Chapter 4: Threads

Operating Systems

Processes and threads | Shared and private state | Multithreaded server | Concurrency and parallelism | Amdahl's law | User and kernel threads | 1:1, N:1, M:N models | Pthreads and Java | Pools and fork-join | OpenMP and GCD | Race conditions | TLS, cancellation, signals

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_04_Threads.html

Learning Outcomes

Concept

Distinguish a thread from a process; say which state is shared and which is private to a thread.

Performance

Tell concurrency from parallelism and compute the limit on speedup with Amdahl's law.

Models

Compare user-level and kernel-level threads and the N:1, 1:1 and M:N mapping models.

Programming

Create and join threads with Pthreads and Java; recognize implicit approaches such as pools, fork-join, OpenMP and GCD.

Issues

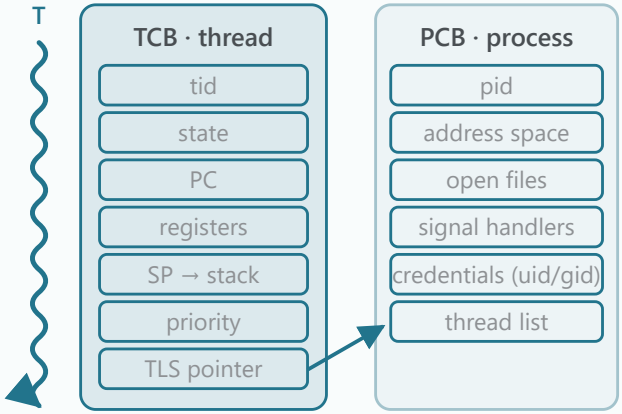
Explain race conditions, non-reentrant code, TLS, cancellation, signals and the behavior of `fork()`.

Real systems

Describe current implementations such as Linux `clone()`, Java 21 virtual threads and Go.

What Is a Thread?

- **Process:** a program in execution; it **owns resources** such as the address space and open files.
- **Thread:** a **flow of execution** within a process; the basic unit of CPU use and scheduling.
- A process contains one or more threads; all share **the same address space**.
- Each thread has its own **PC, register set, stack and state**, kept in its TCB (thread control block).
- Tanenbaum's distinction: a process *groups resources*; a thread *executes*.

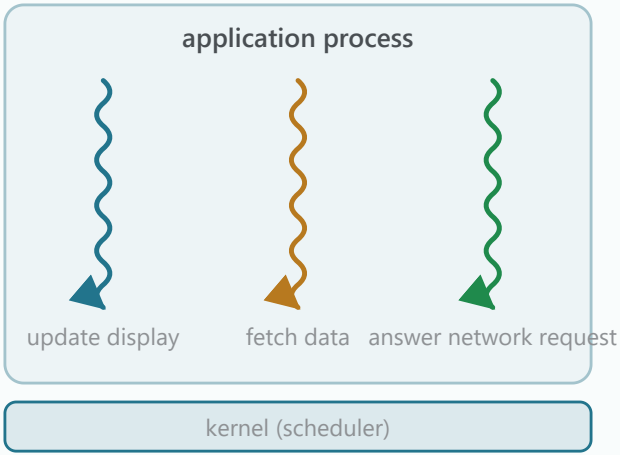


many TCBs → one PCB
(both are task_struct on Linux)

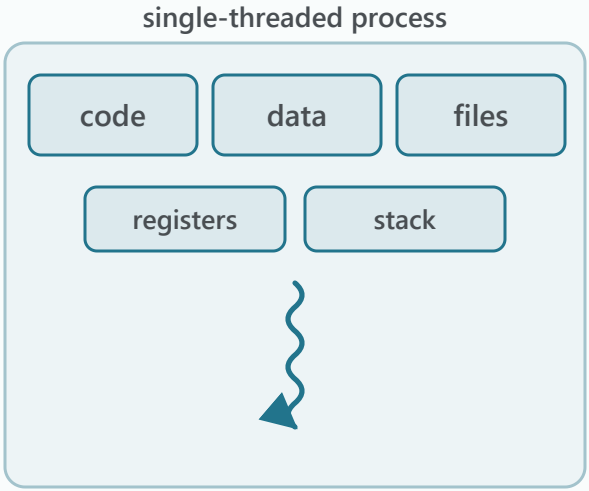
The old name is **lightweight process** (LWP): creating and switching threads is cheap compared with processes, because the address space is not copied.

Why Multiple Threads?

- Most modern applications are **multithreaded**: browsers, IDEs, game engines, databases, web servers.
- Different tasks run in separate threads:
  - updating the display, fetching data, answering network requests, spell checking…
- While one task waits (disk, network), the others **keep making progress**.
- Creating a process is expensive; creating a thread is much cheaper.
- Flows written like sequential code **simplify** the program.
- The OS kernel is multithreaded too (Linux: `kthreadd`, `kworker`, `ksoftirqd`).

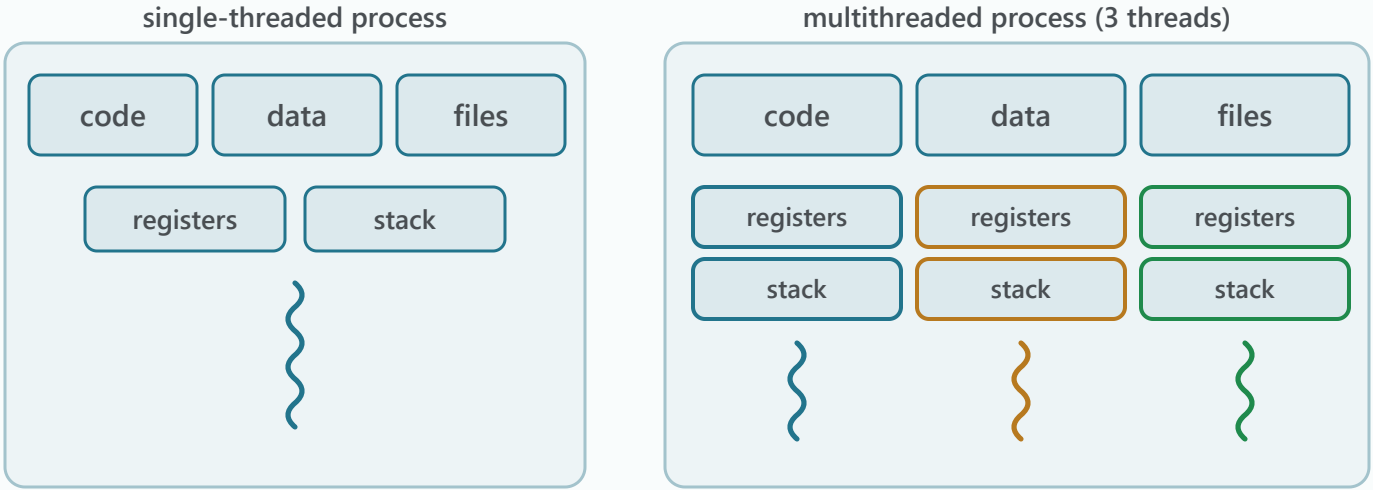


Single-Threaded and Multithreaded Processes



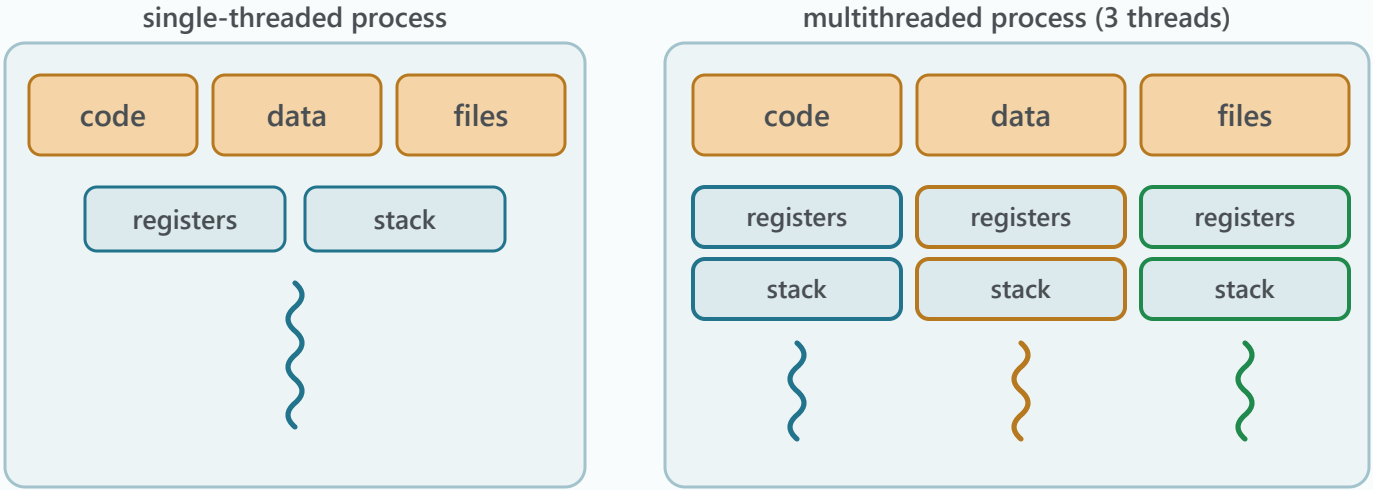
A **single-threaded** process: one flow of execution, one PC, one register set and one stack.

Single-Threaded and Multithreaded Processes



A **multithreaded** process: code, data and files exist **once**; each flow has its own registers and stack.

Single-Threaded and Multithreaded Processes



top row: shared · bottom row: per thread

all three flows see the same code and global data

The orange row is the **shared** resources: when one thread writes a global variable, the others see it at once.

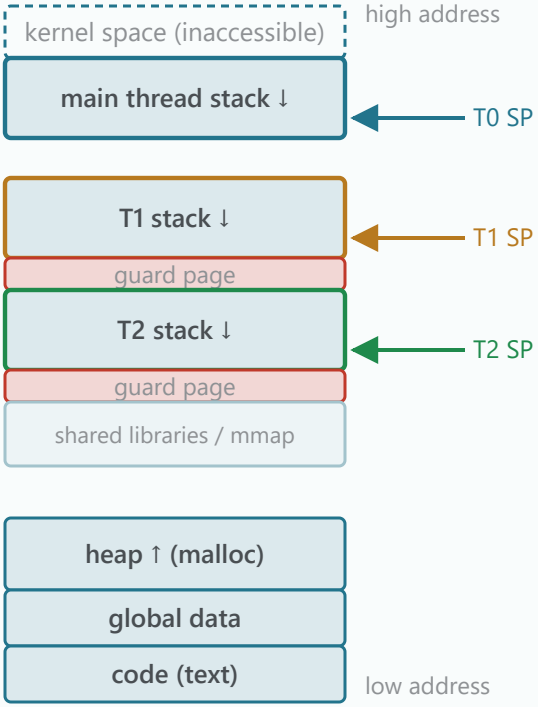
Shared and Private Items

Per process (shared by all threads)
Address space: code, global variables, heap
Open files, sockets
Child processes
Pending alarms (timers)
Signal handlers
Identity and accounting information

Per thread (private)
Program counter (PC)
Registers
Stack and stack pointer
State: running, ready, waiting, terminated
Thread ID, signal mask
Thread-local storage (TLS), `errno`

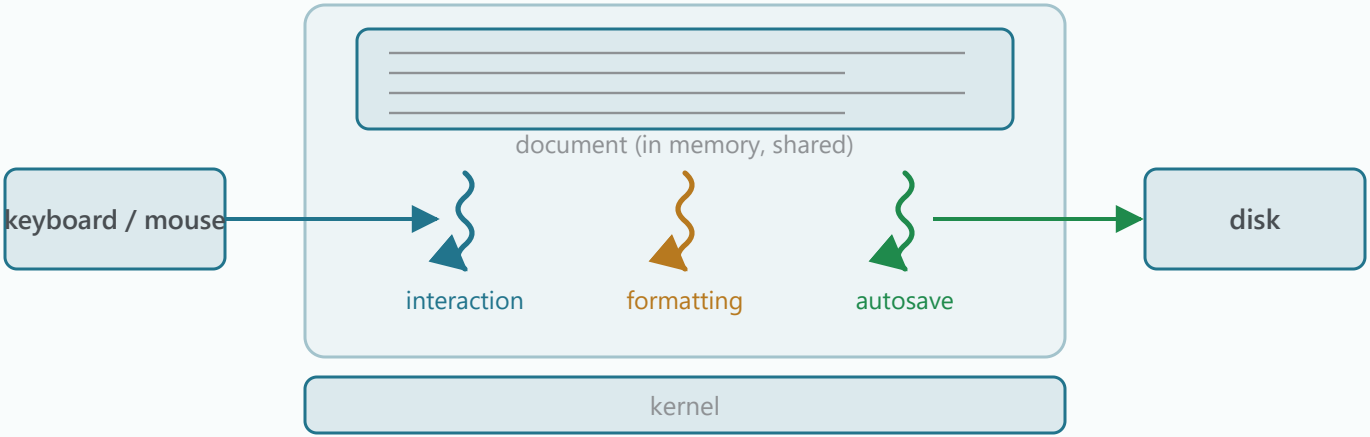
The stack is **private but not protected**: since all live in one address space, a bad pointer can corrupt another thread's stack. Access to shared data needs synchronization.

Each Thread Has Its Own Stack



- Function calls and local variables must be kept separately for each flow: that is why **every thread has its own stack**.
- On Linux/glibc a new thread's stack is allocated with `mmap`; the default size is usually the `ulimit -s` value (often 8 MiB).
- The **guard page** at the end of the stack is inaccessible; an overflowing stack raises `SIGSEGV` instead of silently corrupting its neighbor.
- Setting the size: `pthread_attr_setstacksize`, `-Xss` in Java.
- The stack frames of Java virtual threads live on the heap, which allows millions of virtual threads.

Example: A Word Processor with Three Threads



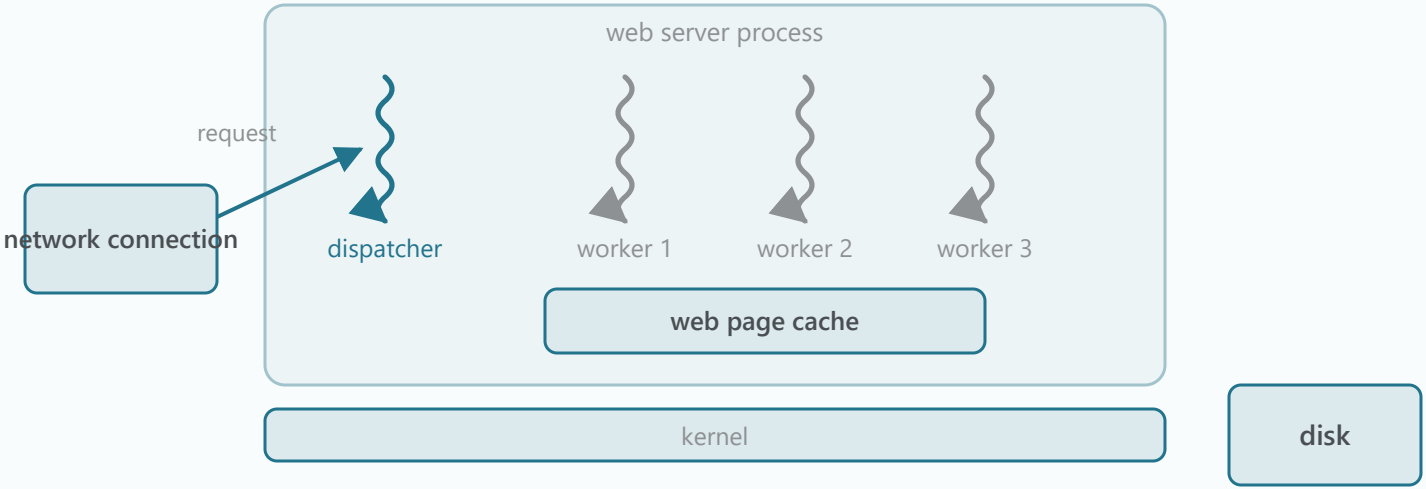
Interaction
Handles key and mouse events at once; the UI never freezes.

Formatting
Reflows hundreds of pages in the background when a line is deleted.

Autosave
Writes the document to disk periodically; the user never waits on its I/O.

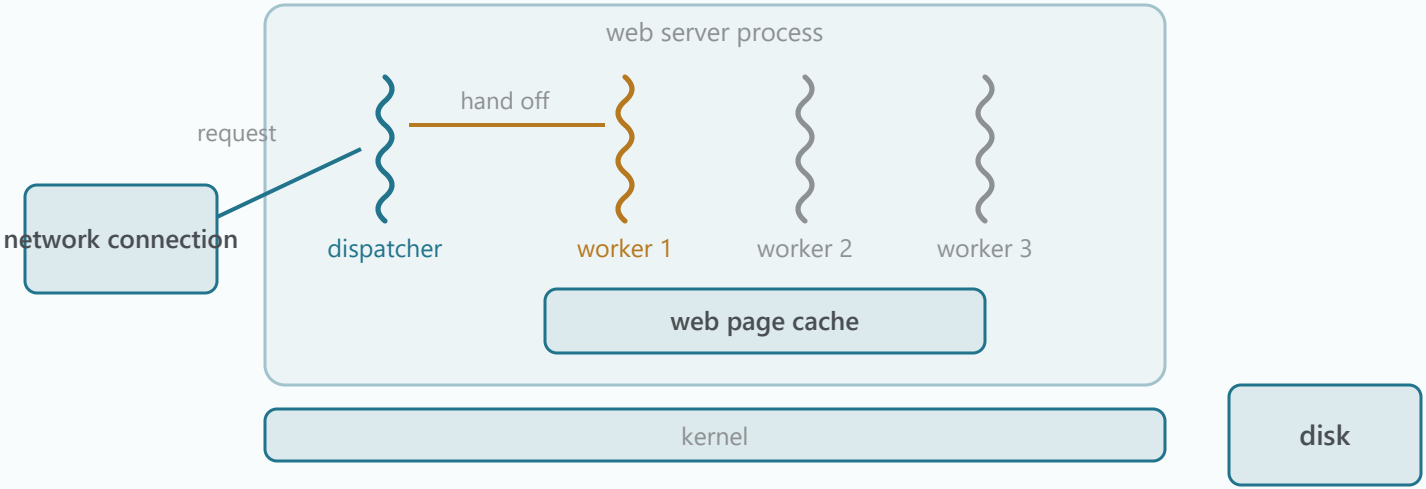
Three separate processes could not do this: all three must work on **the same document**. The shared address space is the key advantage here.

A Multithreaded Web Server



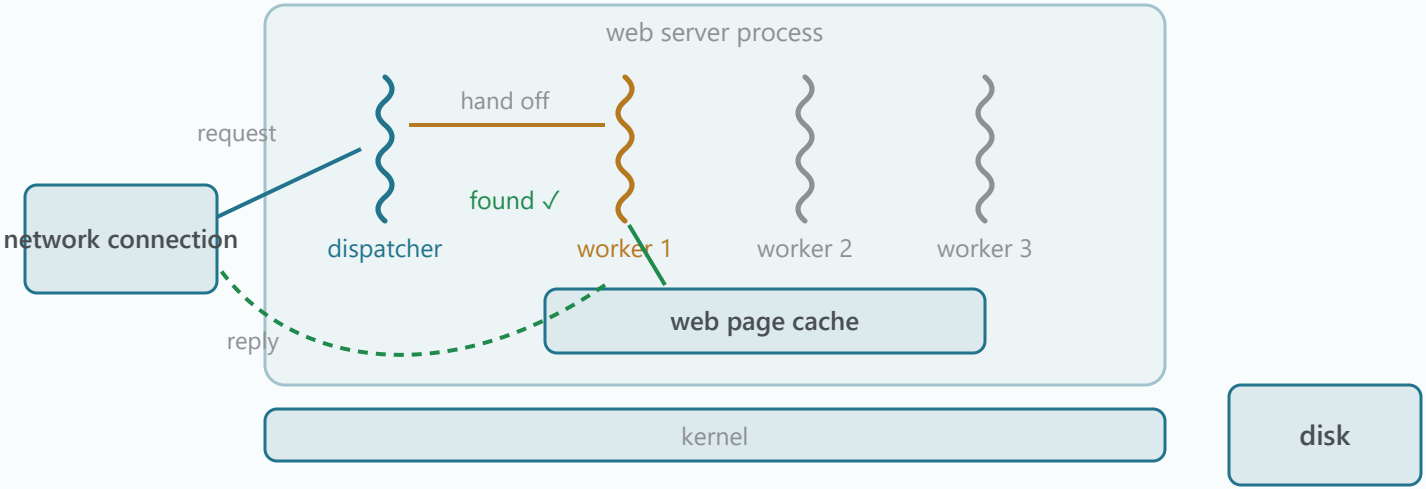
A request arrives: the **dispatcher** thread reads it from the network.

A Multithreaded Web Server



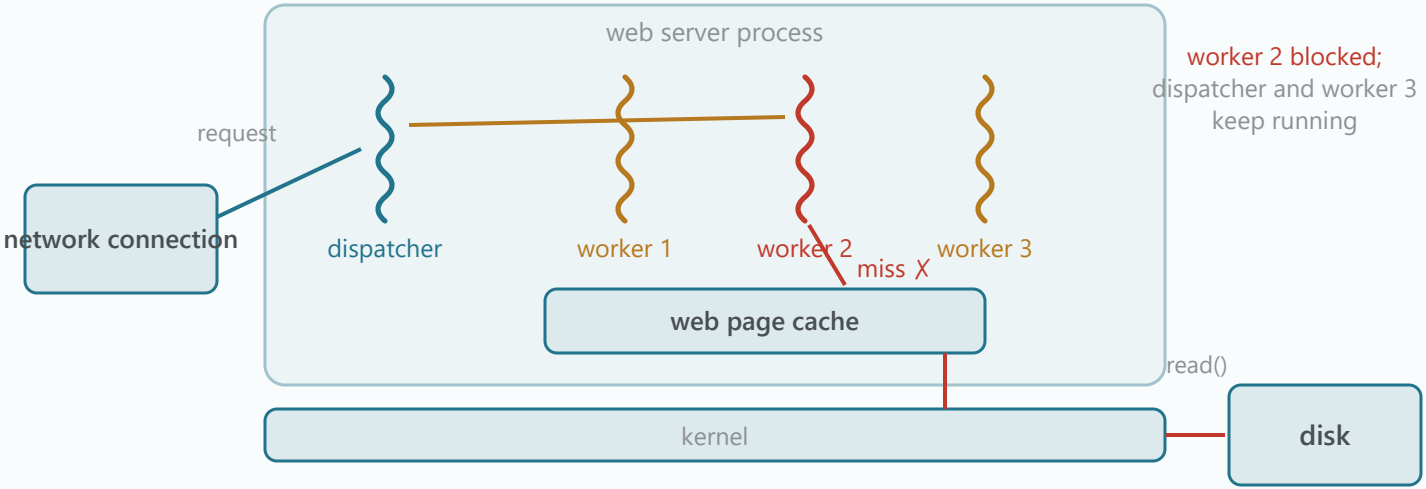
The dispatcher picks an idle **worker** thread, hands it the request and wakes it up.

A Multithreaded Web Server



Worker 1 **finds the page in the cache** and sends the reply; no disk access is needed.

A Multithreaded Web Server



A new request goes to worker 2; if the page is not cached, worker 2 **blocks** on the disk. Meanwhile the dispatcher and worker 3 handle other requests.

Dispatcher and Worker Code

(a) dispatcher thread

```
while (TRUE) {
  get_next_request(&buf);   /* from net */
  handoff_work(&buf);       /* to worker */
}
```

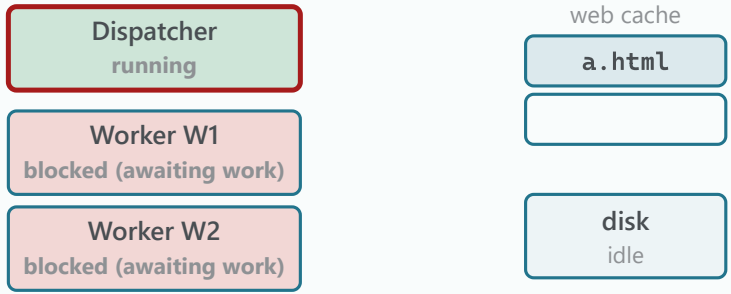
(b) worker thread

```
while (TRUE) {
  wait_for_work(&buf);
  look_for_page_in_cache(&buf, &page);
  if (page_not_in_cache(&page))
    read_page_from_disk(&buf, &page);
  return_page(&page);
}
```

- Both pieces of code are **sequential** and readable: a blocking `read` stops only that worker.
- In a single-threaded server, a page that is not cached blocks the whole process; the CPU idles while the disk is awaited.
- In a multithreaded server the work is handed to a worker and the server keeps accepting new requests.

Code Trace: A Multithreaded Web Server

```
1  /* dispatcher thread */
2  while (TRUE) {
3    get_next_request(&buf);
4    handoff_work(&buf);
5  }
6  /* worker thread */
7  while (TRUE) {
8    wait_for_work(&buf);
9    look_for_page_in_cache(&buf, &page);
10   if (page_not_in_cache(&page))
11     read_page_from_disk(&buf, &page);
12   return_page(&page);
13 }
```



The server process has three threads: the dispatcher listens on the network, the two workers are blocked in `wait_for_work`. Only a.html is in the cache.

Step 1 / 11

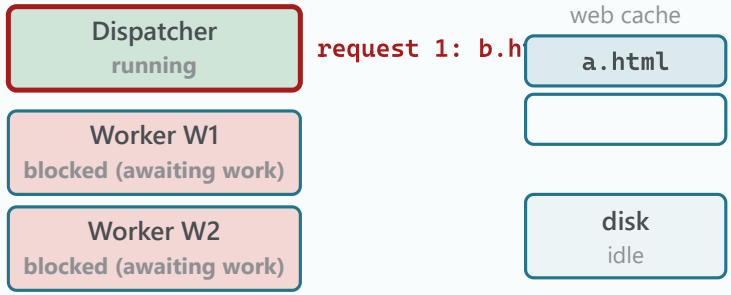
VARIABLES

`buf = ?`

OUTPUT

Code Trace: A Multithreaded Web Server

```
1  /* dispatcher thread */
2  while (TRUE) {
▶    get_next_request(&buf);
4    handoff_work(&buf);
5  }
6  /* worker thread */
7  while (TRUE) {
8    wait_for_work(&buf);
9    look_for_page_in_cache(&buf, &page);
10   if (page_not_in_cache(&page))
11     read_page_from_disk(&buf, &page);
12   return_page(&page);
13 }
```



Request 1 arrives: **b.html**.

Step 2 / 11

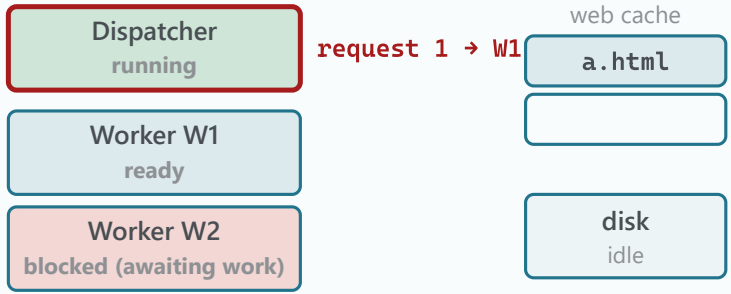
VARIABLES

`buf = b.html`

OUTPUT

Code Trace: A Multithreaded Web Server

```
1  /* dispatcher thread */
2  while (TRUE) {
3    get_next_request(&buf);
4    handoff_work(&buf);
5  }
6  /* worker thread */
7  while (TRUE) {
8    wait_for_work(&buf);
9    look_for_page_in_cache(&buf, &page);
10   if (page_not_in_cache(&page))
11     read_page_from_disk(&buf, &page);
12   return_page(&page);
13 }
```



`handoff_work` gives the request to the idle worker, W1, and wakes it up. The dispatcher goes straight back to the network.

Step 3 / 11

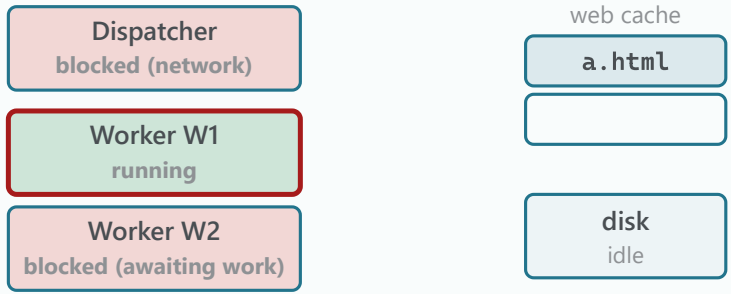
VARIABLES

`buf = b.html`

OUTPUT

Code Trace: A Multithreaded Web Server

```
1  /* dispatcher thread */
2  while (TRUE) {
3    get_next_request(&buf);
4    handoff_work(&buf);
5  }
6  /* worker thread */
7  while (TRUE) {
▶    wait_for_work(&buf);
▶    look_for_page_in_cache(&buf, &page);
▶    if (page_not_in_cache(&page))
11     read_page_from_disk(&buf, &page);
12   return_page(&page);
13 }
```



W1 looks for the page in the cache: **not there** (a miss).

Step 4 / 11

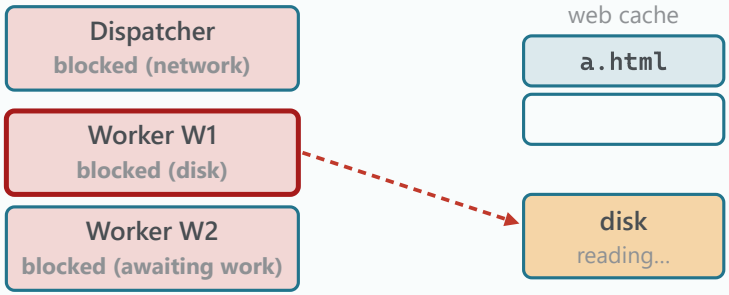
VARIABLES

`W1.buf = b.html` `W1.page = none`

OUTPUT

Code Trace: A Multithreaded Web Server

```
1  /* dispatcher thread */
2  while (TRUE) {
3    get_next_request(&buf);
4    handoff_work(&buf);
5  }
6  /* worker thread */
7  while (TRUE) {
8    wait_for_work(&buf);
9    look_for_page_in_cache(&buf, &page);
10   if (page_not_in_cache(&page))
▶      read_page_from_disk(&buf, &page);
12   return_page(&page);
13 }
```



W1 **blocks** to read from disk. Only W1 stops; the other threads in the same process keep running.

Step 5 / 11

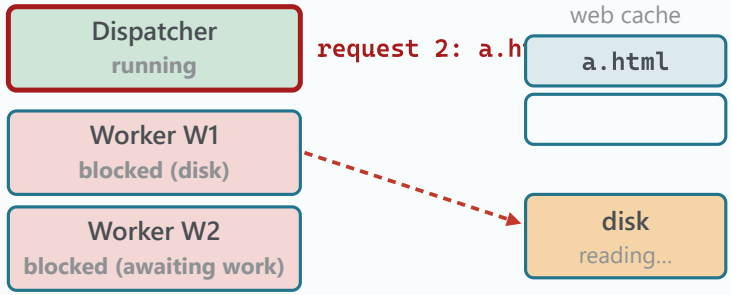
VARIABLES

`W1.buf = b.html` `W1.page = none`

OUTPUT

Code Trace: A Multithreaded Web Server

```
1  /* dispatcher thread */
2  while (TRUE) {
▶    get_next_request(&buf);
4    handoff_work(&buf);
5  }
6  /* worker thread */
7  while (TRUE) {
8    wait_for_work(&buf);
9    look_for_page_in_cache(&buf, &page);
10   if (page_not_in_cache(&page))
11     read_page_from_disk(&buf, &page);
12   return_page(&page);
13 }
```



While the disk works, the dispatcher takes request 2: **a.html**.

Step 6 / 11

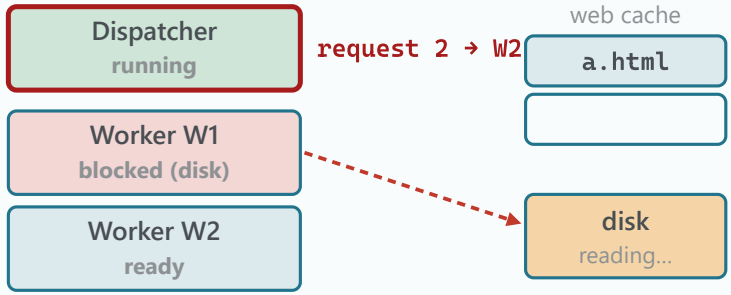
VARIABLES

`buf = a.html`

OUTPUT

Code Trace: A Multithreaded Web Server

```
1  /* dispatcher thread */
2  while (TRUE) {
3    get_next_request(&buf);
4    handoff_work(&buf);
5  }
6  /* worker thread */
7  while (TRUE) {
8    wait_for_work(&buf);
9    look_for_page_in_cache(&buf, &page);
10   if (page_not_in_cache(&page))
11     read_page_from_disk(&buf, &page);
12   return_page(&page);
13 }
```



W1 is busy, so request 2 goes to W2.

Step 7 / 11

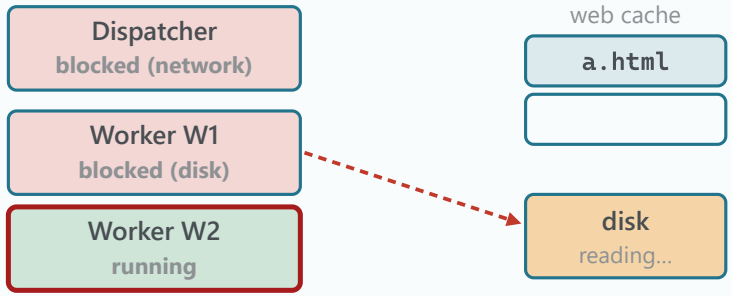
VARIABLES

`buf = a.html`

OUTPUT

Code Trace: A Multithreaded Web Server

```
1  /* dispatcher thread */
2  while (TRUE) {
3    get_next_request(&buf);
4    handoff_work(&buf);
5  }
6  /* worker thread */
7  while (TRUE) {
▶    wait_for_work(&buf);
▶    look_for_page_in_cache(&buf, &page);
▶    if (page_not_in_cache(&page))
11     read_page_from_disk(&buf, &page);
12   return_page(&page);
13 }
```



W2 **finds** a.html in the cache: no need to go to disk.

Step 8 / 11

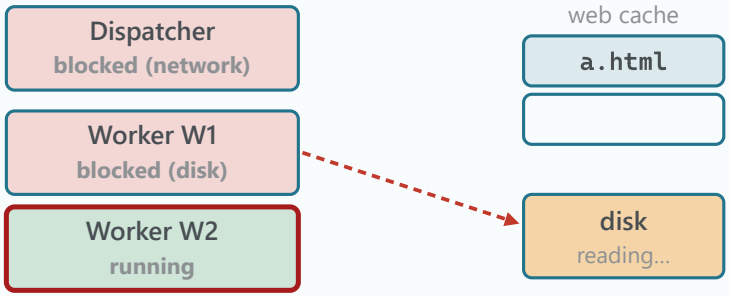
VARIABLES

`W2.buf = a.html` `W2.page = a.html`

OUTPUT

Code Trace: A Multithreaded Web Server

```
1  /* dispatcher thread */
2  while (TRUE) {
3    get_next_request(&buf);
4    handoff_work(&buf);
5  }
6  /* worker thread */
7  while (TRUE) {
8    wait_for_work(&buf);
9    look_for_page_in_cache(&buf, &page);
10   if (page_not_in_cache(&page))
11     read_page_from_disk(&buf, &page);
▶  12   return_page(&page);
13 }
```



W2 returns the page. **Request 2 was answered before request 1**.

Step 9 / 11

VARIABLES

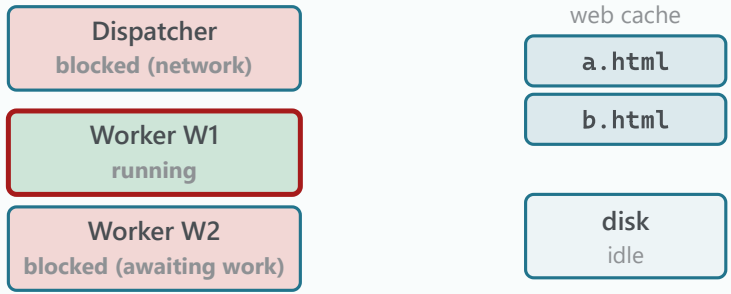
`W2.buf = a.html` `W2.page = a.html`

OUTPUT

```
a.html → client 2
```

Code Trace: A Multithreaded Web Server

```
1  /* dispatcher thread */
2  while (TRUE) {
3    get_next_request(&buf);
4    handoff_work(&buf);
5  }
6  /* worker thread */
7  while (TRUE) {
8    wait_for_work(&buf);
9    look_for_page_in_cache(&buf, &page);
10   if (page_not_in_cache(&page))
▶      read_page_from_disk(&buf, &page);
▶    return_page(&page);
13 }
```



The disk read is done: W1 wakes up, the page is also put in the cache and sent to client 1.

Step 10 / 11

VARIABLES

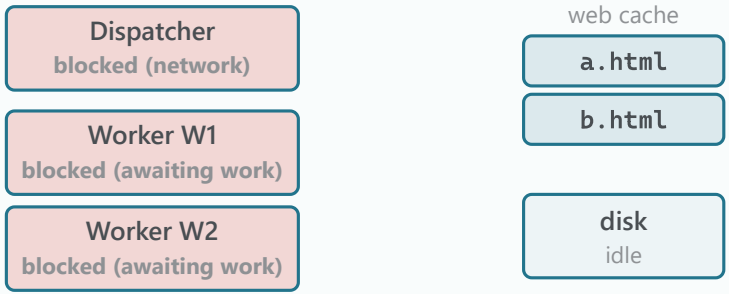
`W1.buf = b.html` `W1.page = b.html`

OUTPUT

```
a.html → client 2
b.html → client 1
```

Code Trace: A Multithreaded Web Server

```
1  /* dispatcher thread */
2  while (TRUE) {
3    get_next_request(&buf);
4    handoff_work(&buf);
5  }
6  /* worker thread */
7  while (TRUE) {
▶    wait_for_work(&buf);
9    look_for_page_in_cache(&buf, &page);
10   if (page_not_in_cache(&page))
11     read_page_from_disk(&buf, &page);
12   return_page(&page);
13 }
```



Both workers are waiting for work again. In a single-threaded server request 2 **would have waited** for the disk read; here each piece of code is still sequential and readable.

Step 11 / 11

VARIABLES

`buf = –`

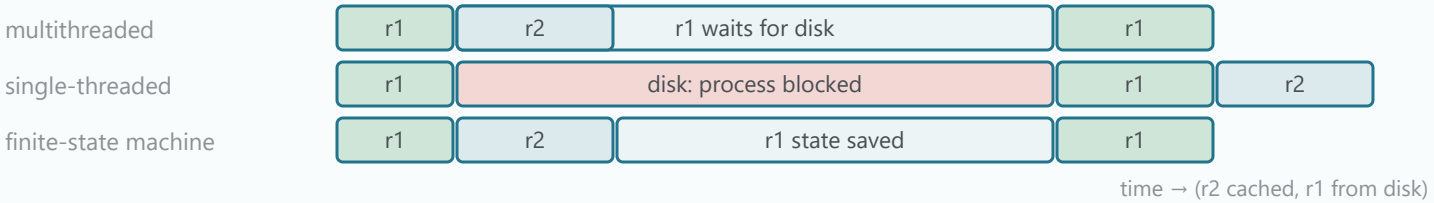
OUTPUT

```
a.html → client 2
b.html → client 1
```

Three Ways to Build a Server

Model	Parallelism / overlap	System calls	Programming
Multithreaded process	yes	blocking	sequential code, easy
Single-threaded process	no	blocking	easy but slow
Finite-state machine (event-driven)	yes (I/O overlaps)	nonblocking + notification / interrupt	hard: each request's state is saved by hand

Today: the event-driven model is common in systems such as nginx, Node.js and Redis, using `epoll` and `io_uring` on Linux and `kqueue` on BSD/macOS. Most servers combine both: **one event loop per core** + a thread pool for blocking work.



Source: the comparison in Tanenbaum & Bos, Modern Operating Systems, Figure 2-10.

Benefits of Multithreading

Responsiveness

The program keeps responding even if part of it blocks or runs a long operation; especially important for user interfaces.

Resource sharing

Threads share the process's memory and files automatically; easier than setting up shared memory or message passing.

Economy

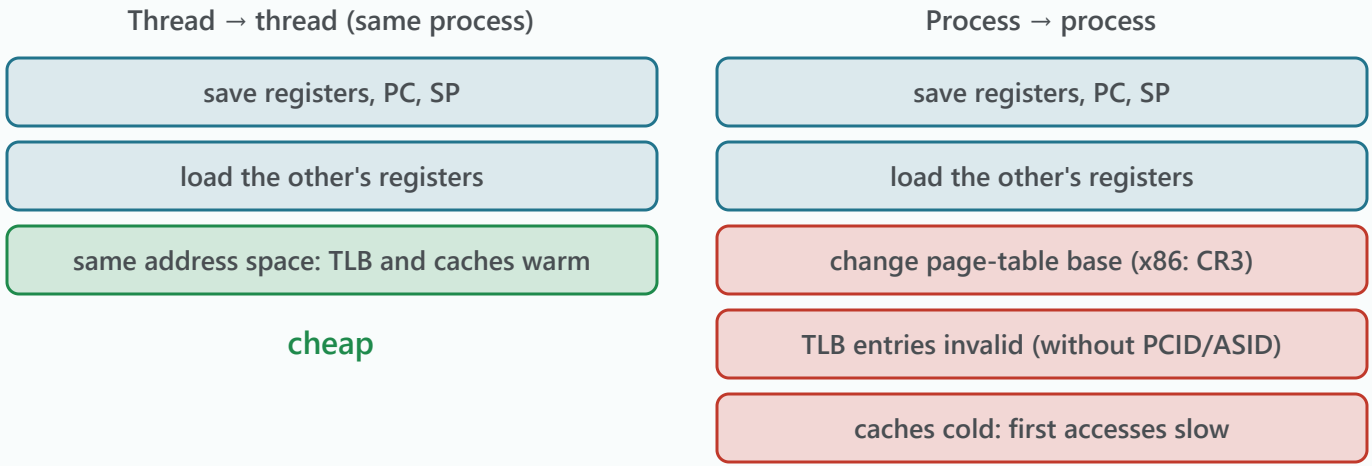
Creating and switching threads is cheaper than for processes: no address space is copied and the page tables do not change.

Scalability

A single multithreaded process can run on several cores **at once**; a single-threaded process uses only one core.

The oft-quoted figures “30× faster to create, 5× faster to switch” come from an old **Solaris** measurement. On Linux both processes and threads are created with `clone()`, so the gap is smaller; the real gain is not copying the memory map and keeping the TLB.

The Cost of a Context Switch



- With user-level threads the switch is a library function call that never enters the kernel (no trap): cheaper still.
- The direct cost is on the order of microseconds; the indirect cost (cold caches, TLB misses) is often larger.

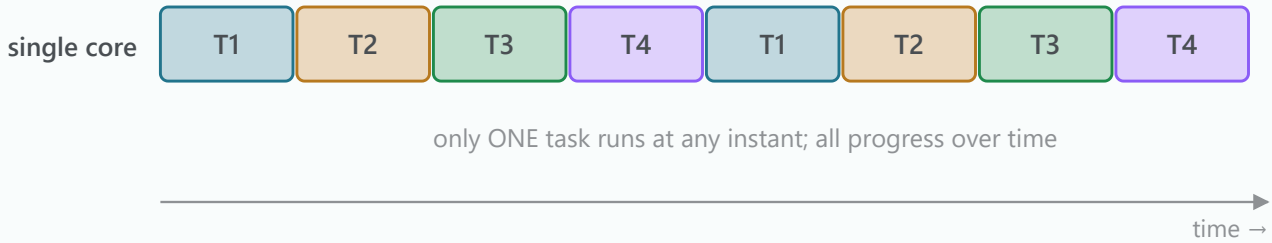
Concurrency and Parallelism

Concurrency

Several tasks make progress *over the same time interval*; possible even on one core by interleaving.

Parallelism

Several tasks execute *at the same instant*; needs at least two cores (or processing units).



Time sharing on one core: **concurrent but not parallel**.

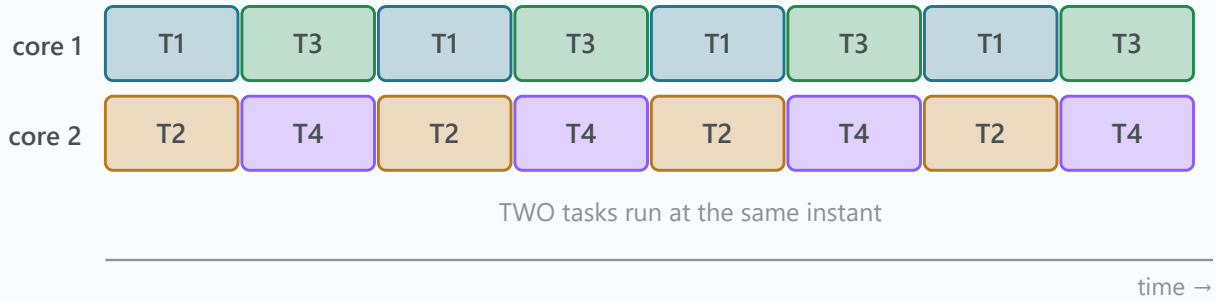
Concurrency and Parallelism

Concurrency

Several tasks make progress *over the same time interval*; possible even on one core by interleaving.

Parallelism

Several tasks execute *at the same instant*; needs at least two cores (or processing units).



On two cores T1/T3 and T2/T4 run **at the same time**: parallel execution.

Concurrency and Parallelism

Concurrency

Several tasks make progress *over the same time interval*; possible even on one core by interleaving.

Parallelism

Several tasks execute *at the same instant*; needs at least two cores (or processing units).



Parallelism requires concurrency; the converse is not true. The same program behaves differently on the two machines.

Challenges of Multicore Programming

Identifying tasks
Finding independent activities in the application that can run in parallel.

Balance
Tasks should be of similar size; dedicating a core to a tiny task gains nothing.

Data splitting
The data the tasks access must also be divided among the cores.

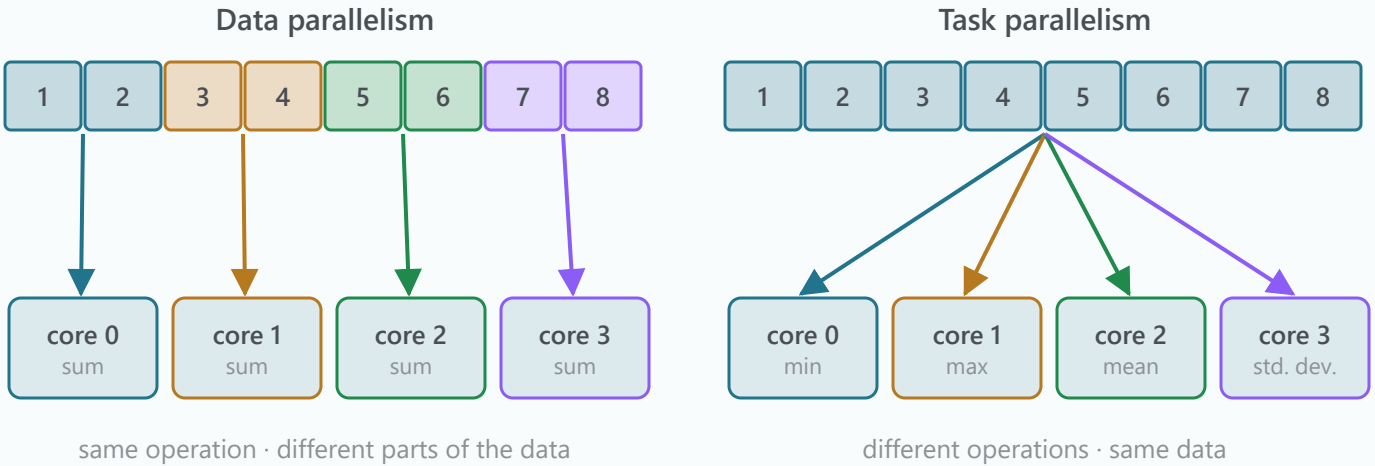
Data dependency
If one task waits for another's result, access must be synchronized.

Testing and debugging
The execution order can change on every run; bugs are hard to reproduce.

Memory effects
Cores writing to the same cache line slow each other down (*false sharing*).

Tools: ThreadSanitizer (`-fsanitize=thread`), Helgrind, Java Flight Recorder; but no tool catches every race. Reducing sharing at design time is the most effective approach.

Data and Task Parallelism

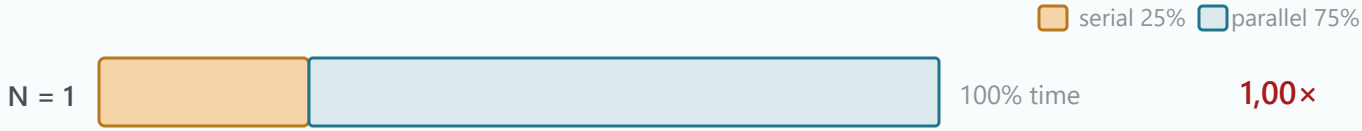


- **Data parallelism:** to sum an `N`-element array, each core sums `N/4` elements and the partial sums are combined at the end (GPUs are the extreme case of this model).
- **Task parallelism:** the work is split into independent tasks; the tasks may use the same or different data.
- Real applications usually combine the two.

Amdahl's Law

$$speedup \le 1 / (S + (1 - S) / N)$$

S: serial (non-parallelizable) fraction · *N*: number of cores · as $N \to \infty$ the limit is **1 / *S***



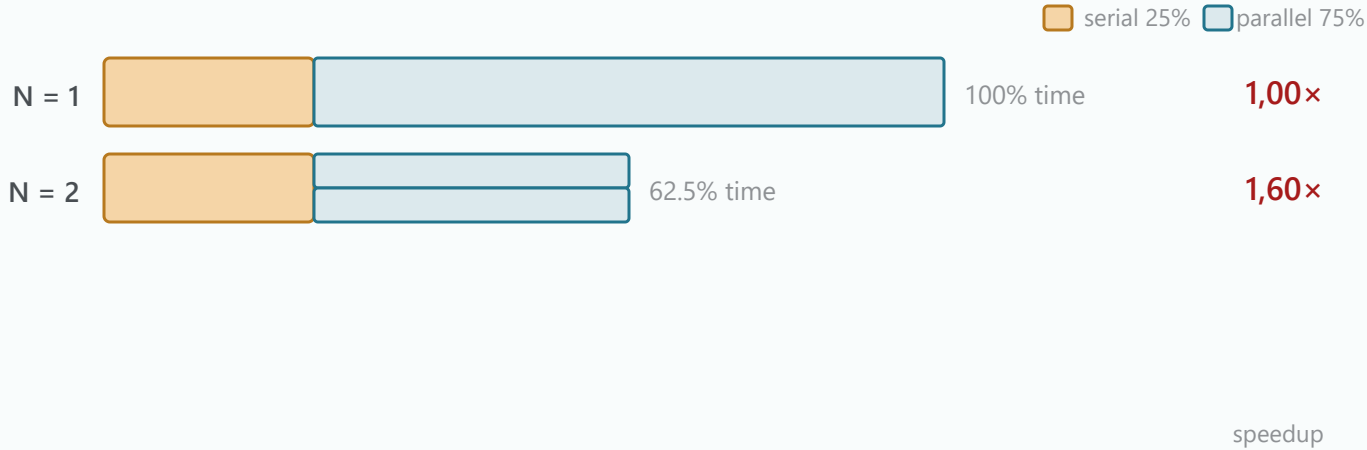
speedup

Example: **S = 0.25**. On one core the time is 100%, the speedup 1×.

Amdahl's Law

$$speedup \le 1 / (S + (1 - S) / N)$$

S: serial (non-parallelizable) fraction · *N*: number of cores · as $N \to \infty$ the limit is **1 / *S***

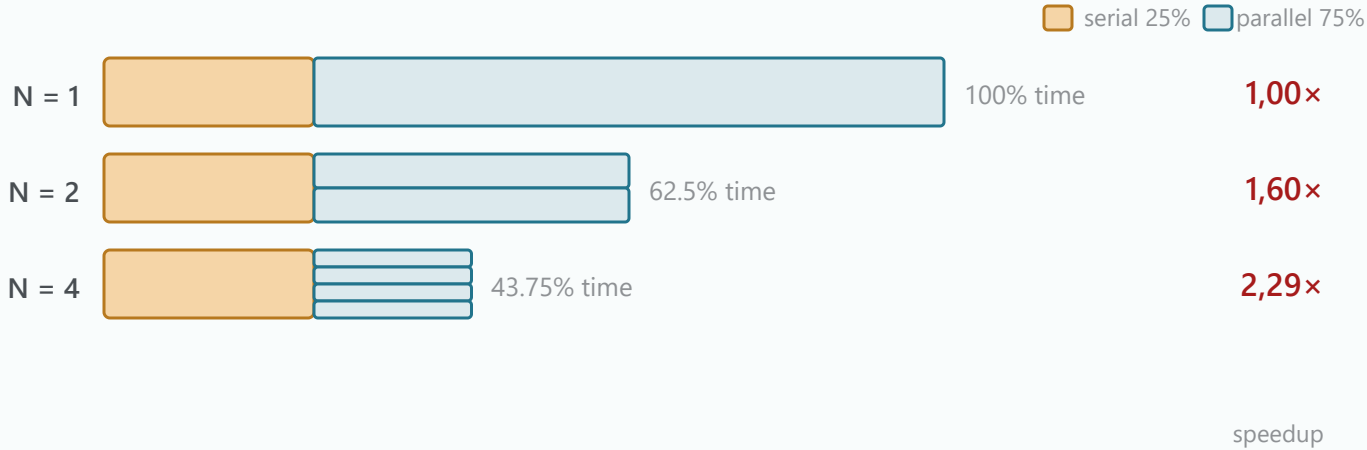


N = 2: 0.25 + 0.75/2 = 0.625 → speedup 1 / 0.625 = **1.60×** (not 2×).

Amdahl's Law

$$speedup \le 1 / (S + (1 - S) / N)$$

S: serial (non-parallelizable) fraction · *N*: number of cores · as $N \to \infty$ the limit is **1 / *S***

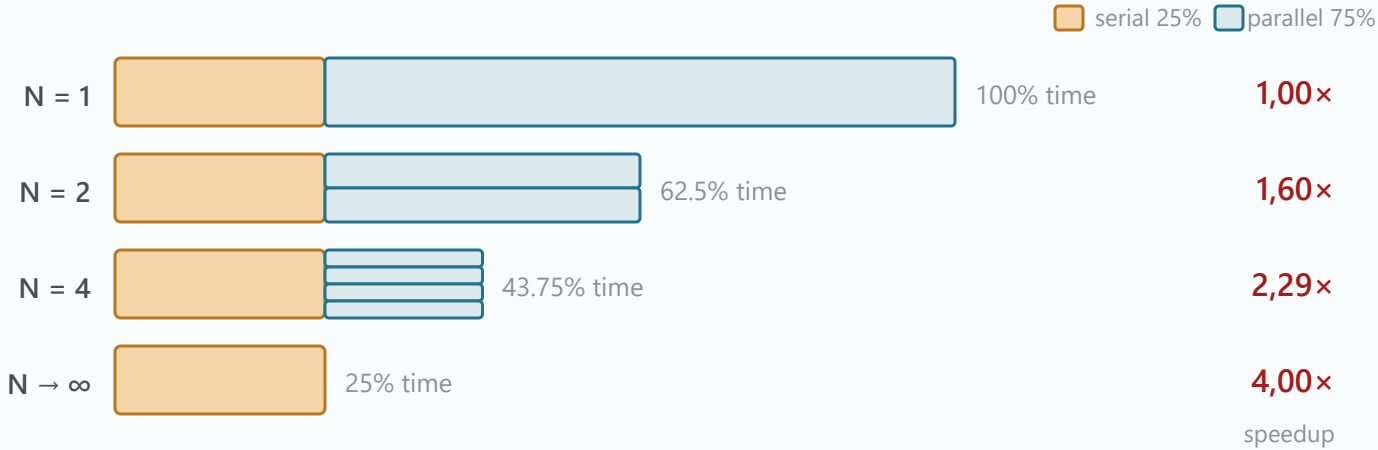


N = 4: 0.25 + 0.75/4 = 0.4375 → speedup **2.29×**.

Amdahl's Law

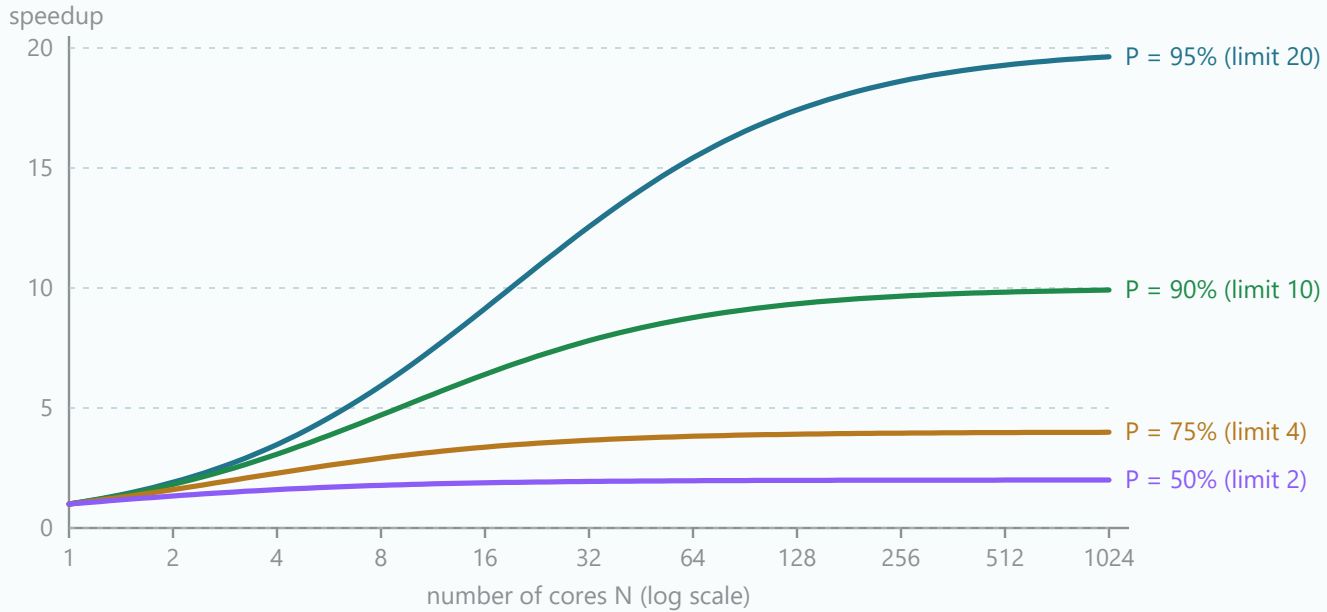
$$speedup \le 1 / (S + (1 - S) / N)$$

S: serial (non-parallelizable) fraction · *N*: number of cores · as $N \to \infty$ the limit is **1 / *S***



N → ∞: the parallel part shrinks toward zero and the serial 25% remains → at most **4×**.

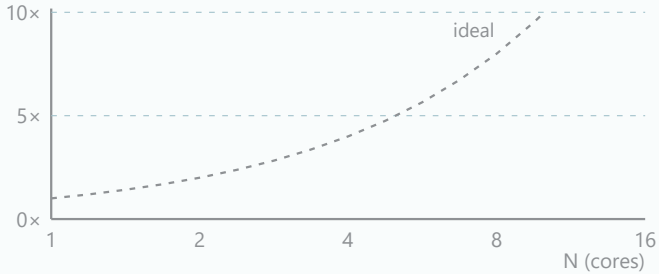
Amdahl Curves



- If **95%** of the program is parallel, the theoretical limit on speedup is 1 / 0.05 = **20×**; even on 1024 cores it is about 19.6×.
- In practice synchronization and communication overhead push the speedup lower still. **Gustafson's law:** if the problem size grows with the cores, the scaled speedup is S + (1 − S)·N.

Code Trace: Amdahl Speedup

```
▶ double S = 0.1;           /* serial fraction
  */
▶ int N = 1;                /* cores */
3 while (N <= 16) {
4   double sp = 1 / (S + (1 - S) / N);
5   printf("N=%2d  %.2fx\n", N, sp);
6   N = N * 2;
7 }
8 printf("limit %.0fx\n", 1 / S);
```



10% of the program is serial; 90% can be parallelized. On one core the time is 1 unit.

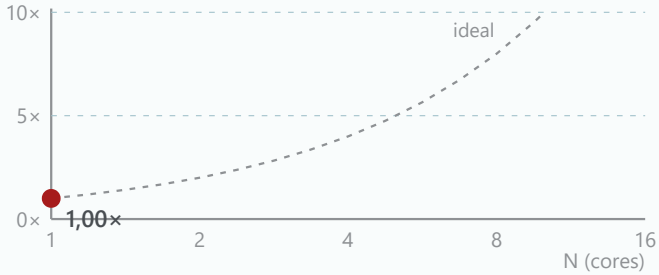
Step 1 / 9

VARIABLES

`S = 0.1` `N = 1`

Code Trace: Amdahl Speedup

```
1 double S = 0.1;            /* serial fraction
  */
2 int N = 1;                 /* cores */
▶ while (N <= 16) {
▶   double sp = 1 / (S + (1 - S) / N);
5   printf("N=%2d  %.2fx\n", N, sp);
6   N = N * 2;
7 }
8 printf("limit %.0fx\n", 1 / S);
```



N = 1: 0.1 + 0.9 / 1 = 1 → **sp = 1**.

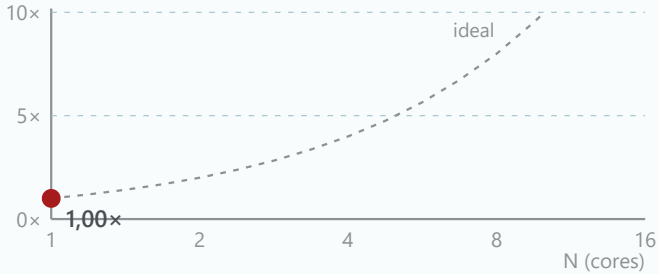
Step 2 / 9

VARIABLES

S = 0.1 | N = 1 | sp = 1

Code Trace: Amdahl Speedup

```
1 double S = 0.1;            /* serial fraction
  */
2 int N = 1;                 /* cores */
3 while (N <= 16) {
4   double sp = 1 / (S + (1 - S) / N);
▶   printf("N=%2d  %.2fx\n", N, sp);
▶   N = N * 2;
7 }
8 printf("limit %.0fx\n", 1 / S);
```



The result is printed and the core count doubles: N = 2.

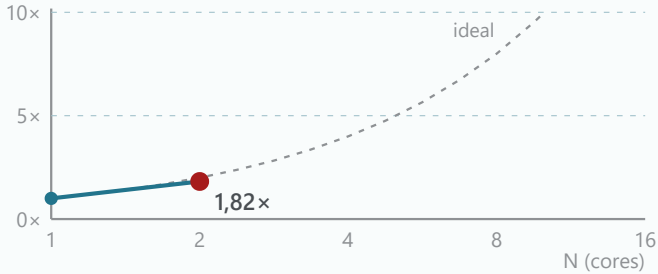
Step 3 / 9

VARIABLES

`S = 0.1` `N = 2` `sp = 1`

Code Trace: Amdahl Speedup

```
1 double S = 0.1;           /* serial fraction
  */
2 int N = 1;                /* cores */
▶ while (N <= 16) {
▶   double sp = 1 / (S + (1 - S) / N);
▶   printf("N=%2d  %.2fx\n", N, sp);
▶   N = N * 2;
7 }
8 printf("limit %.0fx\n", 1 / S);
```



N = 2: 0.1 + 0.9 / 2 = 0.55 → sp = **1.82×**.

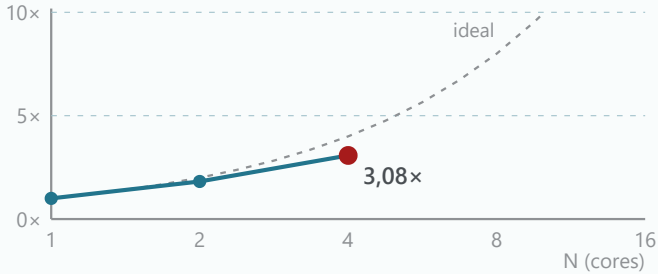
Step 4 / 9

VARIABLES

`S = 0.1` `N = 2` `sp = 1.818`

Code Trace: Amdahl Speedup

```
1 double S = 0.1;            /* serial fraction
  */
2 int N = 1;                 /* cores */
▶ while (N <= 16) {
▶   double sp = 1 / (S + (1 - S) / N);
▶   printf("N=%2d  %.2fx\n", N, sp);
▶   N = N * 2;
7 }
8 printf("limit %.0fx\n", 1 / S);
```



N = 4: 0.1 + 0.9 / 4 = 0.325 → sp = **3.08×**.

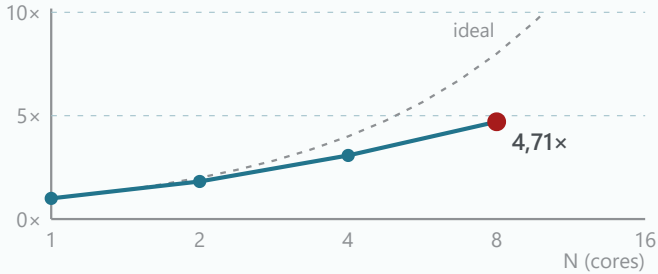
Step 5 / 9

VARIABLES

`S = 0.1` `N = 4` `sp = 3.077`

Code Trace: Amdahl Speedup

```
1 double S = 0.1;            /* serial fraction
  */
2 int N = 1;                 /* cores */
▶ while (N <= 16) {
▶   double sp = 1 / (S + (1 - S) / N);
▶   printf("N=%2d  %.2fx\n", N, sp);
▶   N = N * 2;
7 }
8 printf("limit %.0fx\n", 1 / S);
```



N = 8: 0.1 + 0.9 / 8 = 0.2125 → sp = **4.71×**. Cores doubled; speedup rose by 53%.

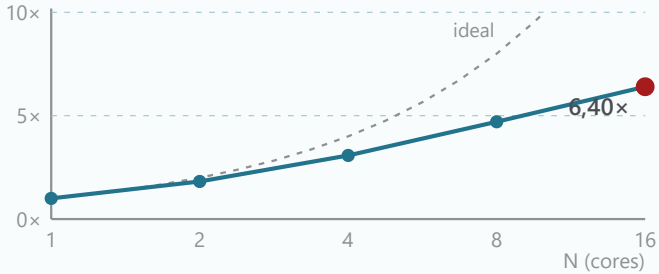
Step 6 / 9

VARIABLES

`S = 0.1` `N = 8` `sp = 4.706`

Code Trace: Amdahl Speedup

```
1 double S = 0.1;            /* serial fraction
  */
2 int N = 1;                 /* cores */
▶ while (N <= 16) {
▶   double sp = 1 / (S + (1 - S) / N);
▶   printf("N=%2d  %.2fx\n", N, sp);
▶   N = N * 2;
7 }
8 printf("limit %.0fx\n", 1 / S);
```



N = 16: 0.1 + 0.9 / 16 = 0.15625 → sp = **6.40×**. Cores doubled; speedup rose by 36%.

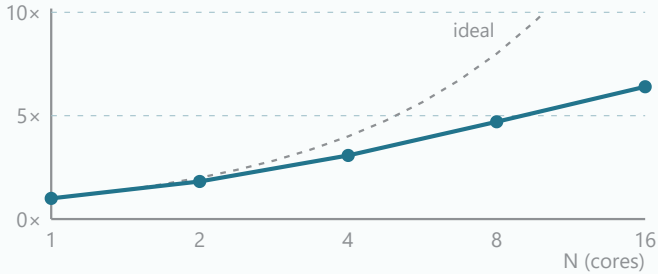
Step 7 / 9

VARIABLES

`S = 0.1` `N = 16` `sp = 6.4`

Code Trace: Amdahl Speedup

```
1 double S = 0.1;            /* serial fraction
  */
2 int N = 1;                 /* cores */
▶ while (N <= 16) {
4   double sp = 1 / (S + (1 - S) / N);
5   printf("N=%2d  %.2fx\n", N, sp);
6   N = N * 2;
7 }
8 printf("limit %.0fx\n", 1 / S);
```



N = 32 > 16: the loop ends.

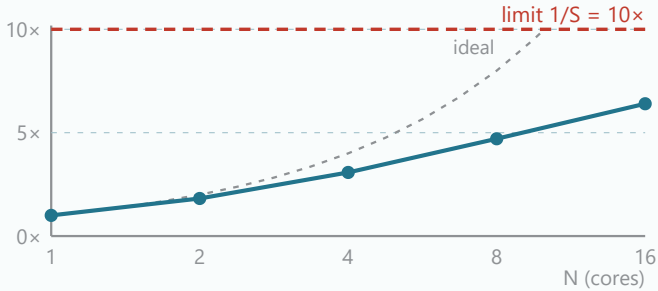
Step 8 / 9

VARIABLES

`S = 0.1` `N = 32`

Code Trace: Amdahl Speedup

```
1 double S = 0.1;            /* serial fraction
  */
2 int N = 1;                 /* cores */
3 while (N <= 16) {
4   double sp = 1 / (S + (1 - S) / N);
5   printf("N=%2d  %.2fx\n", N, sp);
6   N = N * 2;
7 }
▶ printf("limit %.0fx\n", 1 / S);
```



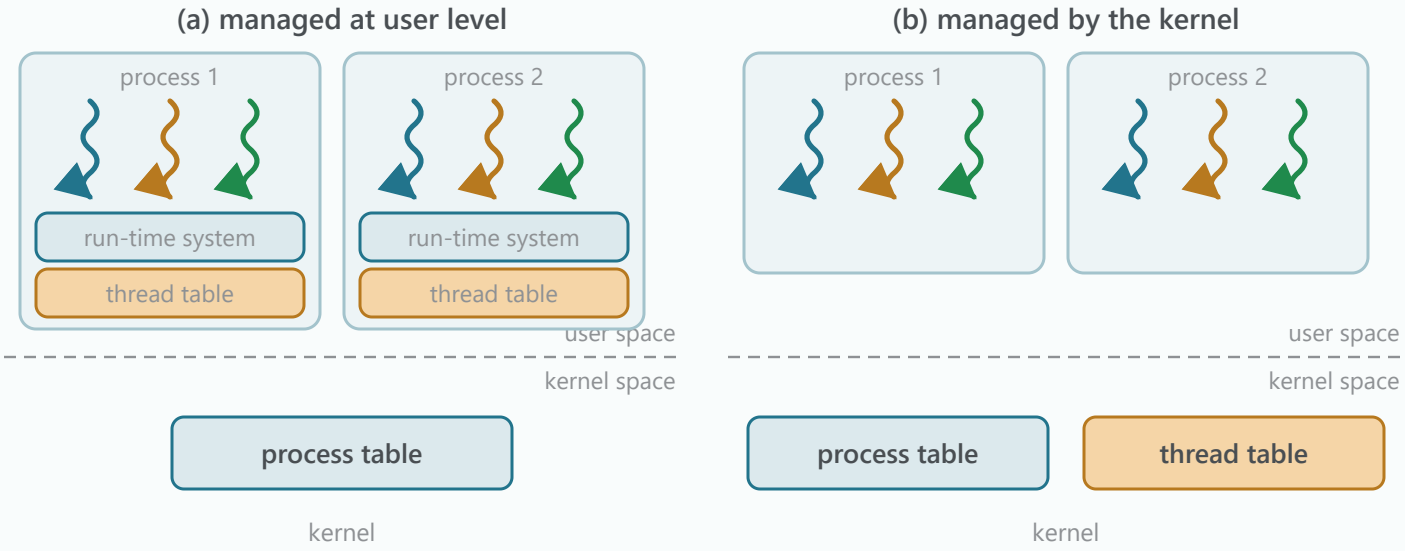
As N → ∞ the parallel part shrinks to zero and the serial 10% remains: the speedup can never exceed **10×**. Even 16 cores give only 6.4×.

Step 9 / 9

VARIABLES

`S = 0.1` `N = 32`

User and Kernel Threads



- **User level:** threads are managed in a library (the run-time system); the kernel thinks each process has one flow.
- **Kernel level:** the kernel knows every thread and schedules it itself. This is how Linux, Windows and macOS work.

User-Level Threads

Advantages

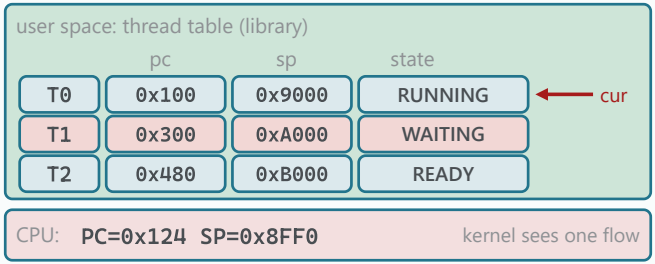
+ Can be implemented as a library even on a kernel without thread support.
+ Each process keeps its own **thread table**: PC, registers, SP, state.
+ Switching is very fast: **no trap**, no kernel entry, the memory map does not change, the TLB is not flushed.
+ Each process can use its own scheduling algorithm.

Disadvantages

– A blocking system call (e.g. `read`) stops **the whole process**, and so all its threads.
– The workarounds are clumsy: wrappers (jackets) that test the call with `select` first, nonblocking calls.
– A page fault also blocks the whole process.
– Clock interrupts do not reach the library: if a thread **never yields** (`yield`), the others never run.
– One process cannot use multiple cores.

Code Trace: `thread_yield` at User Level

```
1  enum { READY, RUNNING, WAITING };
2  struct tcb { int pc, sp, state; } T[3];
▶  int cur = 0;                 /* running */
4  void thread_yield(void) {   /* in library */
5    T[cur].pc = PC;   T[cur].sp = SP;
6    T[cur].state = READY;
7    do cur = (cur + 1) % 3;
8    while (T[cur].state != READY);
9    T[cur].state = RUNNING;
10   PC = T[cur].pc;   SP = T[cur].sp;
11 }
```



Three threads in one process. T0 is running (PC = 0x124 on the CPU); T1 is **waiting** for a lock in the library; T2 is ready. T0 voluntarily calls `thread_yield()`.

Step 1 / 10

VARIABLES

`cur = 0` `PC = 0x124` `SP = 0x8FF0`

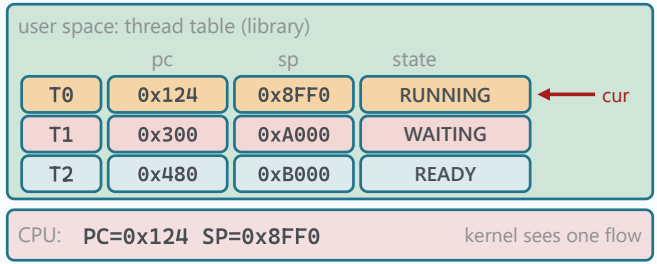
CALL STACK

`thread_yield()`

`T0`

Code Trace: `thread_yield` at User Level

```
1  enum { READY, RUNNING, WAITING };
2  struct tcb { int pc, sp, state; } T[3];
3  int cur = 0;                 /* running */
4  void thread_yield(void) {    /* in library */
▶    T[cur].pc = PC;   T[cur].sp = SP;
6    T[cur].state = READY;
7    do cur = (cur + 1) % 3;
8    while (T[cur].state != READY);
9    T[cur].state = RUNNING;
10   PC = T[cur].pc;   SP = T[cur].sp;
11 }
```



T0's PC and SP are saved in **the table**. This is an ordinary function call: no trap, no kernel entry.

Step 2 / 10

VARIABLES

`cur = 0` `PC = 0x124` `SP = 0x8FF0`

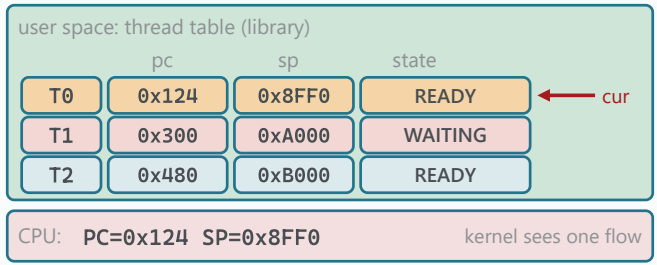
CALL STACK

`thread_yield()`

`T0`

Code Trace: `thread_yield` at User Level

```
1  enum { READY, RUNNING, WAITING };
2  struct tcb { int pc, sp, state; } T[3];
3  int cur = 0;                 /* running */
4  void thread_yield(void) {    /* in library */
5    T[cur].pc = PC;   T[cur].sp = SP;
▶    T[cur].state = READY;
7    do cur = (cur + 1) % 3;
8    while (T[cur].state != READY);
9    T[cur].state = RUNNING;
10   PC = T[cur].pc;   SP = T[cur].sp;
11 }
```



T0 is ready.

Step 3 / 10

VARIABLES

`cur = 0` `PC = 0x124` `SP = 0x8FF0`

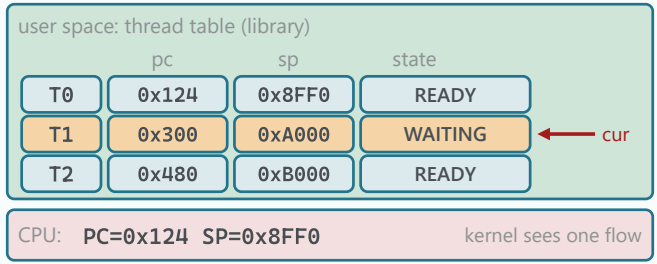
CALL STACK

`thread_yield()`

`T0`

Code Trace: `thread_yield` at User Level

```
1  enum { READY, RUNNING, WAITING };
2  struct tcb { int pc, sp, state; } T[3];
3  int cur = 0;                 /* running */
4  void thread_yield(void) {    /* in library */
5    T[cur].pc = PC;   T[cur].sp = SP;
6    T[cur].state = READY;
▶    do cur = (cur + 1) % 3;
8    while (T[cur].state != READY);
9    T[cur].state = RUNNING;
10   PC = T[cur].pc;   SP = T[cur].sp;
11 }
```



cur = 1.

Step 4 / 10

VARIABLES

`cur = 1` `PC = 0x124` `SP = 0x8FF0`

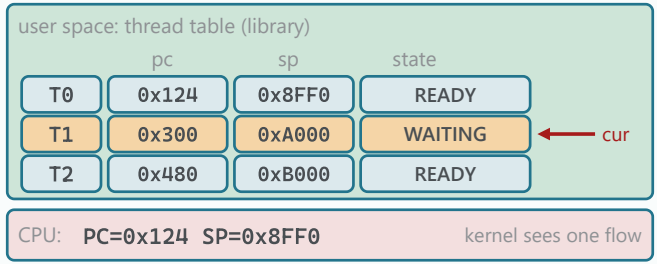
CALL STACK

`thread_yield()`

`T0`

Code Trace: `thread_yield` at User Level

```
1  enum { READY, RUNNING, WAITING };
2  struct tcb { int pc, sp, state; } T[3];
3  int cur = 0;                 /* running */
4  void thread_yield(void) {    /* in library */
5    T[cur].pc = PC;   T[cur].sp = SP;
6    T[cur].state = READY;
7    do cur = (cur + 1) % 3;
▶    while (T[cur].state != READY);
9    T[cur].state = RUNNING;
10   PC = T[cur].pc;   SP = T[cur].sp;
11 }
```



T1 is WAITING, so it is skipped: the loop continues.

Step 5 / 10

VARIABLES

`cur = 1` `PC = 0x124` `SP = 0x8FF0`

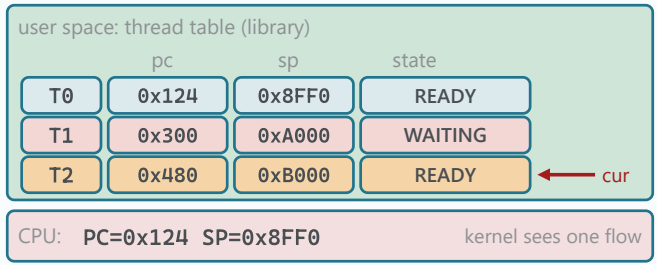
CALL STACK

`thread_yield()`

`T0`

Code Trace: `thread_yield` at User Level

```
1  enum { READY, RUNNING, WAITING };
2  struct tcb { int pc, sp, state; } T[3];
3  int cur = 0;                 /* running */
4  void thread_yield(void) {    /* in library */
5    T[cur].pc = PC;   T[cur].sp = SP;
6    T[cur].state = READY;
▶    do cur = (cur + 1) % 3;
8    while (T[cur].state != READY);
9    T[cur].state = RUNNING;
10   PC = T[cur].pc;   SP = T[cur].sp;
11 }
```



cur = 2.

Step 6 / 10

VARIABLES

`cur = 2` `PC = 0x124` `SP = 0x8FF0`

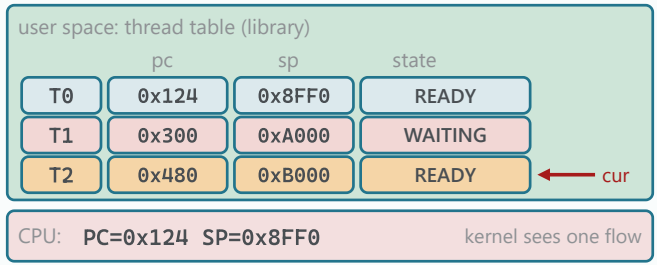
CALL STACK

`thread_yield()`

`T0`

Code Trace: `thread_yield` at User Level

```
1  enum { READY, RUNNING, WAITING };
2  struct tcb { int pc, sp, state; } T[3];
3  int cur = 0;                 /* running */
4  void thread_yield(void) {    /* in library */
5    T[cur].pc = PC;   T[cur].sp = SP;
6    T[cur].state = READY;
7    do cur = (cur + 1) % 3;
▶    while (T[cur].state != READY);
9    T[cur].state = RUNNING;
10   PC = T[cur].pc;   SP = T[cur].sp;
11 }
```



T2 is READY: the search ends.

Step 7 / 10

VARIABLES

`cur = 2` `PC = 0x124` `SP = 0x8FF0`

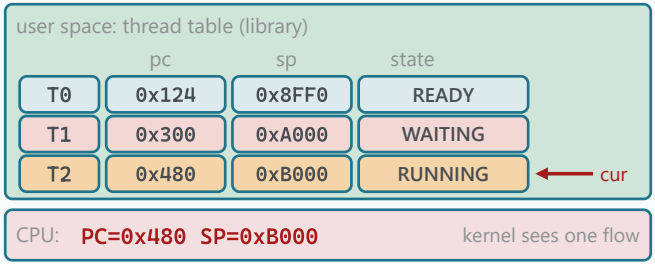
CALL STACK

`thread_yield()`

`T0`

Code Trace: `thread_yield` at User Level

```
1  enum { READY, RUNNING, WAITING };
2  struct tcb { int pc, sp, state; } T[3];
3  int cur = 0;                 /* running */
4  void thread_yield(void) {  /* in library */
5    T[cur].pc = PC;  T[cur].sp = SP;
6    T[cur].state = READY;
7    do cur = (cur + 1) % 3;
8    while (T[cur].state != READY);
▶    T[cur].state = RUNNING;
▶    PC = T[cur].pc;  SP = T[cur].sp;
11 }
```



T2 is marked running and its registers are **loaded**: PC = 0x480. The return instruction now returns on T2's stack; the switch took a few instructions.

Step 8 / 10

VARIABLES

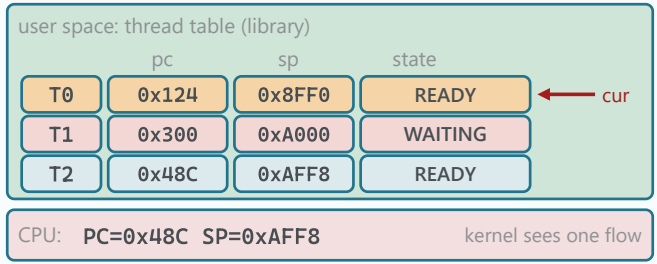
`cur = 2` `PC = 0x480` `SP = 0xB000`

CALL STACK

T2

Code Trace: `thread_yield` at User Level

```
1  enum { READY, RUNNING, WAITING };
2  struct tcb { int pc, sp, state; } T[3];
3  int cur = 0;                 /* running */
4  void thread_yield(void) {    /* in library */
▶    T[cur].pc = PC;  T[cur].sp = SP;
▶    T[cur].state = READY;
▶    do cur = (cur + 1) % 3;
▶    while (T[cur].state != READY);
9    T[cur].state = RUNNING;
10   PC = T[cur].pc;  SP = T[cur].sp;
11 }
```



After running a while (PC = 0x48C), T2 also calls `thread_yield()` : its state is saved, cur = 0 and T0 is ready.

Step 9 / 10

VARIABLES

`cur = 0` `PC = 0x48C` `SP = 0xAFF8`

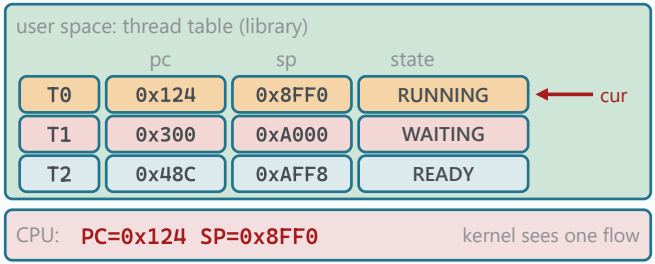
CALL STACK

`thread_yield()`

T2

Code Trace: `thread_yield` at User Level

```
1  enum { READY, RUNNING, WAITING };
2  struct tcb { int pc, sp, state; } T[3];
3  int cur = 0;                 /* running */
4  void thread_yield(void) {    /* in library */
5    T[cur].pc = PC;   T[cur].sp = SP;
6    T[cur].state = READY;
7    do cur = (cur + 1) % 3;
8    while (T[cur].state != READY);
▶    T[cur].state = RUNNING;
▶    PC = T[cur].pc;   SP = T[cur].sp;
11 }
```



T0 resumes where it left off, at PC = 0x124. Had T2 never yielded, T0 would have waited forever: **clock interrupts do not reach** the library.

Step 10 / 10

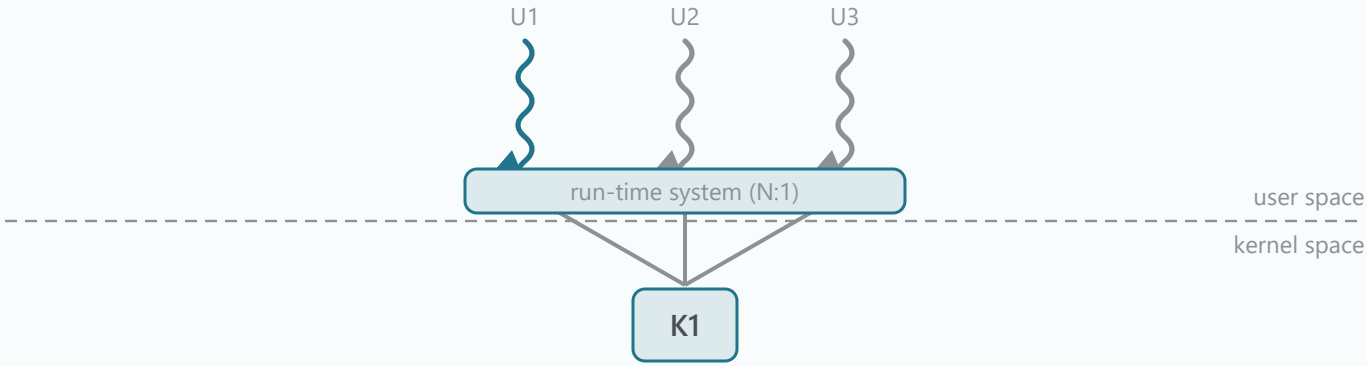
VARIABLES

`cur = 0` `PC = 0x124` `SP = 0x8FF0`

CALL STACK

`T0`

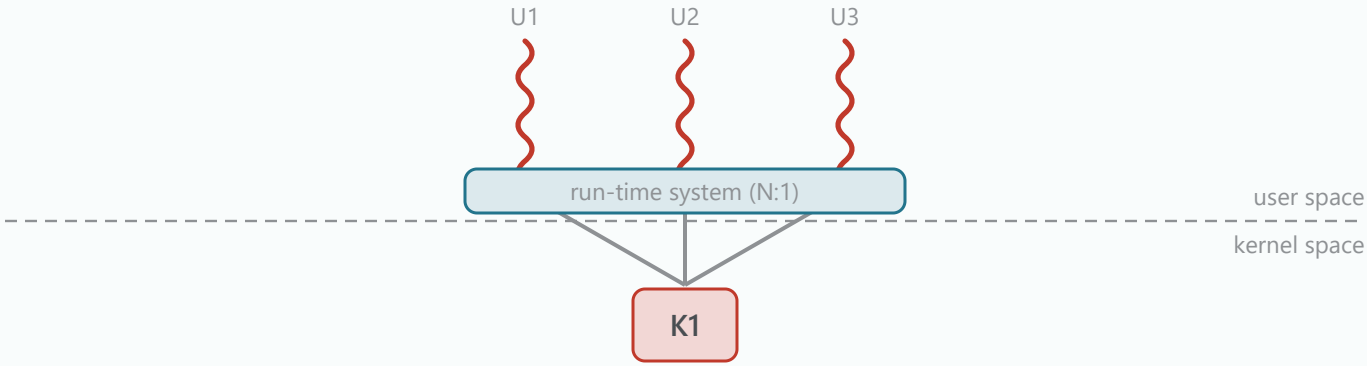
A Blocking Call: N:1 and 1:1



U1 running

Many-to-one (N:1): three user threads on a single kernel thread (K1); the library does the scheduling.

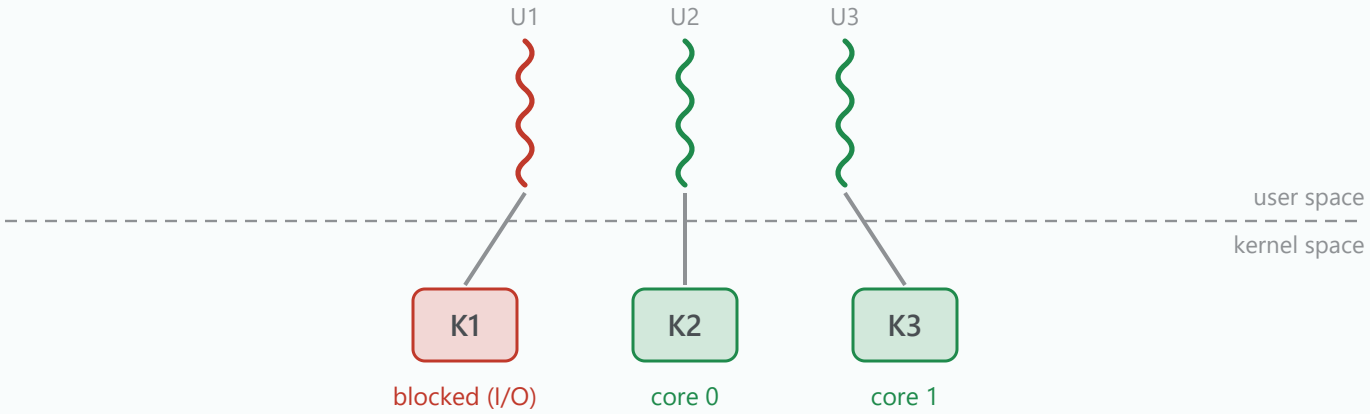
A Blocking Call: N:1 and 1:1



U1 called read() → K1 blocked → U2 and U3 wait too

U1 makes a blocking `read()` call: the kernel sees only K1 and blocks it → **the whole process stops**.

A Blocking Call: N:1 and 1:1



One-to-one (1:1): U1 blocking affects only K1; U2 and U3 run **in parallel** on two cores.

Kernel-Level Threads

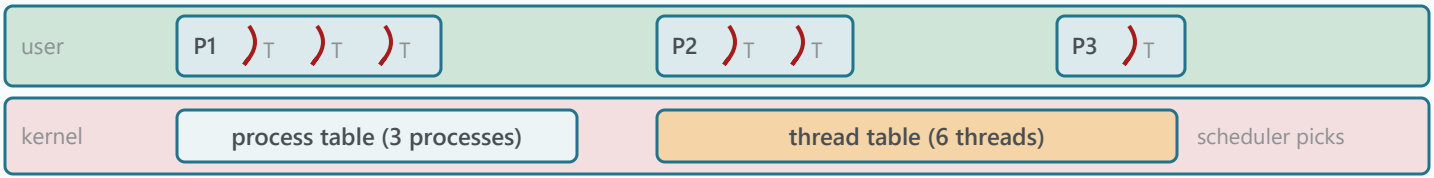
- The kernel keeps a **thread table** with the same information as at user level (registers, state…).
- Creating and terminating threads are **system calls**: more expensive than at user level.
- If a thread blocks, the kernel picks another; it **need not be from the same process**.
- A page fault makes only the faulting thread wait.
- A process's threads run **truly in parallel** on different cores.
- Each thread takes kernel memory (TCB, kernel stack): their number is not unlimited.

Today

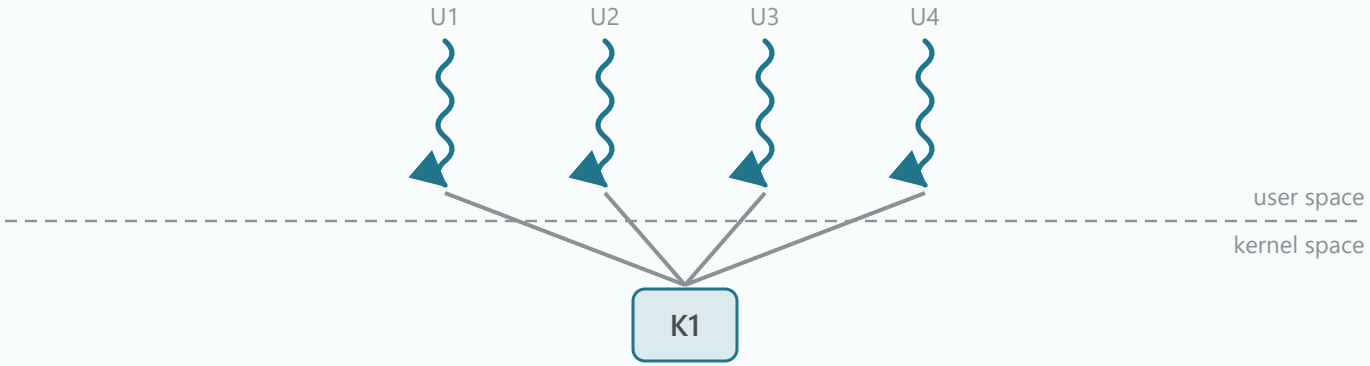
Linux (NPTL), Windows and macOS use the **1:1** model. With creation costs down, tens of thousands of kernel threads are no problem.

Limits on Linux: `/proc/sys/kernel/threads-max`, `ulimit -u`, the cgroup `pids.max`.

To observe them: `ps -eLf`, `top -H`, `/proc/<pid>/task/`.

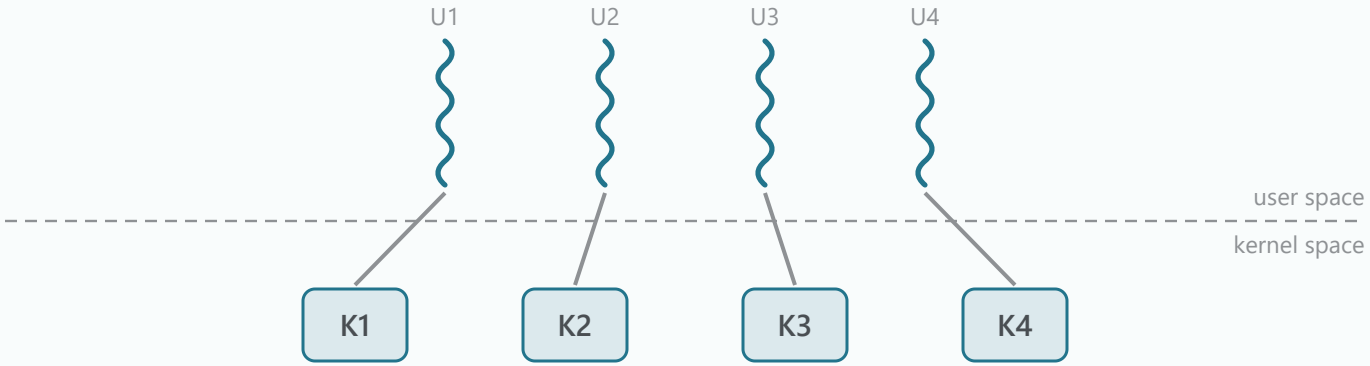


Mapping Models



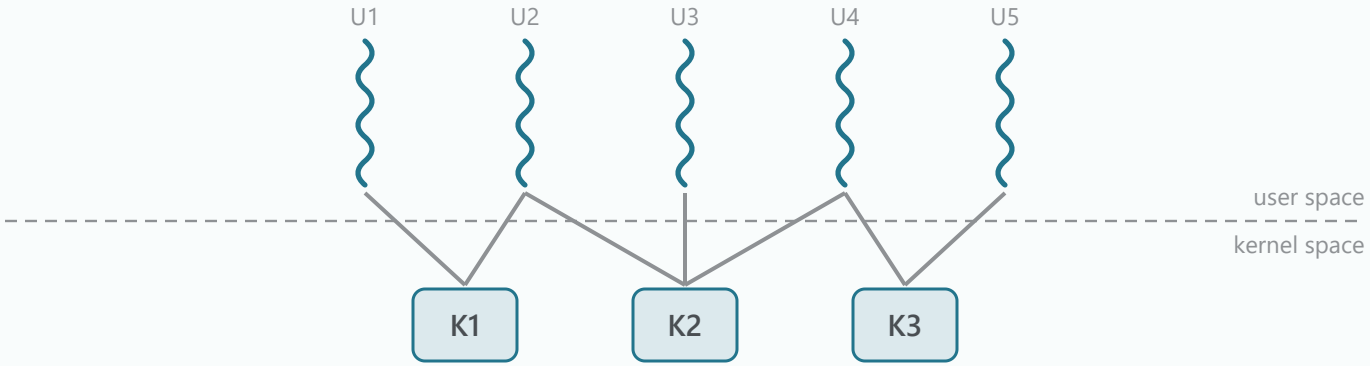
Many-to-one (N:1): fast switching, but one blocking call stops the whole process and there is no parallelism. Example: Solaris *green threads*, GNU Portable Threads (historical).

Mapping Models



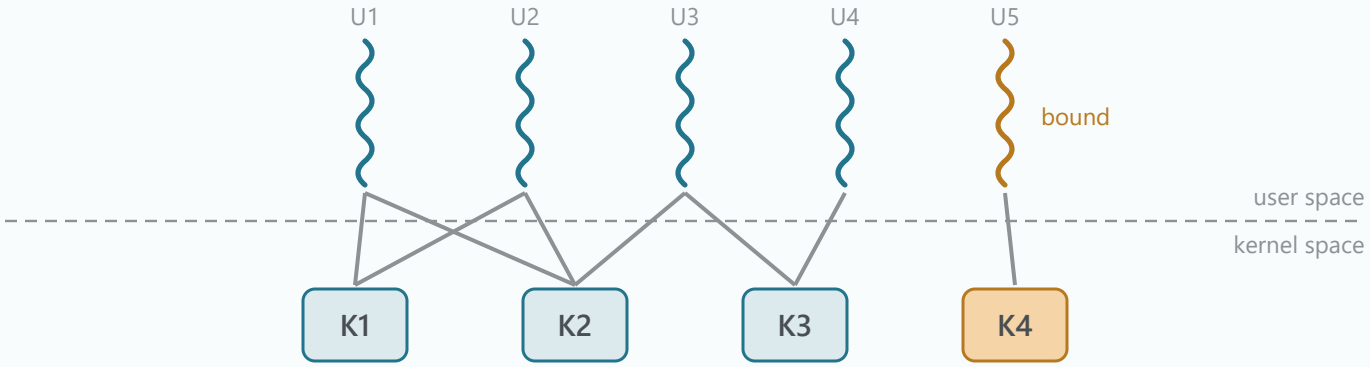
One-to-one (1:1): true parallelism and independent blocking; every thread costs kernel resources. Linux, Windows, macOS.

Mapping Models



Many-to-many (M:N): many user threads multiplexed onto fewer kernel threads; hard to implement.

Mapping Models



Two-level / hybrid: M:N, plus a user thread can be **bound** to a single kernel thread.

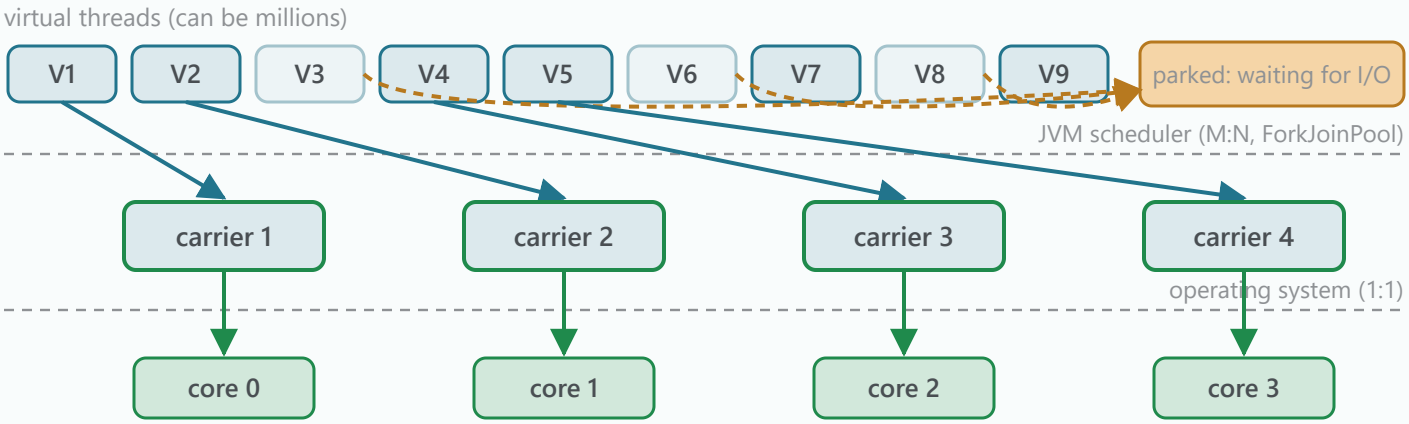
Comparing the Models

Criterion	N:1	1:1	M:N
Switch cost	very low	medium (kernel)	low
Blocking call	whole process stops	only that thread	only that thread*
Multiple cores	not usable	used	used
Number of threads	very many	limited by kernel memory	very many
Implementation	easy (library)	easy	hard (two schedulers)

* If the run-time notices the blocking call and moves to another kernel thread.

M:N was **abandoned** at the OS level: Solaris 9 and FreeBSD 7 moved to 1:1. But M:N is back in **language run-times**: Go goroutines, Erlang processes, Java 21 virtual threads.

Modern M:N: Virtual Threads



V3, V6, V8 are parked, not blocked: their carriers run other virtual threads

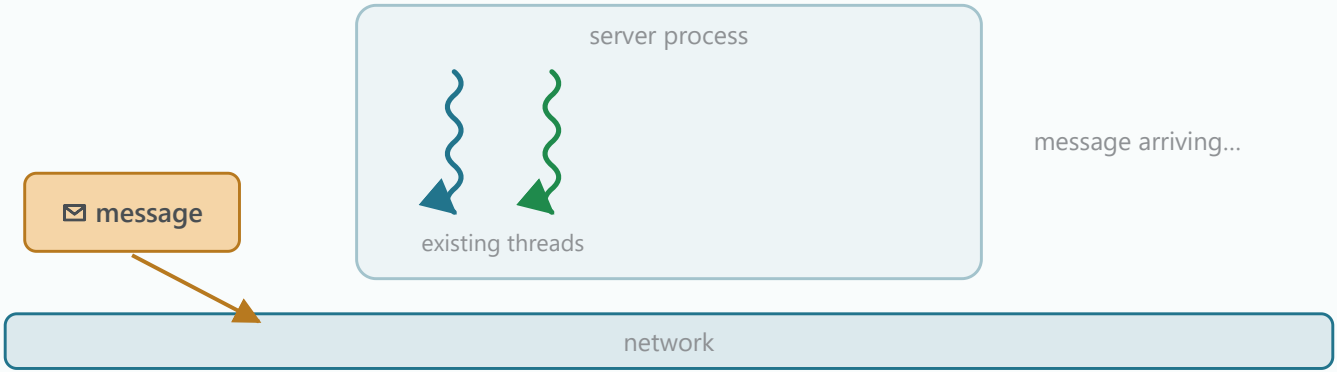
Java 21+ (Loom)

`Thread.ofVirtual().start(r)`; on a blocking I/O call the JVM unmounts the virtual thread from its carrier.

Go

G (goroutine) · M (OS thread) · P (logical processor, `GOMAXPROCS`); an idle P steals work.

Pop-Up Threads



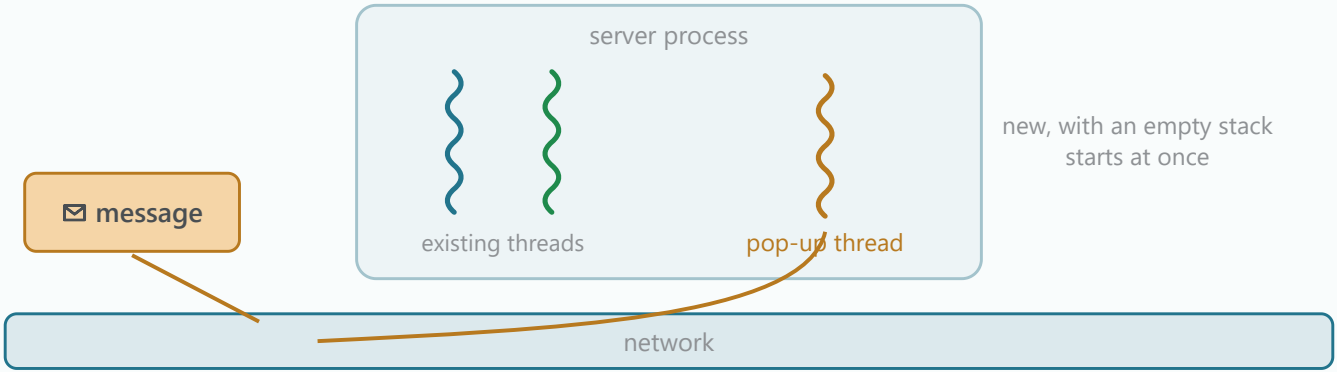
(a) Before the message arrives: no thread in the process is waiting for it.

The alternative: a thread blocked in `receive`, whose state (registers, stack) must be restored for every message.

A pop-up thread is new, with no history to restore: **low latency**. It can be used inside the kernel too (interrupt handling).

```
/* on message arrival */
pthread_t t;
pthread_create(&t, NULL, handle, msg);
pthread_detach(t); /* no join */
```

Pop-Up Threads



(b) When the message arrives, the system creates **a new thread** to handle it.

Step 2 / 2

The alternative: a thread blocked in `receive`, whose state (registers, stack) must be restored for every message.

A pop-up thread is new, with no history to restore: **low latency**. It can be used inside the kernel too (interrupt handling).

```
/* on message arrival */
pthread_t t;
pthread_create(&t, NULL, handle, msg);
pthread_detach(t); /* no join */
```

Thread Libraries

POSIX Pthreads

Linux, macOS, BSD; a C API. Creation, joining, mutexes, condition variables.

Windows API

`CreateThread`, `WaitForSingleObject`; kernel level, 1:1.

Java

`Thread`, `Runnable`, `ExecutorService`; the JVM uses OS threads (and, since Java 21, virtual threads).

C and C++ standards

C11 `<threads.h>`; C++11 `std::thread`, C++20 `std::jthread` (replacing the old Boost.Thread).

Python

`threading`; no parallelism for CPU work because of the GIL. A free-threaded (no-GIL) build: experimental in 3.13, supported in 3.14.

Go

Goroutines with `go f()`; an M:N run-time, communication over channels.

Asynchronous threading: the parent starts a child and carries on with its own work (e.g. a server). **Synchronous**: the parent waits for its children to finish (`join`) and combines their results: fork-join.

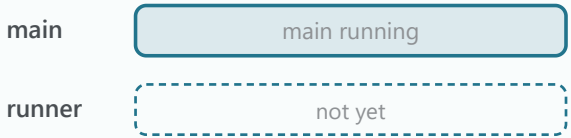
POSIX Pthreads

- The IEEE 1003.1c (POSIX.1c) standard: an **API** for thread creation and synchronization.
- A **specification**, not an implementation: it says how the library behaves; it can be implemented at user or kernel level.
- On Linux glibc NPTL (1:1); on macOS and the BSDs it also runs on kernel threads.
- Compile with: `gcc prog.c -pthread`.

Call	Description
`pthread_create`	Creates a new thread that runs the given function.
`pthread_exit`	Terminates the calling thread.
`pthread_join`	Waits for a specific thread to finish.
`pthread_detach`	Its resources are released automatically when it ends; it is not joined.
`pthread_self`	Returns the caller's own ID.
`sched_yield`	Gives up the CPU. (`pthread_yield` is nonstandard and was deprecated in glibc 2.34.)
`pthread_attr_init` / `_destroy`	Initializes / destroys an attribute object (stack size, detached state…).

Pthreads: Create and Join

```c
int sum;                /* shared */
void *runner(void *param) {
  int upper = atoi(param);
  sum = 0;
  for (int i = 1; i <= upper; i++)
    sum += i;
  pthread_exit(0);
}
int main(void) {
  pthread_t tid;
  pthread_create(&tid, NULL, runner, "5");
  pthread_join(tid, NULL);
  printf("sum = %d\n", sum);
  return 0;
}
```



The process starts with one thread (main); the global `sum` is 0.

Step 1 / 7

VARIABLES

`sum = 0`

OUTPUT

Adapted from Silberschatz Figure 4.11; the `#include <pthread.h>`, `<stdio.h>`, `<stdlib.h>` lines are not shown.

Pthreads: Create and Join

```
1  int sum;                /* shared */
2  void *runner(void *param) {
3    int upper = atoi(param);
4    sum = 0;
5    for (int i = 1; i <= upper; i++)
6      sum += i;
7    pthread_exit(0);
8  }
9  int main(void) {
10   pthread_t tid;
▶    pthread_create(&tid, NULL, runner, "5");
12   pthread_join(tid, NULL);
13   printf("sum = %d\n", sum);
14   return 0;
15 }
```

main	pthread_create
runner	ready (eligible to run)

`pthread_create` creates the new thread; `runner("5")` will run.

Step 2 / 7

VARIABLES

`sum = 0` `tid = T1`

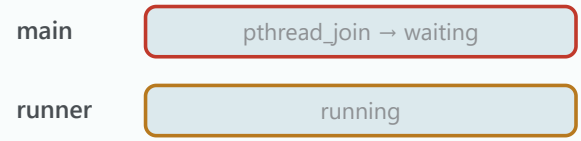
OUTPUT

Adapted from Silberschatz Figure 4.11; the `#include <pthread.h>`, `<stdio.h>`, `<stdlib.h>` lines are not shown.

Pthreads: Create and Join

```
1  int sum;                /* shared */
2  void *runner(void *param) {
3    int upper = atoi(param);
4    sum = 0;
5    for (int i = 1; i <= upper; i++)
6      sum += i;
7    pthread_exit(0);
8  }
9  int main(void) {
10   pthread_t tid;
11   pthread_create(&tid, NULL, runner, "5");
▶    pthread_join(tid, NULL);
13   printf("sum = %d\n", sum);
14   return 0;
15 }
```

Adapted from Silberschatz Figure 4.11; the `#include <pthread.h>`, `<stdio.h>`, `<stdlib.h>` lines are not shown.



main blocks in `join`. The scheduler decides when runner starts.

Step 3 / 7

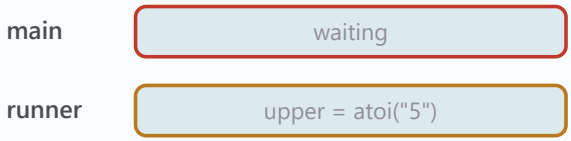
VARIABLES

`sum = 0` `tid = T1`

OUTPUT

Pthreads: Create and Join

```
1  int sum;                /* shared */
2  void *runner(void *param) {
▶    int upper = atoi(param);
4    sum = 0;
5    for (int i = 1; i <= upper; i++)
6      sum += i;
7    pthread_exit(0);
8  }
9  int main(void) {
10   pthread_t tid;
11   pthread_create(&tid, NULL, runner, "5");
12   pthread_join(tid, NULL);
13   printf("sum = %d\n", sum);
14   return 0;
15 }
```



runner creates the local variable `upper` on its own stack.

Step 4 / 7

VARIABLES

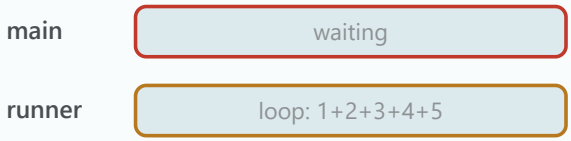
`sum = 0` `tid = T1` `upper = 5`

OUTPUT

Adapted from Silberschatz Figure 4.11; the `#include <pthread.h>`, `<stdio.h>`, `<stdlib.h>` lines are not shown.

Pthreads: Create and Join

```
1  int sum;                /* shared */
2  void *runner(void *param) {
3    int upper = atoi(param);
4    sum = 0;
▶    for (int i = 1; i <= upper; i++)
▶      sum += i;
7    pthread_exit(0);
8  }
9  int main(void) {
10   pthread_t tid;
11   pthread_create(&tid, NULL, runner, "5");
12   pthread_join(tid, NULL);
13   printf("sum = %d\n", sum);
14   return 0;
15 }
```



The loop updates the shared `sum` : 15.

Step 5 / 7

VARIABLES

`sum = 15` `tid = T1` `upper = 5`

OUTPUT

Adapted from Silberschatz Figure 4.11; the `#include <pthread.h>` , `<stdio.h>` , `<stdlib.h>` lines are not shown.

Pthreads: Create and Join

```
1  int sum;                /* shared */
2  void *runner(void *param) {
3    int upper = atoi(param);
4    sum = 0;
5    for (int i = 1; i <= upper; i++)
6      sum += i;
▶    pthread_exit(0);
8  }
9  int main(void) {
10   pthread_t tid;
11   pthread_create(&tid, NULL, runner, "5");
12   pthread_join(tid, NULL);
13   printf("sum = %d\n", sum);
14   return 0;
15 }
```

main	waking up
runner	pthread_exit(0) → done

runner terminates; `join` returns and main wakes up.

Step 6 / 7

VARIABLES

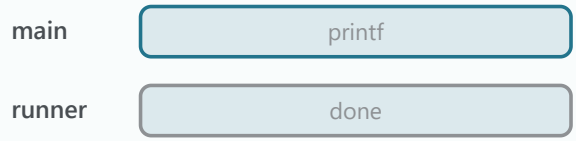
`sum = 15` `tid = T1`

OUTPUT

Adapted from Silberschatz Figure 4.11; the `#include <pthread.h>`, `<stdio.h>`, `<stdlib.h>` lines are not shown.

Pthreads: Create and Join

```
1  int sum;                /* shared */
2  void *runner(void *param) {
3    int upper = atoi(param);
4    sum = 0;
5    for (int i = 1; i <= upper; i++)
6      sum += i;
7    pthread_exit(0);
8  }
9  int main(void) {
10   pthread_t tid;
11   pthread_create(&tid, NULL, runner, "5");
12   pthread_join(tid, NULL);
▶    printf("sum = %d\n", sum);
14   return 0;
15 }
```



Thanks to join the output is the same on every run: **sum = 15**.

Step 7 / 7

VARIABLES

`sum = 15` `tid = T1`

OUTPUT

```
sum = 15
```

Adapted from Silberschatz Figure 4.11; the `#include <pthread.h>`, `<stdio.h>`, `<stdlib.h>` lines are not shown.

Multiple Threads

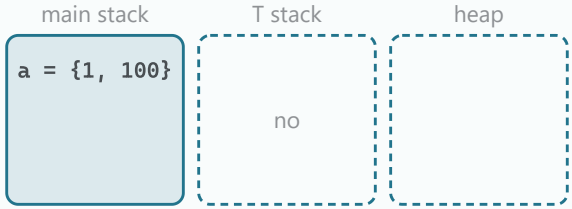
```c
#define N 4
long partial[N];          /* one cell each */

void *add_range(void *arg) {
  long k = (long)arg;     /* 0 .. N-1 */
  for (long i = k*250 + 1; i <= (k+1)*250; i++)
    partial[k] += i;
  return NULL;
}
int main(void) {
  pthread_t t[N];
  for (long k = 0; k < N; k++)
    pthread_create(&t[k], NULL, add_range, (void *)k);
  long total = 0;
  for (int k = 0; k < N; k++) {
    pthread_join(t[k], NULL);
    total += partial[k];
  }
  printf("%ld\n", total);   /* 500500 */
}
```

- The sum of 1..1000 is split among four threads: each adds up 250 numbers (**data parallelism**).
- Each thread writes **only its own** `partial[k]` cell: no race.
- main waits for all of them with `join` and combines the partial sums: the result is always **500500**.
- Passing `&k` as the argument would be a bug: the loop variable keeps changing.
- Neighboring cells share a cache line: heavy writing causes *false sharing*; summing in a local variable and writing once at the end is better.

Passing Arguments and Getting Results

```
1  typedef struct { int lo, hi; } range_t;
2  typedef struct { long total; int count; } result_t;
3
4  void *add_range(void *arg) {
5    range_t *a = arg;           /* void* → type */
6    result_t *s = malloc(sizeof *s); /* on heap */
7    s->total = 0; s->count = 0;
8    for (int i = a->lo; i <= a->hi; i++) {
9      s->total += i; s->count++;
10   }
11   return s;          /* = pthread_exit(s) */
12 }
13 int main(void) {
▶    range_t a = { 1, 100 };
15   pthread_t t;  void *ret;
16   pthread_create(&t, NULL, add_range, &a);
17   pthread_join(t, &ret);          /* ret = s */
18   result_t *s = ret;
19   printf("%d numbers, total %ld\n", s->count, s->total);
20   free(s);
21   return 0;
22 }
```



The argument is the struct `a` on main's stack. That frame lives until main joins T, so passing its address is safe.

Step 1 / 7

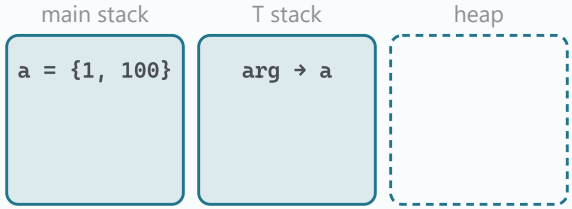
VARIABLES

`a.lo = 1` `a.hi = 100`

OUTPUT

Passing Arguments and Getting Results

```
1  typedef struct { int lo, hi; } range_t;
2  typedef struct { long total; int count; } result_t;
3
4  void *add_range(void *arg) {
5    range_t *a = arg;           /* void* → type */
6    result_t *s = malloc(sizeof *s); /* on heap */
7    s->total = 0; s->count = 0;
8    for (int i = a->lo; i <= a->hi; i++) {
9      s->total += i; s->count++;
10   }
11   return s;          /* = pthread_exit(s) */
12 }
13 int main(void) {
14   range_t a = { 1, 100 };
15   pthread_t t;  void *ret;
▶    pthread_create(&t, NULL, add_range, &a);
17   pthread_join(t, &ret);          /* ret = s */
18   result_t *s = ret;
19   printf("%d numbers, total %ld\n", s->count, s->total);
20   free(s);
21   return 0;
22 }
```



There is a single `void *` parameter: several values go into a struct and its **address** is passed.

Step 2 / 7

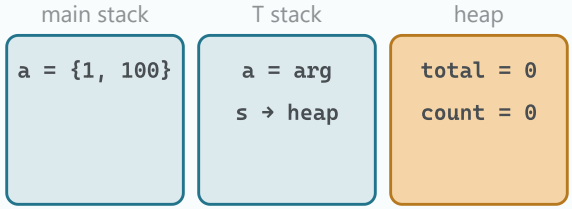
VARIABLES

`a.lo = 1` `a.hi = 100` `t = T1`

OUTPUT

Passing Arguments and Getting Results

```
1  typedef struct { int lo, hi; } range_t;
2  typedef struct { long total; int count; } result_t;
3
4  void *add_range(void *arg) {
▶    range_t *a = arg;            /* void* → type */
▶    result_t *s = malloc(sizeof *s); /* on heap */
7    s->total = 0; s->count = 0;
8    for (int i = a->lo; i <= a->hi; i++) {
9      s->total += i; s->count++;
10   }
11   return s;          /* = pthread_exit(s) */
12 }
13 int main(void) {
14   range_t a = { 1, 100 };
15   pthread_t t;  void *ret;
16   pthread_create(&t, NULL, add_range, &a);
17   pthread_join(t, &ret);           /* ret = s */
18   result_t *s = ret;
19   printf("%d numbers, total %ld\n", s->count, s->total);
20   free(s);
21   return 0;
22 }
```



T casts the `void *` to its real type. The result is allocated **on the heap**: T's own stack will vanish when T ends.

Step 3 / 7

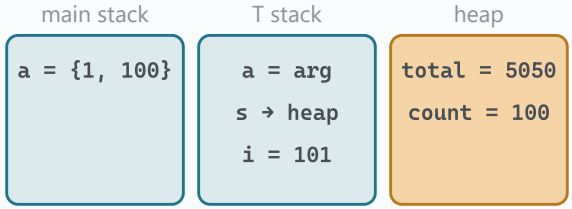
VARIABLES

`a.lo = 1` `a.hi = 100` `t = T1` `s = on heap`

OUTPUT

Passing Arguments and Getting Results

```
1  typedef struct { int lo, hi; } range_t;
2  typedef struct { long total; int count; } result_t;
3
4  void *add_range(void *arg) {
5    range_t *a = arg;           /* void* → type */
6    result_t *s = malloc(sizeof *s); /* on heap */
7    s->total = 0; s->count = 0;
▶    for (int i = a->lo; i <= a->hi; i++) {
▶      s->total += i; s->count++;
10   }
11   return s;          /* = pthread_exit(s) */
12 }
13 int main(void) {
14   range_t a = { 1, 100 };
15   pthread_t t;  void *ret;
16   pthread_create(&t, NULL, add_range, &a);
17   pthread_join(t, &ret);          /* ret = s */
18   result_t *s = ret;
19   printf("%d numbers, total %ld\n", s->count, s->total);
20   free(s);
21   return 0;
22 }
```



The loop sums 1..100: in the struct on the heap, total = 5050, count = 100.

Step 4 / 7

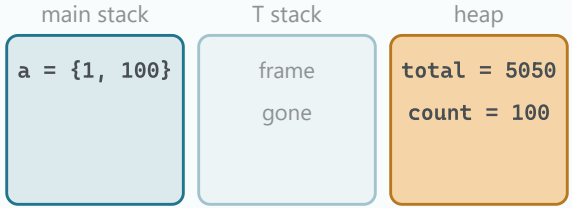
VARIABLES

`a.lo = 1` `a.hi = 100` `t = T1` `s.total = 5050` `s.count = 100`

OUTPUT

Passing Arguments and Getting Results

```
 1 typedef struct { int lo, hi; } range_t;
 2 typedef struct { long total; int count; } result_t;
 3
 4 void *add_range(void *arg) {
 5   range_t *a = arg;           /* void* → type */
 6   result_t *s = malloc(sizeof *s); /* on heap */
 7   s->total = 0; s->count = 0;
 8   for (int i = a->lo; i <= a->hi; i++) {
 9     s->total += i; s->count++;
10   }
▶    return s;          /* = pthread_exit(s) */
12 }
13 int main(void) {
14   range_t a = { 1, 100 };
15   pthread_t t;  void *ret;
16   pthread_create(&t, NULL, add_range, &a);
17   pthread_join(t, &ret);          /* ret = s */
18   result_t *s = ret;
19   printf("%d numbers, total %ld\n", s->count, s->total);
20   free(s);
21   return 0;
22 }
```



`return s` : the pointer is handed to whoever joins. T's local variables are now invalid, but the object on the heap **lives on**.

Step 5 / 7

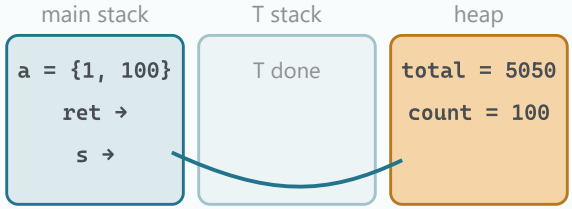
VARIABLES

`t = T1` `s.total = 5050` `s.count = 100`

OUTPUT

Passing Arguments and Getting Results

```
1  typedef struct { int lo, hi; } range_t;
2  typedef struct { long total; int count; } result_t;
3
4  void *add_range(void *arg) {
5    range_t *a = arg;           /* void* → type */
6    result_t *s = malloc(sizeof *s); /* on heap */
7    s->total = 0; s->count = 0;
8    for (int i = a->lo; i <= a->hi; i++) {
9      s->total += i; s->count++;
10   }
11   return s;          /* = pthread_exit(s) */
12 }
13 int main(void) {
14   range_t a = { 1, 100 };
15   pthread_t t;  void *ret;
16   pthread_create(&t, NULL, add_range, &a);
▶    pthread_join(t, &ret);          /* ret = s */
▶    result_t *s = ret;
19   printf("%d numbers, total %ld\n", s->count, s->total);
20   free(s);
21   return 0;
22 }
```



`pthread_join(t, &ret)` writes the return value into `ret` through a `void **`; the cast is done in main.

Step 6 / 7

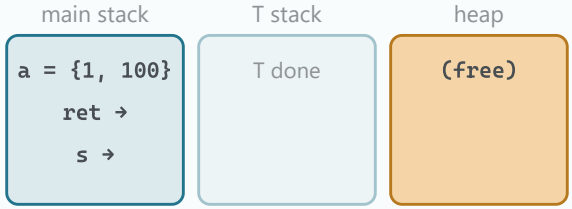
VARIABLES

`ret = s` `s.total = 5050` `s.count = 100`

OUTPUT

Passing Arguments and Getting Results

```
1  typedef struct { int lo, hi; } range_t;
2  typedef struct { long total; int count; } result_t;
3
4  void *add_range(void *arg) {
5    range_t *a = arg;           /* void* → type */
6    result_t *s = malloc(sizeof *s); /* on heap */
7    s->total = 0; s->count = 0;
8    for (int i = a->lo; i <= a->hi; i++) {
9      s->total += i; s->count++;
10   }
11   return s;          /* = pthread_exit(s) */
12 }
13 int main(void) {
14   range_t a = { 1, 100 };
15   pthread_t t;  void *ret;
16   pthread_create(&t, NULL, add_range, &a);
17   pthread_join(t, &ret);          /* ret = s */
18   result_t *s = ret;
▶    printf("%d numbers, total %ld\n", s->count, s->total);
▶    free(s);
21   return 0;
22 }
```



The result is printed and `free(s)` is called: ownership of the object passed to main through join.

Step 7 / 7

VARIABLES

`ret = s` `s.total = 5050` `s.count = 100`

OUTPUT

```
100 numbers, total 5050
```

A Common Bug: Returning a Result on the Stack

Wrong

```
void *add_range(void *arg) {
  result_t s = { 0, 0 }; /* on T's stack */
  /* ... fill in s ... */
  return &s;             /* BUG */
}
```

Right

```
void *add_range(void *arg) {
  result_t *s = malloc(sizeof *s);
  if (s == NULL) return NULL;
  /* ... fill in s ... */
  return s;    /* on heap: lives on */
}  /* main: join, use, free(ret) */
```



T running: s is in add_range's stack frame

In the wrong version the result is the local variable `s` on T's stack, and `&s` is returned.

Step 1 / 3

The same rule applies to arguments: the address passed must stay alive as long as the thread uses it. gcc warns (`-Wreturn-local-addr`); treat the warning as an error.

A Common Bug: Returning a Result on the Stack

Wrong

```
void *add_range(void *arg) {
  result_t s = { 0, 0 }; /* on T's stack */
  /* ... fill in s ... */
  return &s;             /* BUG */
}
```

Right

```
void *add_range(void *arg) {
  result_t *s = malloc(sizeof *s);
  if (s == NULL) return NULL;
  /* ... fill in s ... */
  return s;    /* on heap: lives on */
}  /* main: join, use, free(ret) */
```



T returned: main still has the address, but the memory belongs to no one now

When the function returns its frame is gone; when the thread ends its whole stack is released or may be given to another thread.

Step 2 / 3

The same rule applies to arguments: the address passed must stay alive as long as the thread uses it. gcc warns (`-Wreturn-local-addr`); treat the warning as an error.

A Common Bug: Returning a Result on the Stack

Wrong

```
void *add_range(void *arg) {
  result_t s = { 0, 0 }; /* on T's stack */
  /* ... fill in s ... */
  return &s;             /* BUG */
}
```

Right

```
void *add_range(void *arg) {
  result_t *s = malloc(sizeof *s);
  if (s == NULL) return NULL;
  /* ... fill in s ... */
  return s;    /* on heap: lives on */
}  /* main: join, use, free(ret) */
```



main reads it: undefined behavior

If main reads this address the result is **undefined**: a garbage value, SIGSEGV or, worst of all, a value that sometimes looks right. The heap version on the right is correct: `malloc` is checked for NULL and the result is `free`d after join.

Step 3 / 3

The same rule applies to arguments: the address passed must stay alive as long as the thread uses it. gcc warns (`-Wreturn-local-addr`); treat the warning as an error.

Java: Thread, start and join

```
1  public class Sum {
2    static int total = 0;
3    public static void main(String[] a)
4        throws InterruptedException {
▶      Thread t = new Thread(() -> {
▶        for (int i = 1; i <= 5; i++) total += i;
▶      });
8      t.start();
9      System.out.println("main: waiting");
10     t.join();
11     System.out.println("total = " + total);
12   }
13 }
```

main	RUNNABLE
t	NEW (created, not started)

`new Thread(lambda)` only creates the object; the state is **NEW**.

Step 1 / 6

VARIABLES

`total = 0` `t = NEW`

OUTPUT

Compiled and run with JDK 21; the output is the same every time.

Java: Thread, start and join

```
1  public class Sum {
2    static int total = 0;
3    public static void main(String[] a)
4        throws InterruptedException {
5      Thread t = new Thread(() -> {
6        for (int i = 1; i <= 5; i++) total += i;
7      });
▶      t.start();
9      System.out.println("main: waiting");
10     t.join();
11     System.out.println("total = " + total);
12   }
13 }
```

main — RUNNABLE

t — RUNNABLE

`start()` starts a new OS thread (calling `run()` would not!).

Step 2 / 6

VARIABLES

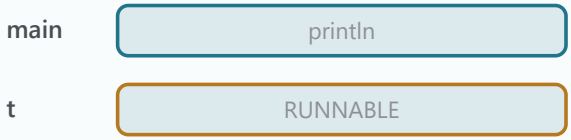
`total = 0` `t = RUNNABLE`

OUTPUT

Compiled and run with JDK 21; the output is the same every time.

Java: Thread, start and join

```
1  public class Sum {
2    static int total = 0;
3    public static void main(String[] a)
4        throws InterruptedException {
5      Thread t = new Thread(() -> {
6        for (int i = 1; i <= 5; i++) total += i;
7      });
8      t.start();
▶      System.out.println("main: waiting");
10     t.join();
11     System.out.println("total = " + total);
12   }
13 }
```



main keeps printing; it may run at the same time as t.

Step 3 / 6

VARIABLES

`total = 0` `t = RUNNABLE`

OUTPUT

```
main: waiting
```

Compiled and run with JDK 21; the output is the same every time.

Java: Thread, start and join

```
1  public class Sum {
2    static int total = 0;
3    public static void main(String[] a)
4        throws InterruptedException {
5      Thread t = new Thread(() -> {
6        for (int i = 1; i <= 5; i++) total += i;
7      });
8      t.start();
9      System.out.println("main: waiting");
▶      t.join();
11     System.out.println("total = " + total);
12   }
13 }
```

main — WAITING (join)

t — RUNNABLE

`join()` : main is **WAITING** until t finishes.

Step 4 / 6

VARIABLES

`total = 0` `t = RUNNABLE`

OUTPUT

```
main: waiting
```

Compiled and run with JDK 21; the output is the same every time.

Java: Thread, start and join

```
1  public class Sum {
2    static int total = 0;
3    public static void main(String[] a)
4        throws InterruptedException {
5      Thread t = new Thread(() -> {
6        for (int i = 1; i <= 5; i++) total += i;
7      });
8      t.start();
9      System.out.println("main: waiting");
10     t.join();
11     System.out.println("total = " + total);
12   }
13 }
```

main	waking up
t	TERMINATED

t finishes the loop: total = 15, state **TERMINATED**.

Step 5 / 6

VARIABLES

`total = 15` `t = TERMINATED`

OUTPUT

```
main: waiting
```

Compiled and run with JDK 21; the output is the same every time.

Java: Thread, start and join

```
1  public class Sum {
2    static int total = 0;
3    public static void main(String[] a)
4        throws InterruptedException {
5      Thread t = new Thread(() -> {
6        for (int i = 1; i <= 5; i++) total += i;
7      });
8      t.start();
9      System.out.println("main: waiting");
10     t.join();
▶      System.out.println("total = " + total);
12   }
13 }
```



join also guarantees that main sees what t wrote (happens-before).

Step 6 / 6

VARIABLES

`total = 15` `t = TERMINATED`

OUTPUT

```
main: waiting
total = 15
```

Compiled and run with JDK 21; the output is the same every time.

Java 21: Virtual Threads

```
import java.time.Duration;
import java.util.concurrent.*;
import java.util.stream.IntStream;

public class Virtual {
  public static void main(String[] args) {
    try (var ex = Executors.newVirtualThreadPerTaskExecutor()) {
      IntStream.range(0, 10_000).forEach(i ->
        ex.submit(() -> {
          Thread.sleep(Duration.ofSeconds(1));
          return i;
        }));
    }  // close(): waits for all tasks
    System.out.println("done");
  }
}
```

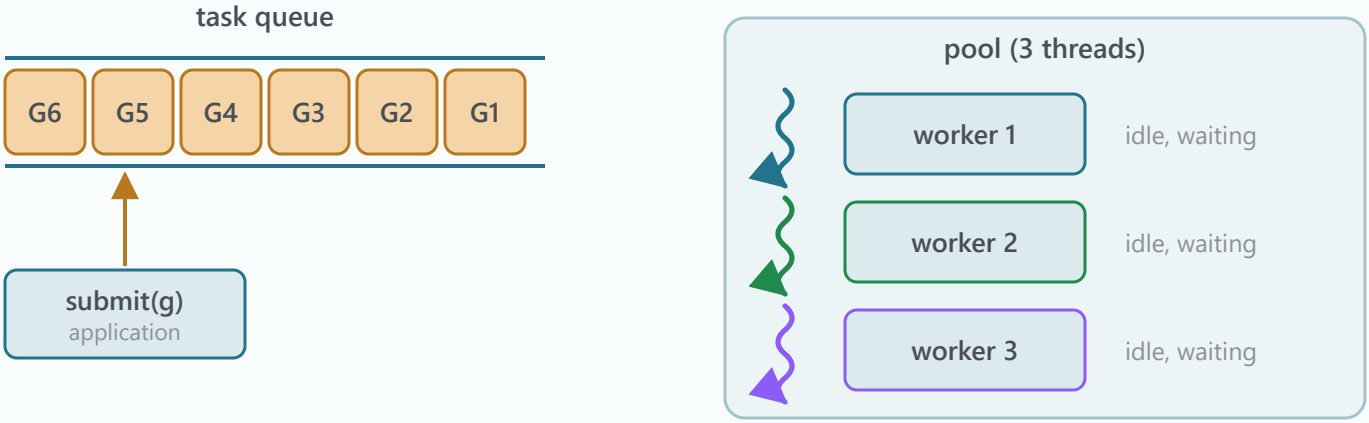
- Each of the 10,000 tasks sleeps 1 s in its own **virtual thread**.
- In our measurement (JDK 21) all finished in about **2 seconds**: a few carrier threads were enough.
- 10,000 platform threads would mean 10,000 kernel threads and their stacks.
- Good fit: many tasks that **wait on I/O** (servers). No gain for CPU-heavy work.
- Classic creation works too: `Thread.ofVirtual().start(r)`, `Thread.ofPlatform()`.

Implicit Threading

- With hundreds of threads, managing correctness by hand gets hard.
- The solution: the programmer identifies **tasks** (work that can run in parallel); **compilers and run-time libraries** create and manage the threads.

Approach	Where	Idea
Thread pool	Java `ExecutorService`, Windows thread pool	Ready-made threads take tasks from a queue.
Fork-join	Java `ForkJoinPool`, parallel streams	Split the task, solve in parallel, combine the results.
OpenMP	C, C++, Fortran	Parallel regions through compiler directives.
Grand Central Dispatch	macOS, iOS (libdispatch)	Blocks are placed on dispatch queues.
oneTBB (formerly Intel TBB)	C++ template library	`parallel_for`, task scheduling, work stealing.

Thread Pools

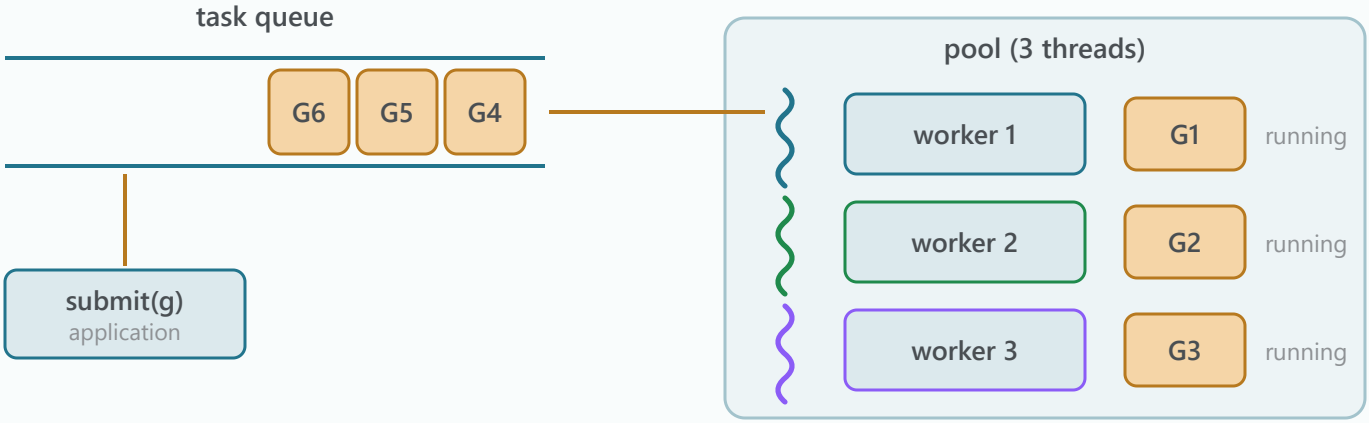


The pool creates a **fixed number** of threads up front; the application puts tasks in the queue.

Step 1 / 4

Benefits: faster service, protection against too many threads, separating how a task runs (delayed, periodic) from the task itself.

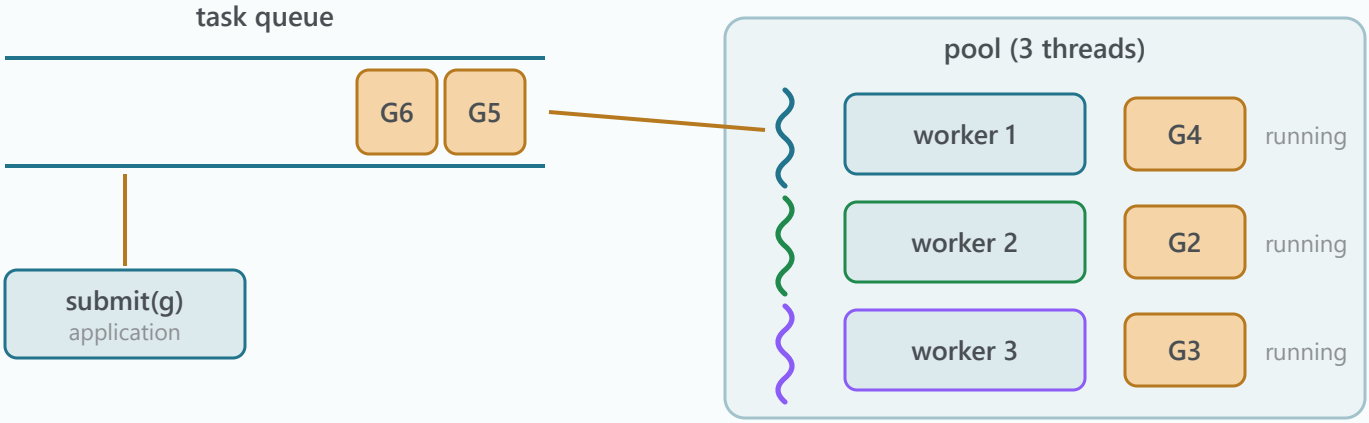
Thread Pools



Idle workers take tasks from the queue: **no thread-creation cost**.

Benefits: faster service, protection against too many threads, separating how a task runs (delayed, periodic) from the task itself.

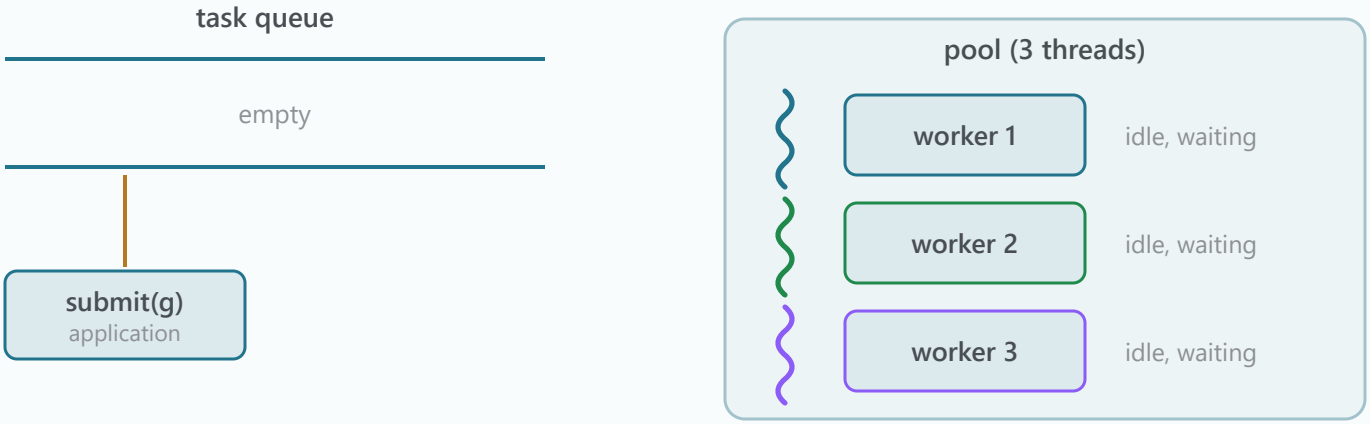
Thread Pools



When worker 1 finishes G1 it does not die; it takes the next task, G4, at once.

Benefits: faster service, protection against too many threads, separating how a task runs (delayed, periodic) from the task itself.

Thread Pools



When the queue is empty the workers wait for the next task. The number of concurrent threads is **bounded** by the pool size.

Benefits: faster service, protection against too many threads, separating how a task runs (delayed, periodic) from the task itself.

Java ExecutorService

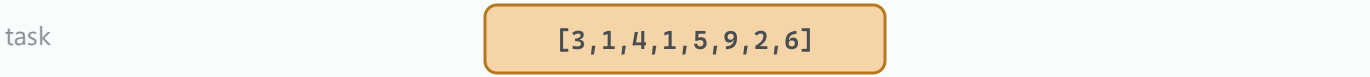
```
ExecutorService pool = Executors.newFixedThreadPool(2);
List<Future<Integer>> results = new ArrayList<>();
for (int g = 1; g <= 4; g++) {
  final int n = g;
  results.add(pool.submit(() -> n * n));
}
int total = 0;
for (Future<Integer> f : results)
  total += f.get();       // waits for result
System.out.println("total = " + total);
pool.shutdown();
```

OUTPUT (JDK 21)

```
total = 30
```

- Four tasks (1^2, 2^2, 3^2, 4^2) are given to a pool of two threads.
- `submit` returns a `Future`; `get()` waits until the result is ready.
- Pool size: ≈ the core count for CPU-bound work (`availableProcessors()`); more for I/O-bound work.
- Other factories: `newCachedThreadPool`, `newScheduledThreadPool`, `newVirtualThreadPerTaskExecutor`.

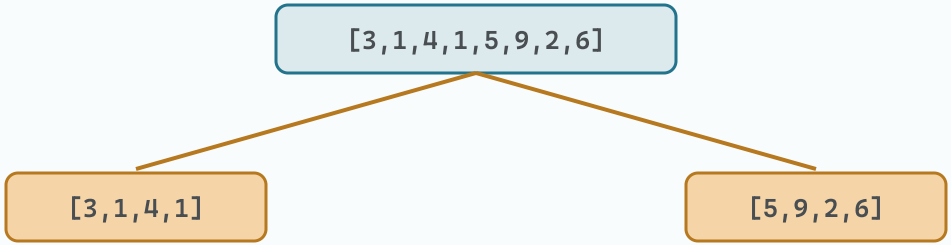
Fork-Join



The big task: the sum of 8 numbers. If it is above the threshold, **split** it.

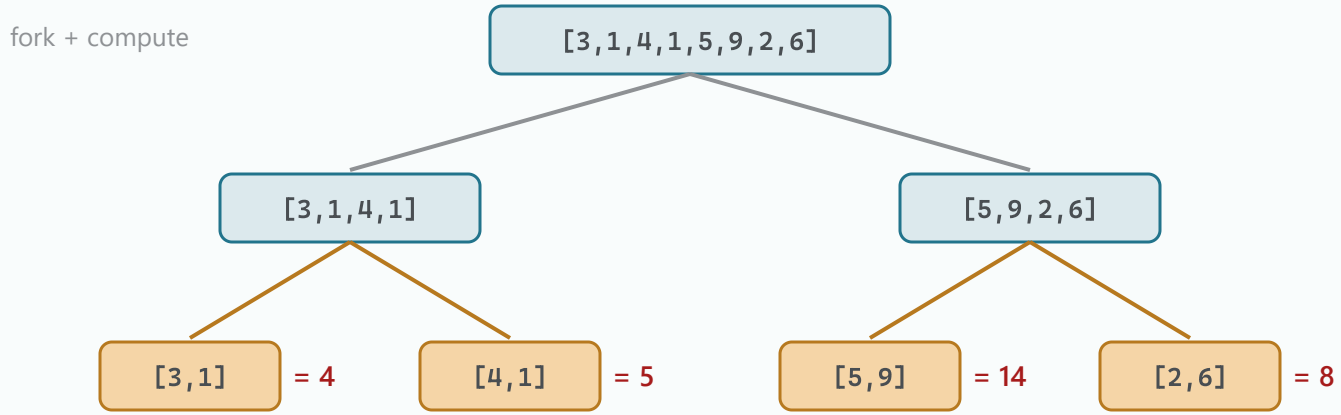
Fork-Join

fork



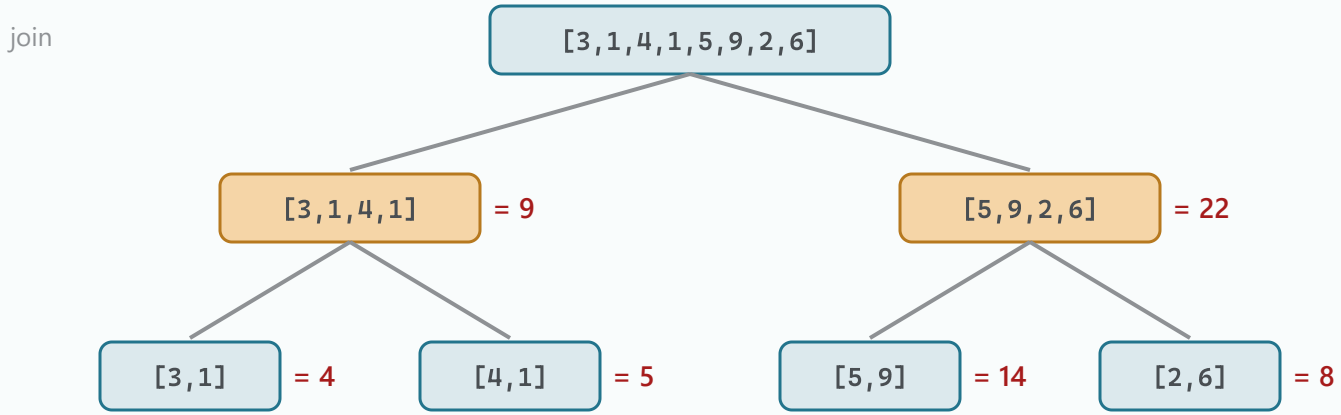
fork: two subtasks are created; they may run on different threads in the pool.

Fork-Join



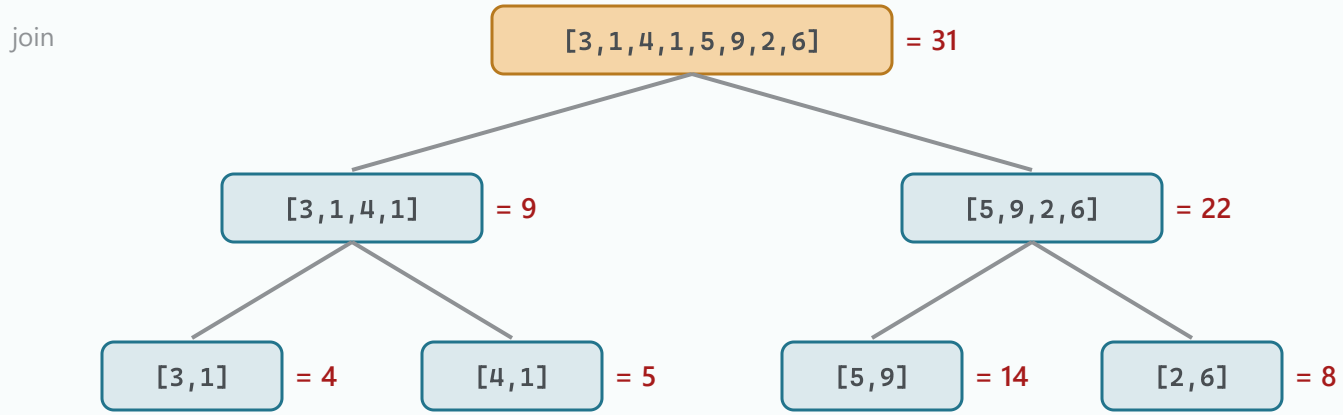
The subtasks are split again; leaves at the threshold (2 elements) are computed directly: 4, 5, 14, 8.

Fork-Join



join: a parent waits for its children and combines them: 4+5 = 9, 14+8 = 22.

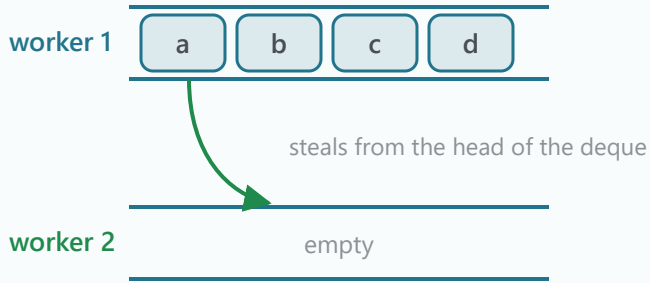
Fork-Join



Root result: 9 + 22 = **31**. A typical case of synchronous threading.

Java ForkJoinPool and Work Stealing

```
class Sum extends RecursiveTask<Long> {
  protected Long compute() {
    if (hi - lo <= LIMIT) {       // small: do it
      long s = 0;
      for (int i = lo; i < hi; i++) s += a[i];
      return s;
    }
    int mid = (lo + hi) / 2;
    Sum left = new Sum(a, lo, mid);
    Sum right = new Sum(a, mid, hi);
    left.fork();                  // start async
    long rightSum = right.compute();
    return left.join() + rightSum;
  }
}
long s = ForkJoinPool.commonPool()
           .invoke(new Sum(a, 0, a.length));
```

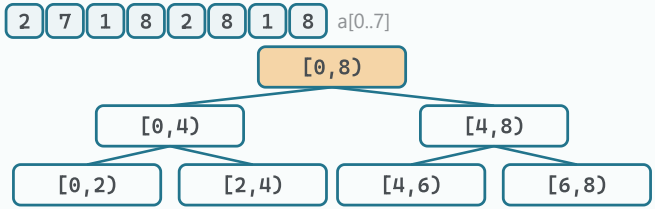


owner takes from its own end (the tail)

Output for the array 1..100 000 (JDK 21): `total = 5000050000`. An idle worker **steals** tasks from another's deque (work stealing); the load balances itself.

Code Trace: Summing with Fork-Join

```
1  protected Long compute() {     // LIMIT = 2
▶    if (hi - lo <= LIMIT) {
3      long s = 0;
4      for (int i = lo; i < hi; i++)
5        s += a[i];
6      return s;
7    }
8    int mid = (lo + hi) / 2;
9    Sum left = new Sum(a, lo, mid);
10   Sum right = new Sum(a, mid, hi);
11   left.fork();                   // to deque
12   long rightSum = right.compute();
13   return left.join() + rightSum;
14 }
```



work deque: empty

The root task sums 8 elements. 8 − 0 = 8 > LIMIT, so it is not computed directly.

Step 1 / 11

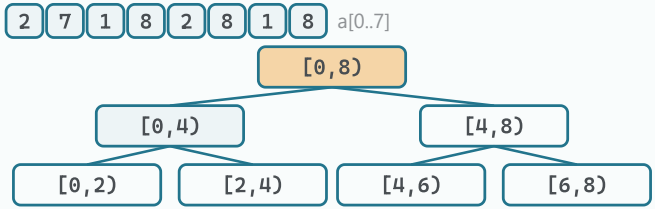
VARIABLES

`lo = 0` `hi = 8`

a = {2, 7, 1, 8, 2, 8, 1, 8}, a single-worker pool: `new ForkJoinPool(1).invoke(new Sum(a, 0, 8))`; output (JDK 21): `total = 37`.

Code Trace: Summing with Fork-Join

```
1  protected Long compute() {     // LIMIT = 2
2    if (hi - lo <= LIMIT) {
3      long s = 0;
4      for (int i = lo; i < hi; i++)
5        s += a[i];
6      return s;
7    }
▶    int mid = (lo + hi) / 2;
▶    Sum left = new Sum(a, lo, mid);
▶    Sum right = new Sum(a, mid, hi);
▶    left.fork();                    // to deque
12   long rightSum = right.compute();
13   return left.join() + rightSum;
14 }
```



work deque: [0,4)

The task is split in two. `left.fork()` puts [0,4) on the worker's **deque**; it does not run yet.

Step 2 / 11

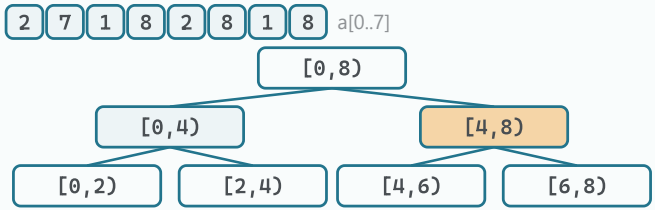
VARIABLES

`lo = 0` `hi = 8` `mid = 4`

a = {2, 7, 1, 8, 2, 8, 1, 8}, a single-worker pool: `new ForkJoinPool(1).invoke(new Sum(a, 0, 8))`; output (JDK 21): `total = 37`.

Code Trace: Summing with Fork-Join

```
1  protected Long compute() {     // LIMIT = 2
2    if (hi - lo <= LIMIT) {
3      long s = 0;
4      for (int i = lo; i < hi; i++)
5        s += a[i];
6      return s;
7    }
8    int mid = (lo + hi) / 2;
9    Sum left = new Sum(a, lo, mid);
10   Sum right = new Sum(a, mid, hi);
11   left.fork();                   // to deque
▶    long rightSum = right.compute();
13   return left.join() + rightSum;
14 }
```



work deque: [0,4)

The right half runs on the same thread, directly via `compute()`.

Step 3 / 11

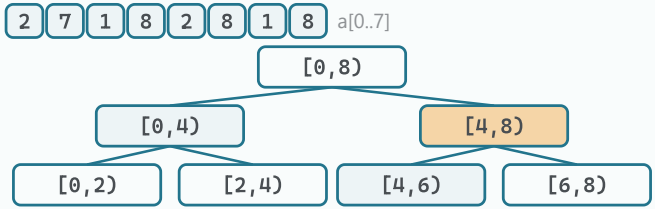
VARIABLES

`lo = 4` `hi = 8`

a = {2, 7, 1, 8, 2, 8, 1, 8}, a single-worker pool: `new ForkJoinPool(1).invoke(new Sum(a, 0, 8))`; output (JDK 21): `total = 37`.

Code Trace: Summing with Fork-Join

```
 1 protected Long compute() {     // LIMIT = 2
 2   if (hi - lo <= LIMIT) {
 3     long s = 0;
 4     for (int i = lo; i < hi; i++)
 5       s += a[i];
 6     return s;
 7   }
 ▶   int mid = (lo + hi) / 2;
 ▶   Sum left = new Sum(a, lo, mid);
 ▶   Sum right = new Sum(a, mid, hi);
 ▶   left.fork();                    // to deque
12   long rightSum = right.compute();
13   return left.join() + rightSum;
14 }
```



work deque: [0,4) [4,6)

[4,8) is split too: [4,6) goes on the deque.

Step 4 / 11

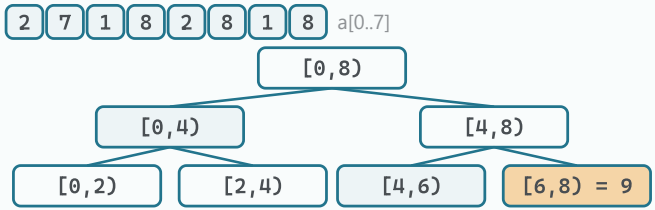
VARIABLES

`lo = 4` `hi = 8` `mid = 6`

a = {2, 7, 1, 8, 2, 8, 1, 8}, a single-worker pool: `new ForkJoinPool(1).invoke(new Sum(a, 0, 8))` ; output (JDK 21): `total = 37` .

Code Trace: Summing with Fork-Join

```
1  protected Long compute() {     // LIMIT = 2
▶    if (hi - lo <= LIMIT) {
▶      long s = 0;
▶      for (int i = lo; i < hi; i++)
▶        s += a[i];
▶      return s;
7    }
8    int mid = (lo + hi) / 2;
9    Sum left = new Sum(a, lo, mid);
10   Sum right = new Sum(a, mid, hi);
11   left.fork();                    // to deque
12   long rightSum = right.compute();
13   return left.join() + rightSum;
14 }
```



work deque: [0,4) [4,6)

[6,8): 2 elements, at the limit. Summed directly: 1 + 8 = **9**.

Step 5 / 11

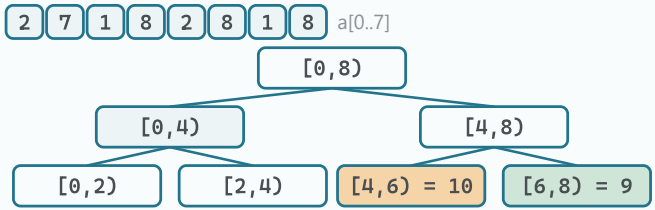
VARIABLES

`lo = 6` `hi = 8` `s = 9`

a = {2, 7, 1, 8, 2, 8, 1, 8}, a single-worker pool: `new ForkJoinPool(1).invoke(new Sum(a, 0, 8))`; output (JDK 21): `total = 37`.

Code Trace: Summing with Fork-Join

```
1  protected Long compute() {     // LIMIT = 2
2    if (hi - lo <= LIMIT) {
3      long s = 0;
4      for (int i = lo; i < hi; i++)
5        s += a[i];
6      return s;
7    }
8    int mid = (lo + hi) / 2;
9    Sum left = new Sum(a, lo, mid);
10   Sum right = new Sum(a, mid, hi);
11   left.fork();                   // to deque
12   long rightSum = right.compute();
▶    return left.join() + rightSum;
14 }
```



work deque: [0,4)

`left.join()` : [4,6) is still at the end of the deque; nobody stole it. The worker pops it and runs it **itself**: 2 + 8 = 10.

Step 6 / 11

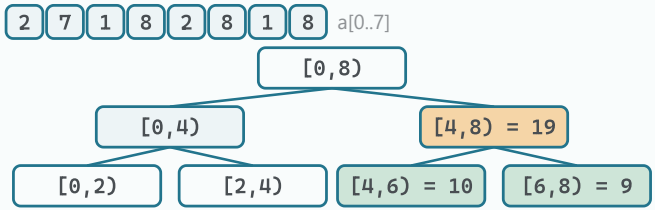
VARIABLES

`lo = 4` `hi = 6` `s = 10`

a = {2, 7, 1, 8, 2, 8, 1, 8}, a single-worker pool: `new ForkJoinPool(1).invoke(new Sum(a, 0, 8))` ; output (JDK 21): `total = 37` .

Code Trace: Summing with Fork-Join

```
1  protected Long compute() {     // LIMIT = 2
2    if (hi - lo <= LIMIT) {
3      long s = 0;
4      for (int i = lo; i < hi; i++)
5        s += a[i];
6      return s;
7    }
8    int mid = (lo + hi) / 2;
9    Sum left = new Sum(a, lo, mid);
10   Sum right = new Sum(a, mid, hi);
11   left.fork();                    // to deque
12   long rightSum = right.compute();
▶    return left.join() + rightSum;
14 }
```



work deque: [0,4)

[4,8) = 10 + 9 = **19**, returned to the root.

Step 7 / 11

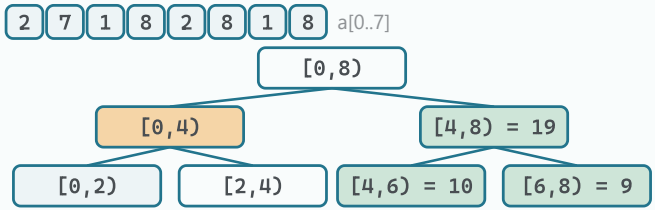
VARIABLES

`lo = 4` `hi = 8` `rightSum = 9`

a = {2, 7, 1, 8, 2, 8, 1, 8}, a single-worker pool: `new ForkJoinPool(1).invoke(new Sum(a, 0, 8))` ; output (JDK 21): `total = 37` .

Code Trace: Summing with Fork-Join

```
1  protected Long compute() {     // LIMIT = 2
2    if (hi - lo <= LIMIT) {
3      long s = 0;
4      for (int i = lo; i < hi; i++)
5        s += a[i];
6      return s;
7    }
▶    int mid = (lo + hi) / 2;
▶    Sum left = new Sum(a, lo, mid);
▶    Sum right = new Sum(a, mid, hi);
▶    left.fork();                    // to deque
12   long rightSum = right.compute();
▶    return left.join() + rightSum;
14 }
```



work deque: [0,2)

The root calls `left.join()`: [0,4) is popped and run; it is split as well, and [0,2) goes on the deque.

Step 8 / 11

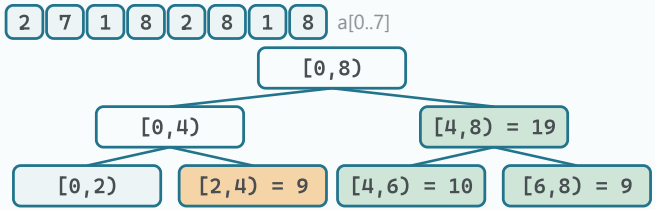
VARIABLES

`lo = 0` `hi = 4` `mid = 2`

a = {2, 7, 1, 8, 2, 8, 1, 8}, a single-worker pool: `new ForkJoinPool(1).invoke(new Sum(a, 0, 8))`; output (JDK 21): `total = 37`.

Code Trace: Summing with Fork-Join

```
1  protected Long compute() {     // LIMIT = 2
▶    if (hi - lo <= LIMIT) {
▶      long s = 0;
▶      for (int i = lo; i < hi; i++)
▶        s += a[i];
▶      return s;
7    }
8    int mid = (lo + hi) / 2;
9    Sum left = new Sum(a, lo, mid);
10   Sum right = new Sum(a, mid, hi);
11   left.fork();                    // to deque
▶    long rightSum = right.compute();
13   return left.join() + rightSum;
14 }
```



work deque: [0,2)

[2,4) is a leaf: 1 + 8 = **9**.

Step 9 / 11

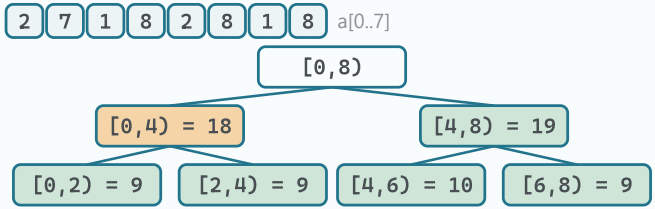
VARIABLES

`lo = 2` `hi = 4` `s = 9`

a = {2, 7, 1, 8, 2, 8, 1, 8}, a single-worker pool: `new ForkJoinPool(1).invoke(new Sum(a, 0, 8))` ; output (JDK 21): `total = 37` .

Code Trace: Summing with Fork-Join

```
1  protected Long compute() {     // LIMIT = 2
2    if (hi - lo <= LIMIT) {
3      long s = 0;
4      for (int i = lo; i < hi; i++)
5        s += a[i];
6      return s;
7    }
8    int mid = (lo + hi) / 2;
9    Sum left = new Sum(a, lo, mid);
10   Sum right = new Sum(a, mid, hi);
11   left.fork();                   // to deque
12   long rightSum = right.compute();
▶    return left.join() + rightSum;
14 }
```



work deque: empty

join runs the leaf [0,2): 2 + 7 = 9; [0,4) = 9 + 9 = **18**.

Step 10 / 11

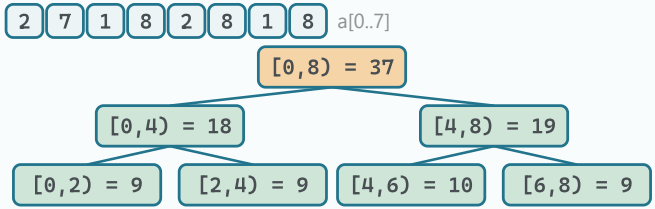
VARIABLES

`lo = 0` `hi = 4` `rightSum = 9`

a = {2, 7, 1, 8, 2, 8, 1, 8}, a single-worker pool: `new ForkJoinPool(1).invoke(new Sum(a, 0, 8))` ; output (JDK 21): `total = 37` .

Code Trace: Summing with Fork-Join

```
1  protected Long compute() {     // LIMIT = 2
2    if (hi - lo <= LIMIT) {
3      long s = 0;
4      for (int i = lo; i < hi; i++)
5        s += a[i];
6      return s;
7    }
8    int mid = (lo + hi) / 2;
9    Sum left = new Sum(a, lo, mid);
10   Sum right = new Sum(a, mid, hi);
11   left.fork();                   // to deque
12   long rightSum = right.compute();
▶    return left.join() + rightSum;
14 }
```



work deque: empty

Root: 18 + 19 = **37**. If the pool had an idle worker, it could **steal** tasks such as [0,4) from the deque; the order changes, the result does not.

Step 11 / 11

VARIABLES

`lo = 0` `hi = 8` `rightSum = 19`

a = {2, 7, 1, 8, 2, 8, 1, 8}, a single-worker pool: `new ForkJoinPool(1).invoke(new Sum(a, 0, 8))`; output (JDK 21): `total = 37`.

OpenMP

```c
#include <omp.h>
#include <stdio.h>

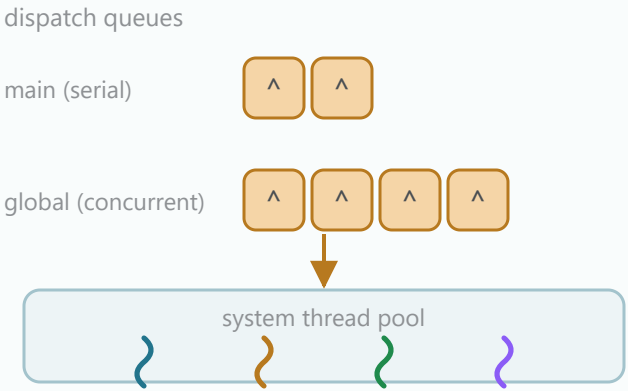
int main(void) {
  #pragma omp parallel
  printf("hello from %d\n", omp_get_thread_num());

  double a[1000], b[1000], c[1000], s = 0;
  /* ... fill a and b ... */
  #pragma omp parallel for reduction(+:s)
  for (int i = 0; i < 1000; i++) {
    c[i] = a[i] + b[i];
    s += c[i];
  }
  return 0;
}
```

- **Compiler directives** and a library for C, C++ and Fortran, for shared-memory systems.
- `#pragma omp parallel` : a **parallel region**; by default as many threads as cores (set with `OMP_NUM_THREADS` or `num_threads(n)`).
- `parallel for` : splits the loop iterations among threads; `reduction` combines the partial sums safely.
- Compile with `gcc -fopenmp` . The order of the “hello” lines can change on every run.

Grand Central Dispatch and oneTBB

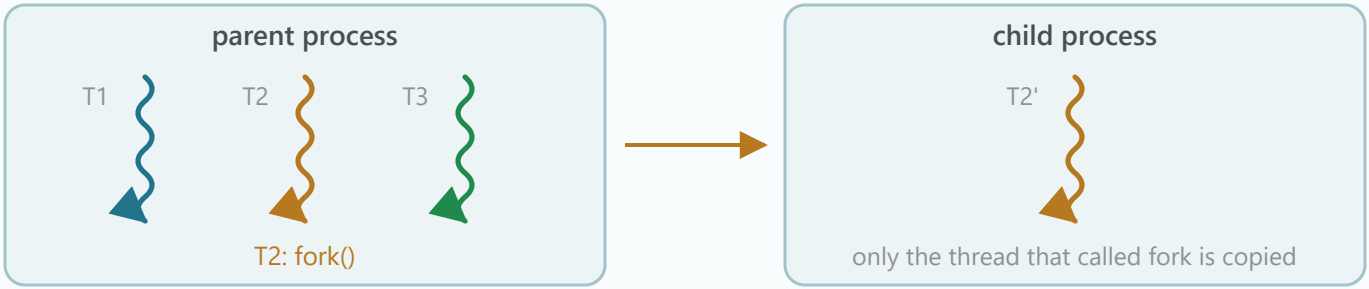
- Apple's run-time library and C/Objective-C language extension (**blocks**) for macOS/iOS: `^{ printf("I am a block"); }`
- Blocks are placed on **dispatch queues**; a serial queue runs them in order, a concurrent queue in parallel. QoS classes set the priority.
- A block removed from a queue is assigned to an idle thread in the pool; thread management is entirely up to GCD.




```
dispatch_queue_t q = dispatch_get_global_queue(
                       QOS_CLASS_USER_INITIATED, 0);
dispatch_async(q, ^{ puts("I am a block"); });
```

oneTBB (the open-source successor of Intel Threading Building Blocks): `tbb::parallel_for(0, n, [&](int i) { f(v[i]); });` Its own scheduler spreads tasks over threads using work stealing. In Swift, `async/await` has largely replaced GCD.

fork() and exec() with Threads



A lock held by T1 may stay locked forever in the child!

- Some UNIX systems offered two versions of `fork` (copying all threads or only the caller). **POSIX:** only the calling thread is copied.
- That is right if `exec` is called right after `fork`: the new program replaces the whole address space anyway.
- In the child, only async-signal-safe functions are safe until `exec`; `pthread_atfork` helps put locks in order. `posix_spawn` is often the better choice.

Signal Handling

A signal notifies a process that an event has occurred. In a multithreaded process the question is: **which thread** should receive it?

Option	Example
To the thread the signal applies to	synchronous: `SIGSEGV`, `SIGFPE`
To every thread in the process	some termination cases
To any thread not blocking it	POSIX process signal: `SIGINT`
To a dedicated thread that receives all signals	a `sigwait` loop

- Signal **handlers** are process-wide; the signal **mask** is per thread: `pthread_sigmask`.
- To a specific thread: `pthread_kill(tid, sig)`.
- Good practice: block signals in all threads and let one thread wait with `sigwait`.
- With user-level threads the kernel **cannot address** the right thread; it knows only the process.
- Windows has no signals; the closest thing is the APC (asynchronous procedure call).



Thread Cancellation

Asynchronous cancellation

The target is terminated *immediately*. It may die holding a lock or halfway through an update: dangerous.

Deferred cancellation (default)

The target checks for a cancellation request at specific *cancellation points* and terminates in an orderly way.



main calls `pthread_cancel(T)`; in deferred mode this is only a **request**, and T keeps running.

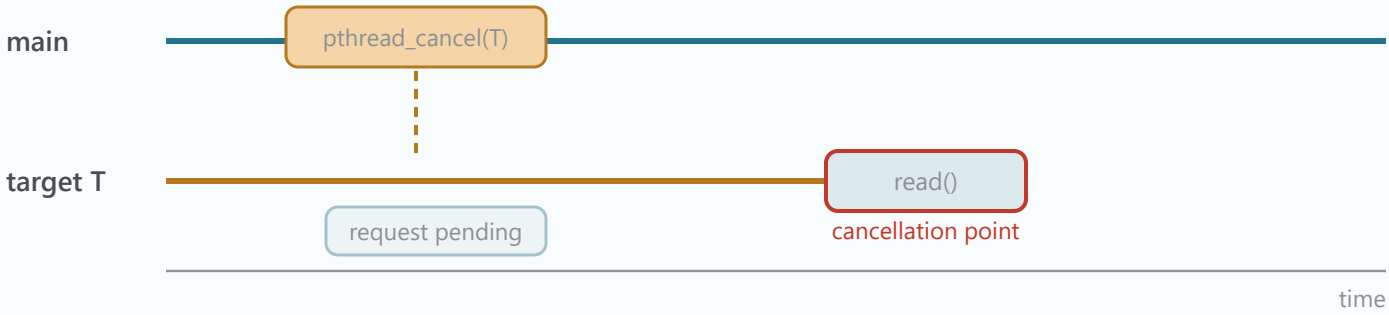
Thread Cancellation

Asynchronous cancellation

The target is terminated *immediately*. It may die holding a lock or halfway through an update: dangerous.

Deferred cancellation (default)

The target checks for a cancellation request at specific *cancellation points* and terminates in an orderly way.



T reaches a **cancellation point**: `read`, `sleep`, `pthread_cond_wait`, `pthread_testcancel()`...

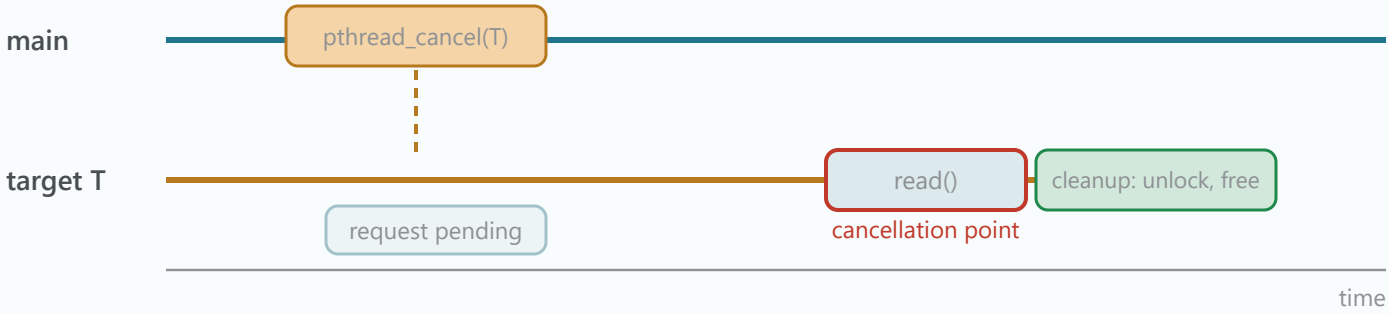
Thread Cancellation

Asynchronous cancellation

The target is terminated *immediately*. It may die holding a lock or halfway through an update: dangerous.

Deferred cancellation (default)

The target checks for a cancellation request at specific *cancellation points* and terminates in an orderly way.



The **cleanup handlers** registered with `pthread_cleanup_push` run: locks are released, memory is freed.

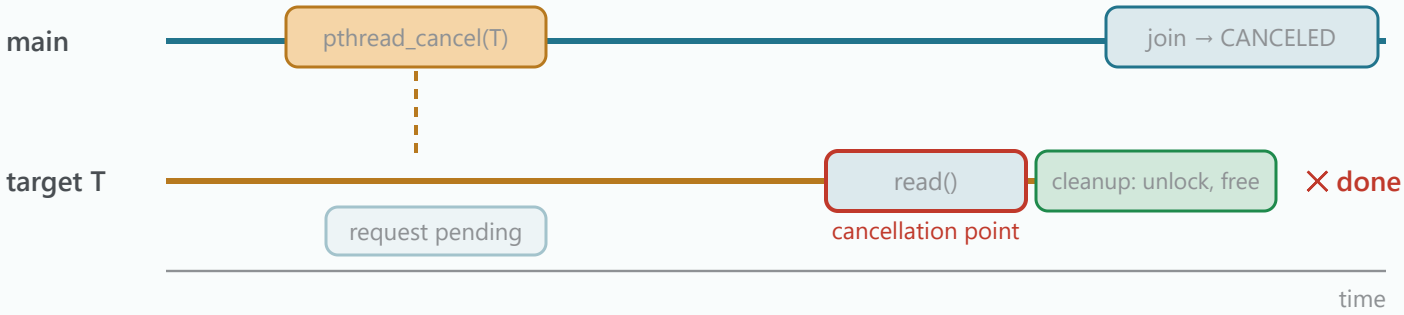
Thread Cancellation

Asynchronous cancellation

The target is terminated *immediately*. It may die holding a lock or halfway through an update: dangerous.

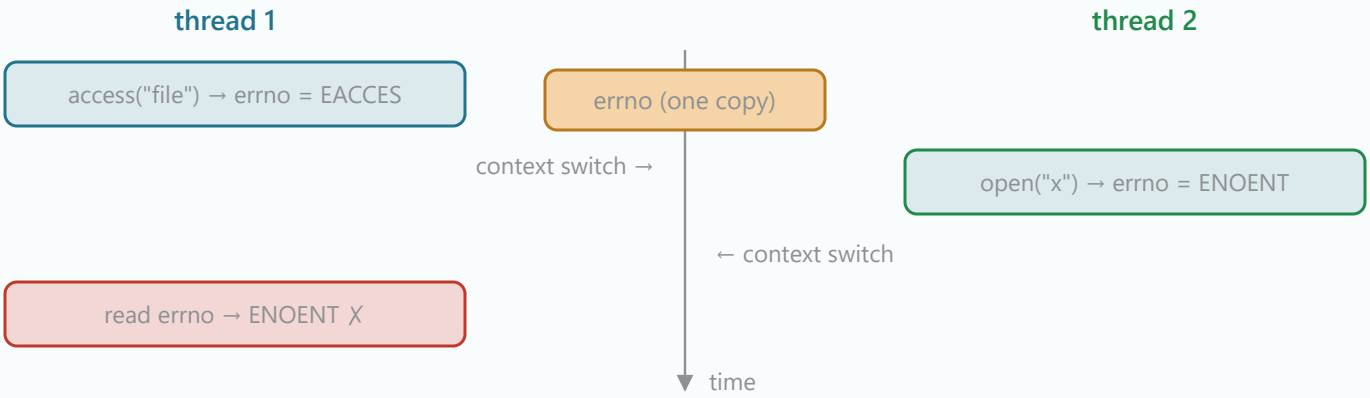
Deferred cancellation (default)

The target checks for a cancellation request at specific *cancellation points* and terminates in an orderly way.



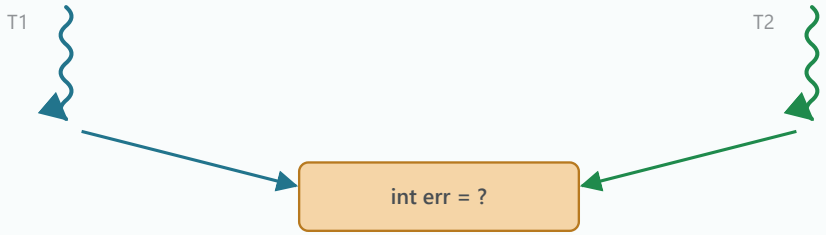
T terminates; `pthread_join` returns `PTHREAD_CANCELED`. Java's counterpart is the cooperative `interrupt()`; `Thread.stop()` has not worked since JDK 20.

Conflicts over Global Variables



- **Global** variables designed for single-threaded programs (UNIX `errno`) clash between threads.
- The fix: forbid the variable, or give each thread **its own copy**. In today's C libraries `errno` is thread-local.

Thread-Local Storage (TLS)



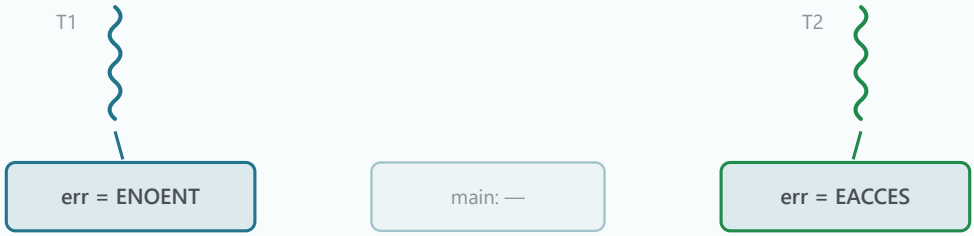
A plain global: `int err;` both threads read and write **the same** variable.

Step 1 / 3

- Unlike a local variable, it lives **across** function calls; like a global, but private to the thread.
- C11 `_Thread_local`, C23/C++ `thread_local`, GCC `__thread`; Pthreads: `pthread_key_create`, `pthread_setspecific`.
- Typical uses: `errno`, a transaction ID, a random-number generator.

```
static ThreadLocal<Integer> id =
  ThreadLocal.withInitial(() -> 0);
```

Thread-Local Storage (TLS)



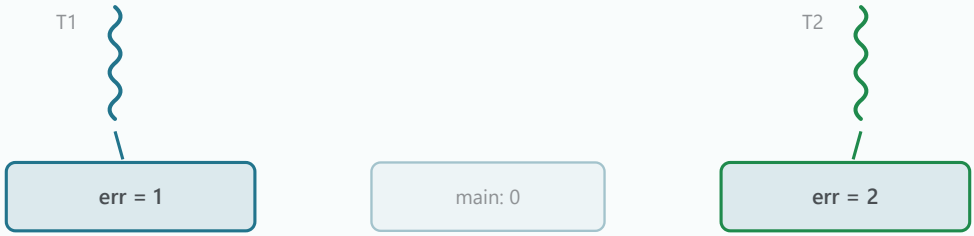
`_Thread_local int err;` (C11) gives each thread a **separate copy**; T2 does not see what T1 wrote.

Step 2 / 3

- Unlike a local variable, it lives **across** function calls; like a global, but private to the thread.
- C11 `_Thread_local`, C23/C++ `thread_local`, GCC `__thread`; Pthreads: `pthread_key_create`, `pthread_setspecific`.
- Typical uses: `errno`, a transaction ID, a random-number generator.

```
static ThreadLocal<Integer> id =
  ThreadLocal.withInitial(() -> 0);
```

Thread-Local Storage (TLS)



Output of Java `ThreadLocal` (JDK 21): `A: 1`, `B: 2`, `main: 0`; main sees the initial value.

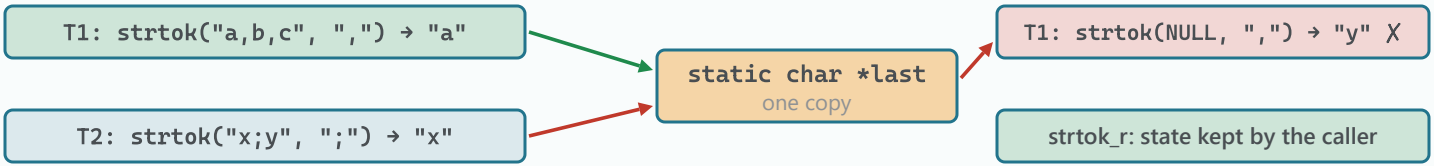
- Unlike a local variable, it lives **across** function calls; like a global, but private to the thread.
- C11 `_Thread_local`, C23/C++ `thread_local`, GCC `__thread`; Pthreads: `pthread_key_create`, `pthread_setspecific`.
- Typical uses: `errno`, a transaction ID, a random-number generator.

```
static ThreadLocal<Integer> id =
  ThreadLocal.withInitial(() -> 0);
```

Reentrant and Thread-Safe Code

- **Non-reentrant** library routines keep their internal state in static variables.
- Example: one thread puts a message in a shared buffer, a context switch occurs, another thread writes to the same buffer → **the first message is lost**.
- While updating its free list, `malloc` leaves the list temporarily inconsistent; a thread cutting in may get an invalid pointer. Today's `malloc` s use internal locks and per-thread arenas.
- **Thread-safe**: can be called from several threads at once (using locks, or stateless). **Reentrant**: uses no shared state.

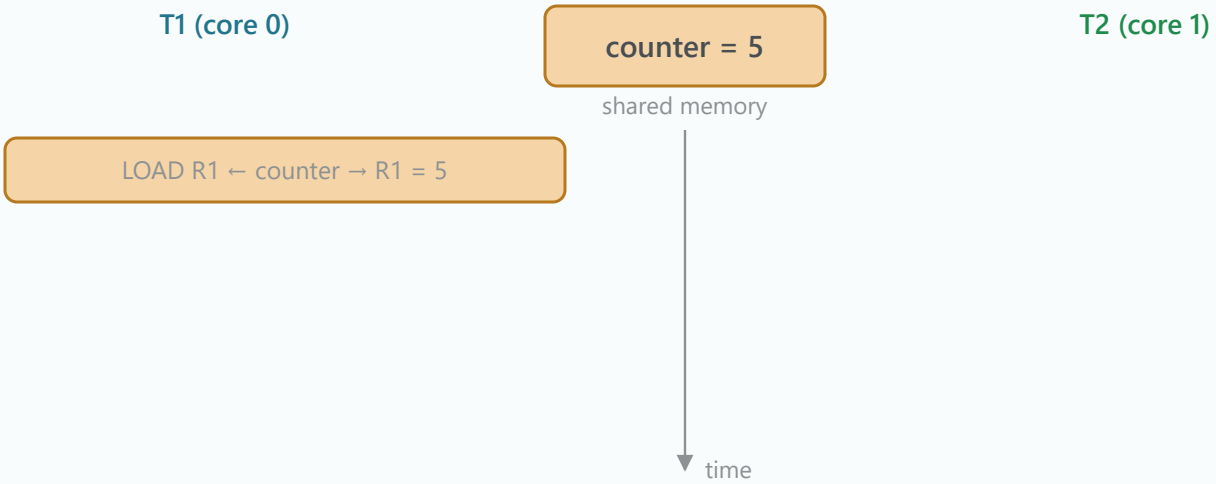
Unsafe	Safe version
`strtok`	`strtok_r`
`localtime`	`localtime_r`
`rand`	`rand_r` / local generator
`asctime` , `ctime`	`strftime`
Java `SimpleDateFormat`	`DateTimeFormatter`



Race Condition: counter++

```
LOAD  R, counter
ADD   R, 1
STORE counter, R
```

`counter++` is one line, but on the machine it is **three steps** (load, add, store). Initially `counter = 5`; two threads increment it once each. Expected result: **7**.

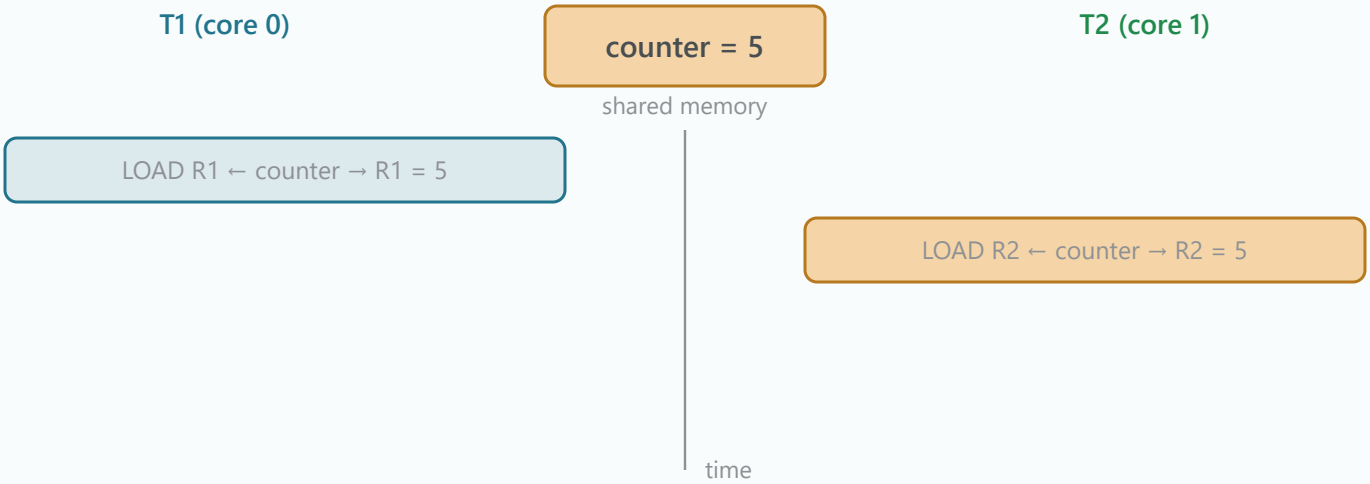


T1 loads the value into its register: R1 = 5.

Race Condition: counter++

```
LOAD  R, counter
ADD   R, 1
STORE counter, R
```

`counter++` is one line, but on the machine it is **three steps** (load, add, store). Initially `counter = 5`; two threads increment it once each. Expected result: **7**.

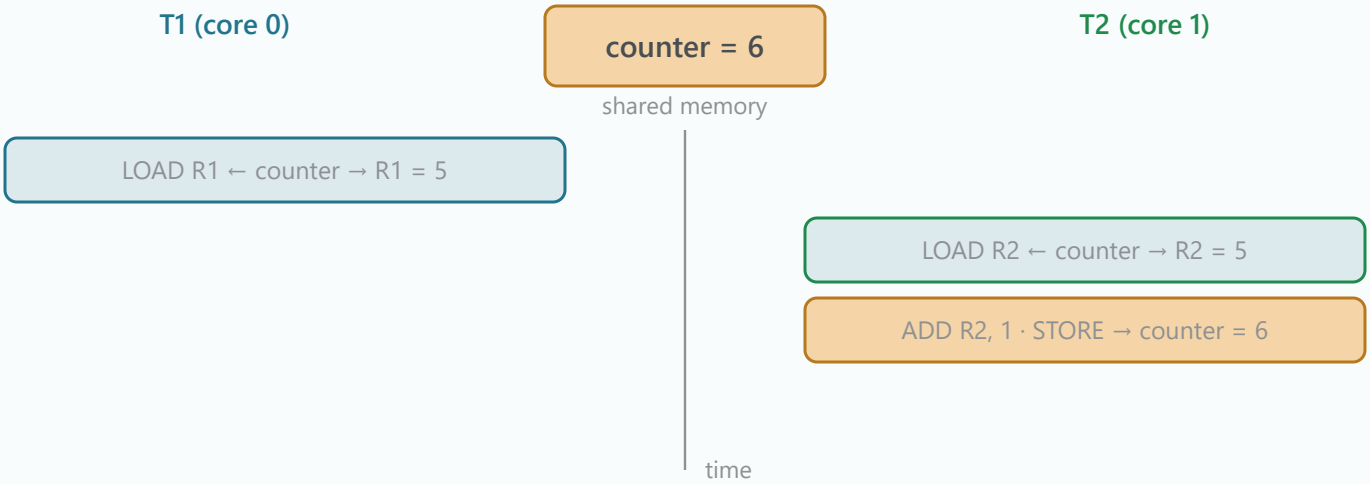


T2 cuts in (or runs at the same time on the other core) and also reads **5**.

Race Condition: counter++

```
LOAD  R, counter
ADD   R, 1
STORE counter, R
```

`counter++` is one line, but on the machine it is **three steps** (load, add, store). Initially `counter = 5`; two threads increment it once each. Expected result: **7**.

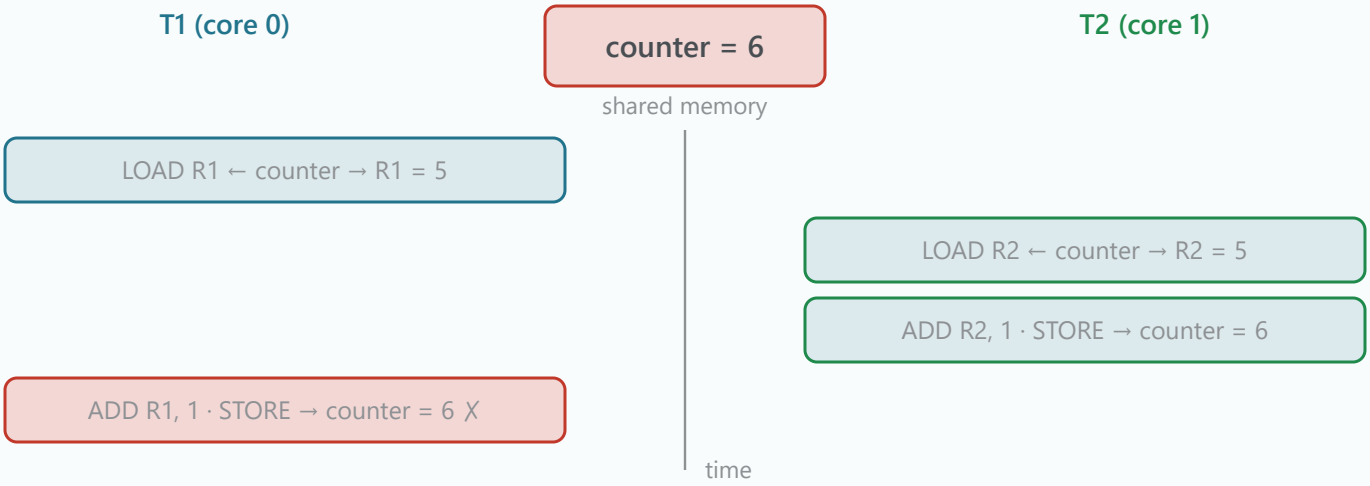


T2 increments and stores: counter = 6.

Race Condition: counter++

```
LOAD  R, counter
ADD   R, 1
STORE counter, R
```

`counter++` is one line, but on the machine it is **three steps** (load, add, store). Initially `counter = 5`; two threads increment it once each. Expected result: **7**.

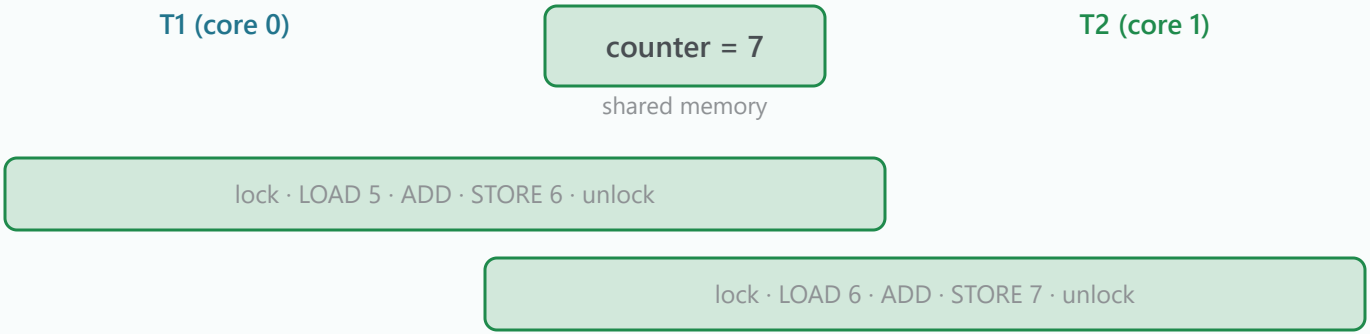


T1 continues from its old value: it stores 5 + 1 = 6. T2's increment is **lost** (a lost update).

Race Condition: counter++

```
LOAD  R, counter
ADD   R, 1
STORE counter, R
```

`counter++` is one line, but on the machine it is **three steps** (load, add, store). Initially `counter = 5`; two threads increment it once each. Expected result: **7**.



the critical section runs one at a time: result 7 ✓

C: pthread_mutex_lock / atomic_fetch_add · Java: synchronized / AtomicInteger

The fix: load-add-store must be **atomic**: a lock or an atomic operation.

Observing the Race (Java)

```java
public class Race {
  static int counter = 0;
  public static void main(String[] a) throws Exception {
    Runnable job = () -> {
      for (int i = 0; i < 1_000_000; i++) counter++;
    };
    Thread t1 = new Thread(job), t2 = new Thread(job);
    t1.start(); t2.start();
    t1.join();  t2.join();
    System.out.println("counter = " + counter);
  }
}
```

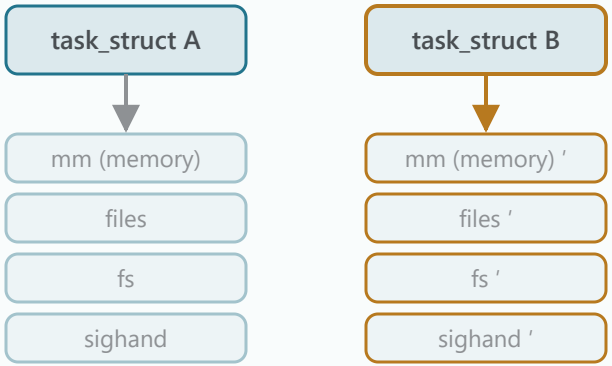
THREE SEPARATE RUNS (JDK 21)

```
counter = 1456465
counter = 1544210
counter = 1910165
```

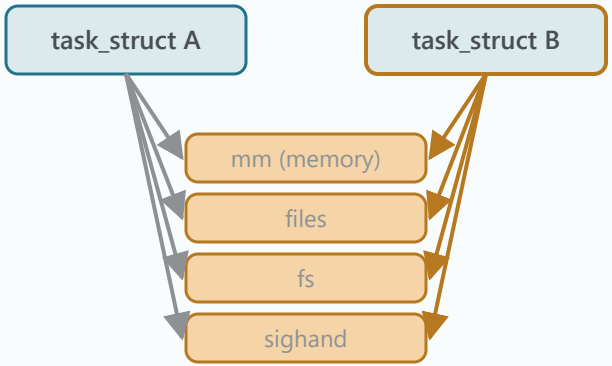
- Expected 2,000,000; every run gives a **different and smaller** result.
- The result is **nondeterministic**: it depends on scheduling, the number of cores and the JIT. That is why the output cannot be traced step by step.
- Fixes: a `synchronized` block, `AtomicInteger.incrementAndGet()`, `LongAdder`.
- Best of all: each thread counts in a local variable and the counts are combined at the end (the fork-join idea).

Threads on Linux: clone()

fork(): copy



clone(CLONE_VM | CLONE_FILES | ...): sharing



- Linux does not distinguish "process" from "thread": all are **tasks** (`task_struct`).
- The `clone()` flags decide what is shared; with no flags it behaves like `fork` .

- `pthread_create` ≈ `clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND | CLONE_THREAD | …)` .
- `CLONE_THREAD` : the same thread group; `getpid()` is the same for all (TGID), `gettid()` differs.

SUMMARY

Advantages and Disadvantages

Advantages

+ **Responsiveness:** when one thread blocks, another handles user I/O.
+ **Resource sharing:** memory, open files, sockets.
+ **Speed:** creation and switching are cheaper than for processes.
+ **Hardware parallelism:** one process uses many cores.
+ Parallelism even with blocking calls (web server), an easy programming model.

Disadvantages

– **Synchronization:** access to shared data must be controlled; risk of races and deadlock.
– **Dependence:** threads of one process are not independent; if one crashes, the process crashes.
– **No memory protection:** one thread can overwrite another's stack.
– Debugging and testing are hard; bugs depend on timing.

Choose a **process** when isolation is needed (browser tabs, security boundaries), a **thread** when you need tight data sharing and low cost. Chrome uses both.

Review Questions (1/2)

1. Which of register values, heap, global variables and stack are shared among the threads of a process?
2. Is there concurrency on a single-core system? Is there parallelism?
3. 80% of a program can be parallelized. What is the speedup on 8 cores, and what is the theoretical upper limit?
4. In the many-to-one model, what happens if a thread blocks in `read()`? Why can it not benefit from multiple cores?

▶ **Answers**

SUMMARY

Review Questions (2/2)

5. Two threads each increment `counter`, initially 0, once. What final values are possible?
6. Give two advantages of a thread pool over creating new threads.
7. What is the difference between deferred and asynchronous cancellation? Why is deferred preferred?
8. In POSIX, how many threads does the child have after `fork()` in a three-threaded process?

▶ **Answers**

References

- A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10th ed., Wiley, 2018 — Chapter 4: Threads & Concurrency.
- A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4th ed. (2014) / 5th ed. (2023), Pearson — Section 2.2: Threads.
- R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), ostep.org — “Concurrency: An Introduction”, “Thread API”.
- POSIX.1-2017 (IEEE Std 1003.1); the `pthreads(7)` and `clone(2)` Linux man pages.
- JEP 444: Virtual Threads (Java 21); the OpenMP 5.2 specification.

The End

Chapter 4: Threads

Previous: Chapter 3: Processes · Next: Chapter 5: CPU Scheduling