL engineers reproduced the bug on a ground replica; the mutex's **priority inheritance** option was turned on by a remotely uploaded patch.
- Today: `PTHREAD_PRIO_INHERIT`, Linux PI-futexes, and `rt_mutex`; PREEMPT_RT, merged into mainline in Linux 6.12, relies on this mechanism.



# Liveness Problems




**Deadlock**

Every process in a set waits for a resource held by another process in the set. It arises when the four necessary conditions hold at once.

**Livelock**

Processes are not blocked; they keep reacting to each other and repeat the same steps endlessly. Remedy: random backoff (e.g., Ethernet).

**Starvation**

A process keeps yielding to others. Remedies: FIFO queues, fair locks, aging. The bounded-waiting requirement rules it out.

# An Introduction to Lock-Free Structures

```
void push(node_t *n) {
  node_t *old = atomic_load(&top);
  do {
    n->next = old;
  } while (!atomic_compare_exchange_weak(
               &top, &old, n));
}  /* on failure old = current top */
```

| | |
|---|---|
| Blocking | If the lock holder stalls, everyone stalls |
| Lock-free | The system as a whole always makes progress |
| Wait-free | Every thread finishes in a bounded number of steps |

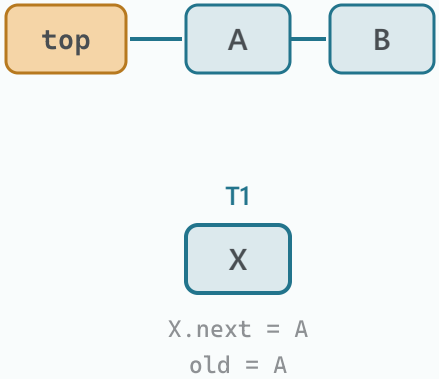



Stack: `top → A → B`. A single atomic pointer instead of a lock.

# An Introduction to Lock-Free Structures

```
void push(node_t *n) {
  node_t *old = atomic_load(&top);
  do {
    n->next = old;
  } while (!atomic_compare_exchange_weak(
               &top, &old, n));
}  /* on failure old = current top */
```

| | |
|---|---|
| Blocking | If the lock holder stalls, everyone stalls |
| Lock-free | The system as a whole always makes progress |
| Wait-free | Every thread finishes in a bounded number of steps |

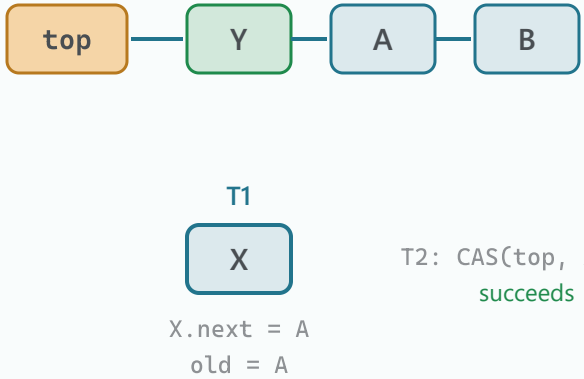



T1 prepares to push X: `old = A`, `X.next = A`.

# An Introduction to Lock-Free Structures

```
void push(node_t *n) {
  node_t *old = atomic_load(&top);
  do {
    n->next = old;
  } while (!atomic_compare_exchange_weak(
               &top, &old, n));
}  /* on failure old = current top */
```

| | |
|---|---|
| Blocking | If the lock holder stalls, everyone stalls |
| Lock-free | The system as a whole always makes progress |
| Wait-free | Every thread finishes in a bounded number of steps |

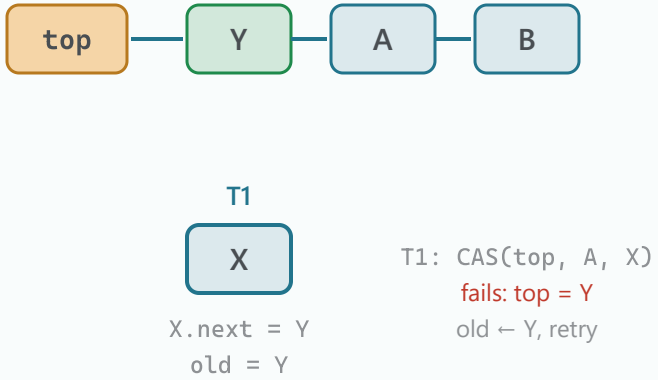



Meanwhile T2 pushes Y: `CAS(top, A, Y)` succeeds.

# An Introduction to Lock-Free Structures

```
void push(node_t *n) {
  node_t *old = atomic_load(&top);
  do {
    n->next = old;
  } while (!atomic_compare_exchange_weak(
               &top, &old, n));
}  /* on failure old = current top */
```

| | |
|---|---|
| Blocking | If the lock holder stalls, everyone stalls |
| Lock-free | The system as a whole always makes progress |
| Wait-free | Every thread finishes in a bounded number of steps |




T1's CAS fails (top is now Y). C11 `compare_exchange` writes the current value into `old`; T1 retries with `X.next = Y`.

# An Introduction to Lock-Free Structures

```
void push(node_t *n) {
  node_t *old = atomic_load(&top);
  do {
    n->next = old;
  } while (!atomic_compare_exchange_weak(
               &top, &old, n));
}  /* on failure old = current top */
```

| | |
|---|---|
| Blocking | If the lock holder stalls, everyone stalls |
| Lock-free | The system as a whole always makes progress |
| Wait-free | Every thread finishes in a bounded number of steps |




**T1: CAS(top, Y, X) succeeds**

no lock; even if one thread stalls, the others keep making progress

The second CAS succeeds. **The ABA problem**: if top goes A→B→A during a `pop`, CAS cannot tell; you need a version counter or safe memory reclamation (hazard pointers, epochs).

# Summary

| Tool | What it provides | When to use |
|---|---|---|
| Peterson, strict alternation | Exclusion using only loads and stores (assuming SC) | Conceptual; not used in real code |
| Memory barrier | Ordering and visibility of loads and stores | Inside lock and atomic implementations |
| TAS, XCHG, CAS | Atomic read-modify-write | The foundation of locks and lock-free structures |
| Atomic variable | Indivisible update of one variable | Counters, flags, reference counts |
| Spinlock | Exclusion without sleeping | Multicore, very short CS, in the kernel |
| Mutex | Exclusion; waiters sleep | General-purpose critical sections |
| Semaphore | Counting, exclusion, and ordering | N identical resources, event notification |
| Monitor + condition variable | Structured exclusion + condition waiting | Complex conditions (Java, pthreads) |
| Reader–writer lock | Many readers, one writer | Read-mostly data |

Every tool provides *mutual exclusion*; *progress* and *bounded waiting* come from correct use and fair queues. Priority inversion and deadlock need separate thought.

# Exercises

The parent calls `thr_join()`; the child thread calls `thr_exit()` when it is done.

```
/* (a) no state variable */
void thr_exit() { lock(&m);
  signal(&c); unlock(&m); }
void thr_join() { lock(&m);
  wait(&c, &m); unlock(&m); }
```

```
/* (b) no lock, uses if */
void thr_exit() { done = 1;
  signal(&c); }
void thr_join() {
  if (done == 0) wait(&c); }
```

1. In (a), what happens if the child finishes before the parent enters `thr_join()`?
2. In (b), give an interleaving in which the parent sleeps forever.
3. Implement `sem_down` / `sem_up` using only one mutex and one condition variable.
4. In an 8-core approximate counter with S = 1024, how far can `get()` lag behind the actual value? What if S = 1?

▶ **Answers**

SUMMARY

# Review Questions (1/2)

1. State the three requirements for a critical-section solution. Which one does strict alternation violate?
2. Why can Peterson's solution fail on x86? How is it fixed?
3. Which requirement does a simple `test_and_set` spinlock fail to guarantee?
4. In a CAS-based increment, what happens if two threads read the same value? Why is the final value correct?

▶ **Answers**

SUMMARY

# Review Questions (2/2)

5. A semaphore starts at 2 and three processes call `wait` in a row. What are the value and the queue (Silberschatz implementation)?
6. In the bounded-buffer solution, what happens if the producer's two `down` calls are swapped?
7. Why must a condition-variable wait under Mesa semantics be inside a `while`?
8. Define priority inversion. How does priority inheritance solve it?

▶ **Answers**

# References


- A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10th ed., Wiley, 2018 (Chapter 6: Synchronization Tools, Chapter 7: Synchronization Examples).
- A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4th ed. (2014) / 5th ed. (2023), Pearson (§2.3 Interprocess Communication, §2.5 Classical IPC Problems).
- R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), ostep.org: "Locks", "Condition Variables", "Semaphores", "Common Concurrency Problems".
- A. B. Downey, *The Little Book of Semaphores*, 2nd ed., Green Tea Press (free): readers–writers and philosophers solutions.
- POSIX.1-2017 (IEEE Std 1003.1): `pthread_mutex_lock`, `pthread_cond_wait`, `sem_overview(7)`; the Linux `futex(2)` man page.
- B. Goetz et al., *Java Concurrency in Practice*, 2006; Java SE 21 API: `java.util.concurrent`; JEP 491.
- G. L. Peterson, “Myths About the Mutual Exclusion Problem”, *IPL*, 1981; M. Jones, “What really happened on Mars?”, 1997.

# The End

## Chapter 6: Synchronization

Previous: Chapter 5: CPU Scheduling · Next: Chapter 7: Deadlocks