Chapter 3: Processes

Operating Systems

The process concept · Memory layout · Creation: fork / exec / wait · Termination, zombies, orphans · Process states · PCB · Scheduling queues · Context switch · Shared memory · Message passing · Pipes and sockets · Multitasking on mobile

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_03_Processes.html

Learning Outcomes

Process model
Explain the difference between a program and a process, and the memory layout of a process.

Life cycle
Describe process states, transitions and the contents of the PCB.

UNIX API
Trace small programs that use `fork()`, `exec()`, `wait()`, `exit()` and `pipe()`.

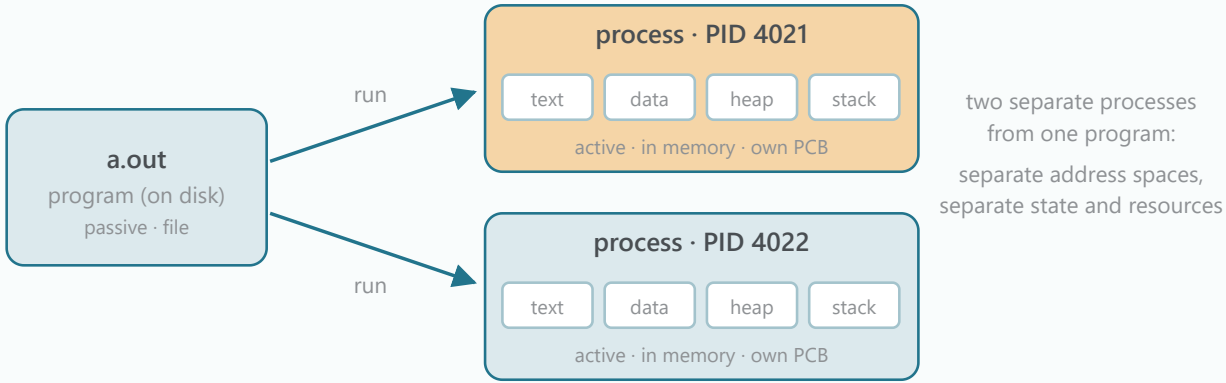
Scheduling machinery
Explain the ready/wait queues and the cost of a context switch.

IPC
Compare shared memory, message passing, pipes and sockets.

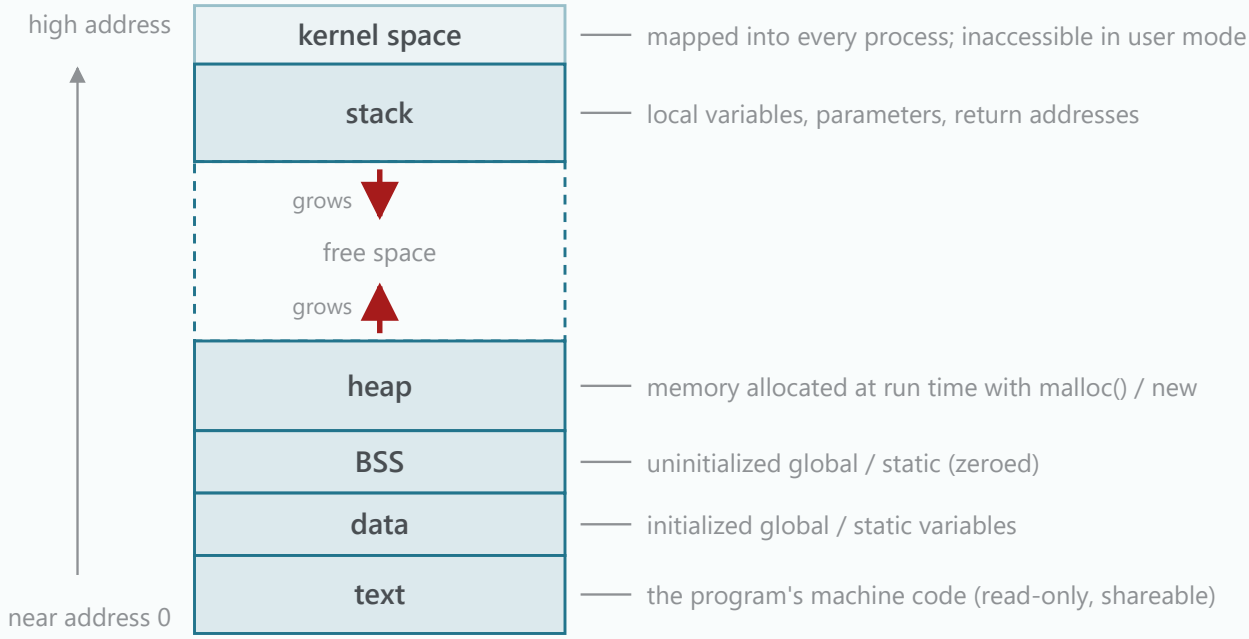
Mobile multitasking
Explain why and how iOS and Android restrict background processes.

What Is a Process?

- **Process:** an instance of a program in execution.
- **A program is passive** (an executable file on disk); **a process is active** (program counter, registers, memory, open files). A program becomes a process when it is loaded into memory.
- One program can give rise to **several processes** (`vim` in two terminals); each has its own address space and resources, though the text section can be shared.
- A single-threaded process executes its instructions **sequentially**; a multithreaded process can run in parallel on many cores.



Process Layout in Memory



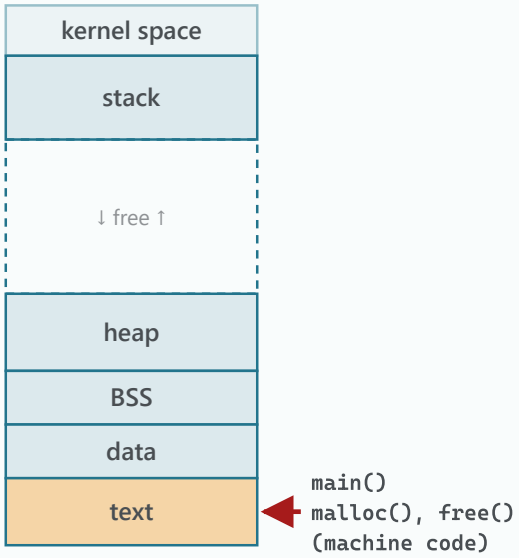
The stack grows down from high addresses, the heap grows up. The program counter is not in memory but is **a register in the processor**; it marks the process's "current position". In a real layout ASLR shifts the addresses randomly on every run.

Where Do Variables Live?

```c
#include <stdlib.h>
int counter = 5;  /* data */
int table[1000];  /* bss */

int main(int argc, char *argv[]) {
    int local = 3;
    static int count;
    int *p = malloc(4 * sizeof *p);
    p[0] = local + counter;
    free(p);
    return 0;
}
```

On Linux, `size a.out` shows the text/data/BSS sizes and `cat /proc/PID/maps` the regions of a running process.



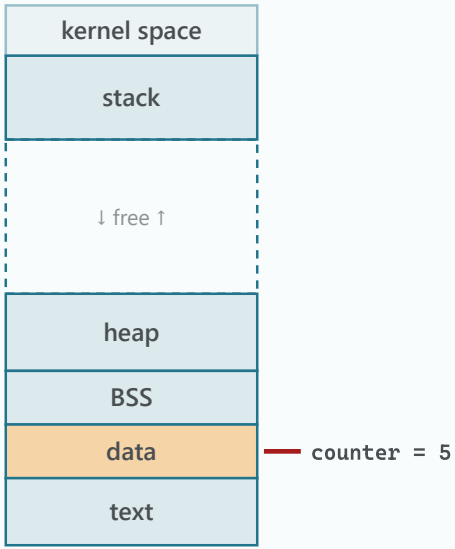
Text: the compiled instructions of `main`; read-only and shareable among processes of the same program.

Where Do Variables Live?

```c
#include <stdlib.h>
int counter = 5;  /* data */
int table[1000];  /* bss */

int main(int argc, char *argv[]) {
    int local = 3;
    static int count;
    int *p = malloc(4 * sizeof *p);
    p[0] = local + counter;
    free(p);
    return 0;
}
```

On Linux, `size a.out` shows the text/data/BSS sizes and `cat /proc/PID/maps` the regions of a running process.



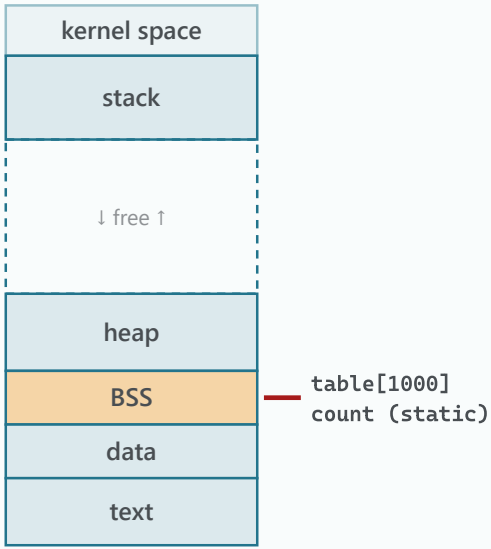
Data: the initialized global `counter`; its value is stored in the executable file.

Where Do Variables Live?

```c
#include <stdlib.h>
int counter = 5;  /* data */
int table[1000];  /* bss */

int main(int argc, char *argv[]) {
    int local = 3;
    static int count;
    int *p = malloc(4 * sizeof *p);
    p[0] = local + counter;
    free(p);
    return 0;
}
```

On Linux, `size a.out` shows the text/data/BSS sizes and `cat /proc/PID/maps` the regions of a running process.



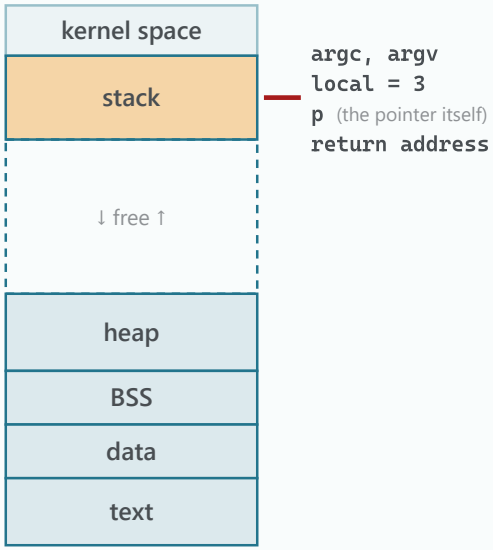
BSS: uninitialized global and `static` variables; the loader fills them with zeros and they take no space in the file.

Where Do Variables Live?

```c
#include <stdlib.h>
int counter = 5;  /* data */
int table[1000];  /* bss */

int main(int argc, char *argv[]) {
    int local = 3;
    static int count;
    int *p = malloc(4 * sizeof *p);
    p[0] = local + counter;
    free(p);
    return 0;
}
```

On Linux, `size a.out` shows the text/data/BSS sizes and `cat /proc/PID/maps` the regions of a running process.



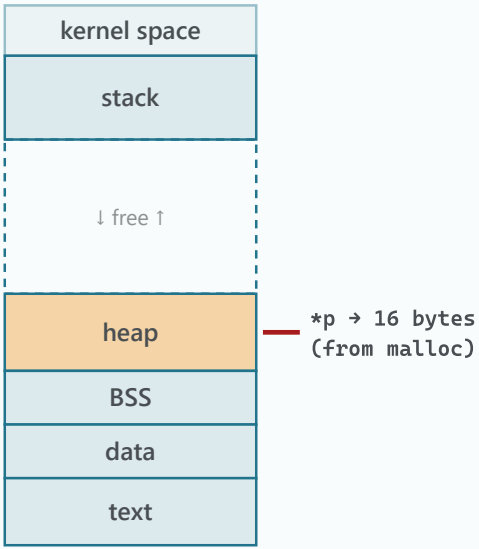
Stack: the call frame of `main`: parameters, local variables and the pointer `p` itself.

Where Do Variables Live?

```c
#include <stdlib.h>
int counter = 5;  /* data */
int table[1000];  /* bss */

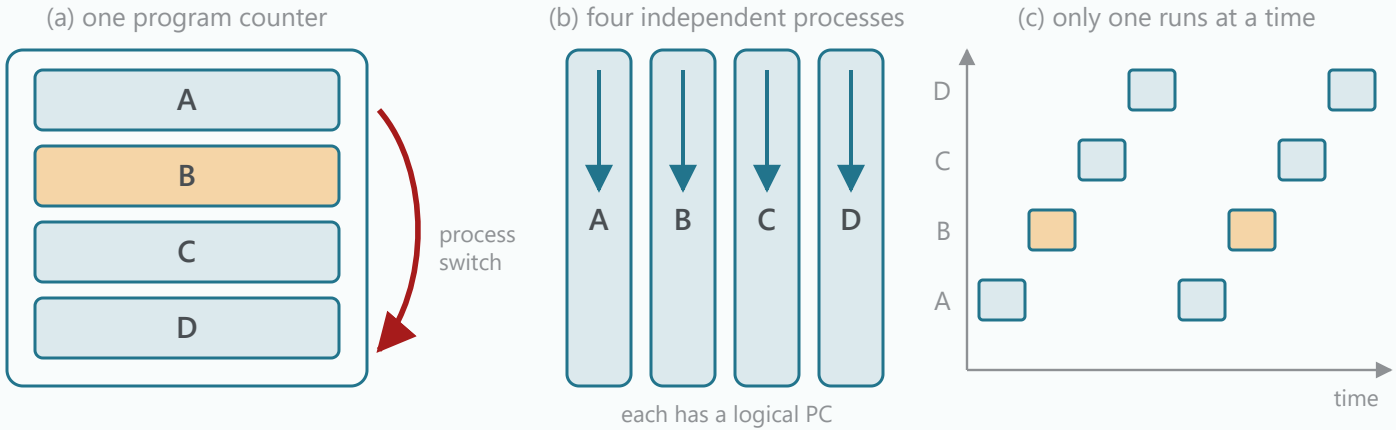
int main(int argc, char *argv[]) {
    int local = 3;
    static int count;
    int *p = malloc(4 * sizeof *p);
    p[0] = local + counter;
    free(p);
    return 0;
}
```

On Linux, `size a.out` shows the text/data/BSS sizes and `cat /proc/PID/maps` the regions of a running process.



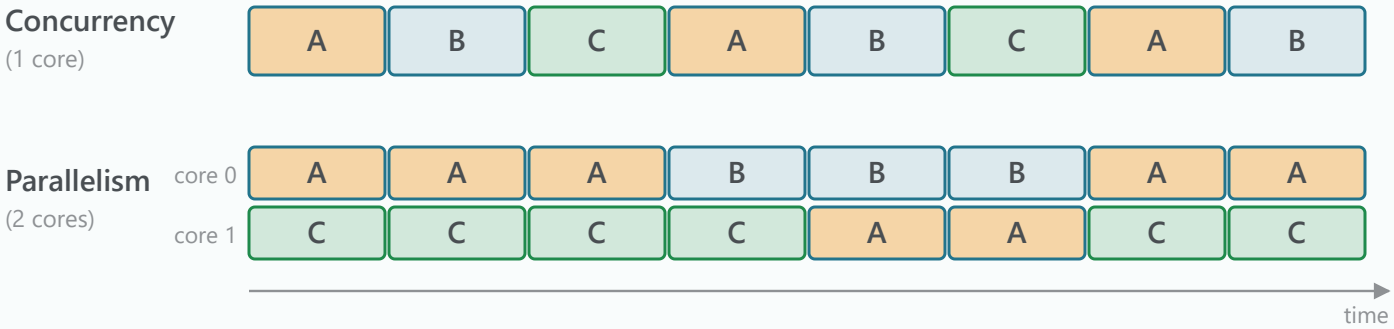
Heap: the 16 bytes allocated by `malloc`; `p` lives on the stack, but the area it points to is on the heap. It is returned with `free`.

Pseudoparallelism and Multiprogramming



- On a single core only **one** process executes at any instant; the scheduler switches rapidly among processes, creating **pseudoparallelism**.
- Switches happen with time slices of a few milliseconds (EEVDF on Linux uses a base slice of ~3 ms; the old CFS had a target latency of 6–24 ms); to the user all processes seem to run at once.
- Good for the user, but hard to manage: when a process will be interrupted is **unpredictable**, so no timing assumptions should be made.

Concurrency and Parallelism



Concurrency

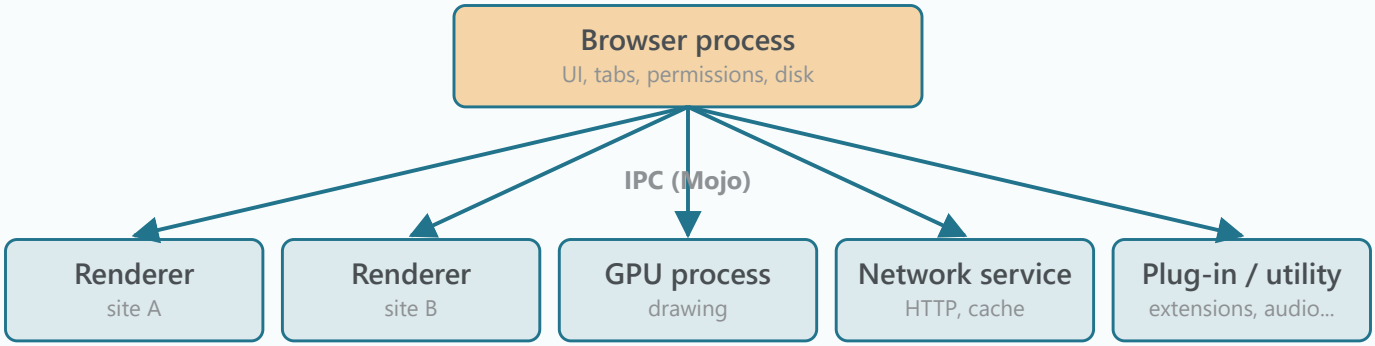
Several tasks *make progress* over the same time interval; on one core by interleaving.

Parallelism

Several tasks execute *at the same instant*; this needs multiple cores or processors.

As of 2026 even phones have 6–10 cores; still, the number of running processes (hundreds) far exceeds the core count, so multiprogramming and switching exist on every system.

Example: Chrome's Multiprocess Architecture



renderers run in a sandbox;
if one tab crashes, the browser and other tabs live on

Browser

Manages the UI, tabs, disk and permissions; the one privileged process.

Renderer

Handles HTML, CSS, JavaScript; with site isolation, one process per site.

GPU / network / utility

Drawing, networking and extensions in separate processes. Old plug-in support was removed: NPAPI in 2015, Flash at the end of 2020.

Process Creation

Events that create processes

1. System initialization (init/systemd, background services: *daemons*).
2. A running process making a process-creation system call.
3. A user starting a command or application.
4. Starting a batch job.

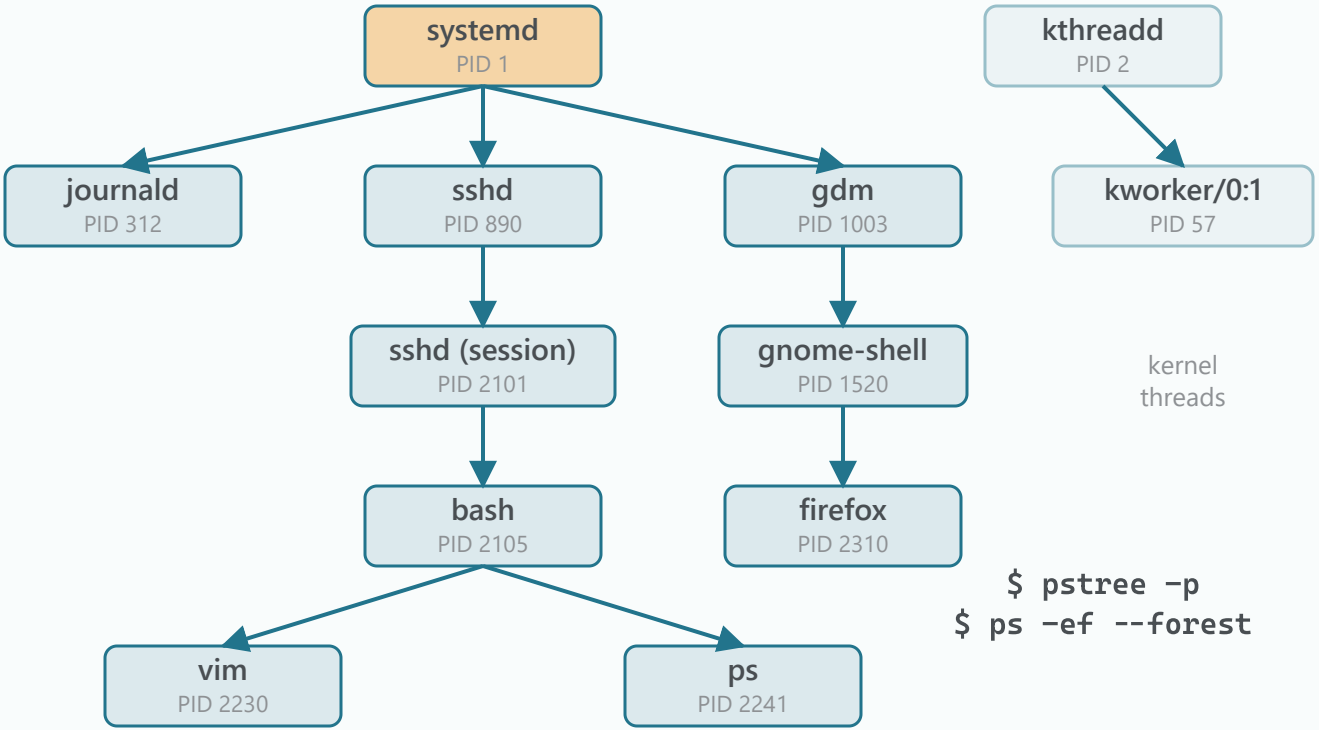
Technically the same thing happens every time: an existing process executes a **system call** that creates a new process.

System calls

System	Call
UNIX / Linux	`fork()` + `exec()`; also `posix_spawn()`, and `clone()` on Linux
Windows	`CreateProcess()`: creates the process and loads the program in one call

A parent process creates child processes; the children create their own children, forming a **process tree**. Each process carries a unique **PID**.

Process Tree



At boot Linux runs `systemd` (or another init) as PID 1; PID 2, `kthreadd`, is the parent of kernel threads. The PID values are examples.

Parent and Child

Resource sharing

The child may share all, some or none of the parent's resources. In UNIX the child **inherits** the open file descriptors.

Execution

Parent and child can run **concurrently**, or the parent can wait for the child to finish with `wait()`.

Address space

UNIX `fork()`: the child gets a **copy** of the parent's address space (it does not share it). It can then load a new program with `exec()`.

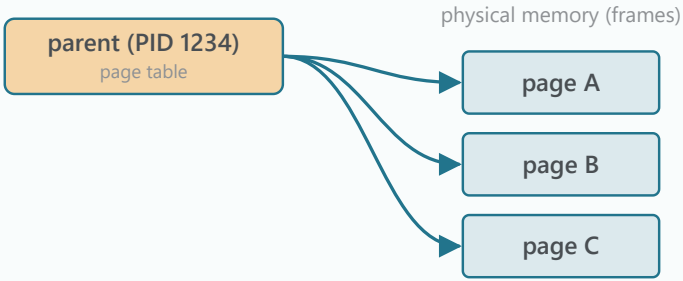
Identifiers

`getpid()` returns the process's own PID, `getppid()` its parent's.

After `fork()` the child's address space is **separate** from the parent's; for efficiency the physical pages are shared only until one is written (copy-on-write).

fork(): One Call, Two Returns

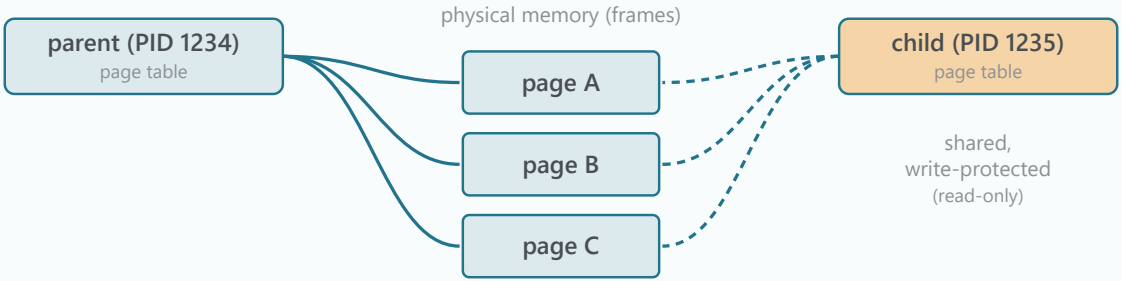
```
pid_t pid = fork();
if (pid < 0)        { perror("fork"); exit(1); }
else if (pid == 0)  { /* child: fork returned 0 */ }
else                { /* parent: pid = child's PID */ }
```



Start: the parent's page table points to physical pages A, B, C.

fork(): One Call, Two Returns

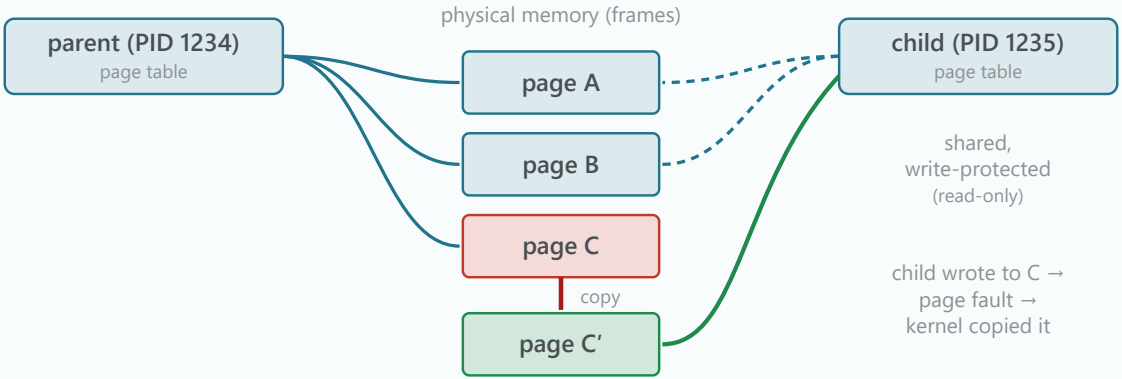
```
pid_t pid = fork();
if (pid < 0)        { perror("fork"); exit(1); }
else if (pid == 0)  { /* child: fork returned 0 */ }
else                { /* parent: pid = child's PID */ }
```



`fork()`: the kernel copies the PCB and the page table; the pages are **not copied** but mapped read-only into both. The parent gets the child's PID, the child gets 0; on error no child is created and only the parent gets −1.

fork(): One Call, Two Returns

```
pid_t pid = fork();
if (pid < 0)        { perror("fork"); exit(1); }
else if (pid == 0)  { /* child: fork returned 0 */ }
else                { /* parent: pid = child's PID */ }
```



When the child writes to page C a page fault occurs; the kernel copies only that page (C′). This is **copy-on-write (COW)**.

Code Trace: fork() + wait()

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int main(void) {
▶      int x = 10;
8      pid_t pid = fork();
9      if (pid < 0) { perror("fork"); exit(1);
   }
10     if (pid == 0) {
11         x = x + 5;
12         printf("child: x=%d\n", x);
13         return 0;
14     }
15     wait(NULL);
16     printf("parent: x=%d\n", x);
17     return 0;
18 }
```

VARIABLES

`parent.x = 10`

OUTPUT

parent · 1234
running

There is one process: the parent (PID 1234), `x = 10`.

Code Trace: fork() + wait()

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int main(void) {
7      int x = 10;
▶      pid_t pid = fork();
9      if (pid < 0) { perror("fork"); exit(1);
   }
10     if (pid == 0) {
11         x = x + 5;
12         printf("child: x=%d\n", x);
13         return 0;
14     }
15     wait(NULL);
16     printf("parent: x=%d\n", x);
17     return 0;
18 }
```

VARIABLES

`parent.x = 10` `parent.pid = 1235`

`child.x = 10` `child.pid = 0`

OUTPUT

parent · 1234
fork() → 1235

child · 1235
fork() → 0

`fork()` creates child 1235. In the parent `pid = 1235`, in the child `pid = 0`; there are two separate `x`s. (A failed fork returns −1: line 9 prints the error and exits.)

Code Trace: fork() + wait()

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int main(void) {
7      int x = 10;
8      pid_t pid = fork();
9      if (pid < 0) { perror("fork"); exit(1);
   }
10     if (pid == 0) {
11         x = x + 5;
12         printf("child: x=%d\n", x);
13         return 0;
14     }
▶      wait(NULL);
16     printf("parent: x=%d\n", x);
17     return 0;
18 }
```

VARIABLES

`parent.x = 10` `parent.pid = 1235`

`child.x = 10` `child.pid = 0`

OUTPUT

parent · 1234
wait() → waiting

child · 1235
ready

In the parent `pid > 0`: both `if`s are skipped, and `wait(NULL)` puts the parent in the **waiting** state until the child finishes.

Code Trace: fork() + wait()

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int main(void) {
7      int x = 10;
8      pid_t pid = fork();
9      if (pid < 0) { perror("fork"); exit(1);
   }
▶      if (pid == 0) {
11         x = x + 5;
12         printf("child: x=%d\n", x);
13         return 0;
14     }
15     wait(NULL);
16     printf("parent: x=%d\n", x);
17     return 0;
18 }
```

VARIABLES

`parent.x = 10` `parent.pid = 1235`

`child.x = 10` `child.pid = 0`

OUTPUT

parent · 1234
waiting

child · 1235
running

In the child `pid == 0` is true: it enters the child branch. (Had the child run first, the result would be the same.)

Code Trace: fork() + wait()

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int main(void) {
7      int x = 10;
8      pid_t pid = fork();
9      if (pid < 0) { perror("fork"); exit(1);
   }
10     if (pid == 0) {
▶          x = x + 5;
12         printf("child: x=%d\n", x);
13         return 0;
14     }
15     wait(NULL);
16     printf("parent: x=%d\n", x);
17     return 0;
18 }
```

VARIABLES

`parent.x = 10` `parent.pid = 1235`

`child.x = 15` `child.pid = 0`

OUTPUT



parent · 1234	child · 1235
waiting	running

The child changes its own copy: `x = 15`. The parent's `x` stays **10**.

Code Trace: fork() + wait()

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int main(void) {
7      int x = 10;
8      pid_t pid = fork();
9      if (pid < 0) { perror("fork"); exit(1);
   }
10     if (pid == 0) {
11         x = x + 5;
▶          printf("child: x=%d\n", x);
13         return 0;
14     }
15     wait(NULL);
16     printf("parent: x=%d\n", x);
17     return 0;
18 }
```

VARIABLES

`parent.x = 10` `parent.pid = 1235`

`child.x = 15` `child.pid = 0`

OUTPUT

```
child: x=15
```

parent · 1234	child · 1235
waiting	running

The child prints.

Code Trace: fork() + wait()

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int main(void) {
7      int x = 10;
8      pid_t pid = fork();
9      if (pid < 0) { perror("fork"); exit(1);
   }
10     if (pid == 0) {
11         x = x + 5;
12         printf("child: x=%d\n", x);
▶          return 0;
14     }
15     wait(NULL);
16     printf("parent: x=%d\n", x);
17     return 0;
18 }
```

VARIABLES

`parent.x = 10` `parent.pid = 1235`

`child.x = 15` `child.pid = 0`

OUTPUT

```
child: x=15
```

parent · 1234	child · 1235
woken (SIGCHLD)	zombie

The child exits with `return 0` (`exit(0)`); it is briefly a zombie, and the kernel wakes the parent.

Code Trace: fork() + wait()

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int main(void) {
7      int x = 10;
8      pid_t pid = fork();
9      if (pid < 0) { perror("fork"); exit(1);
   }
10     if (pid == 0) {
11         x = x + 5;
12         printf("child: x=%d\n", x);
13         return 0;
14     }
▶      wait(NULL);
16     printf("parent: x=%d\n", x);
17     return 0;
18 }
```

VARIABLES

`parent.x = 10` `parent.pid = 1235`

OUTPUT

```
child: x=15
```

parent · 1234	child · 1235
running	reaped

`wait` returns 1235; the child's entry is removed from the process table.

Code Trace: fork() + wait()

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int main(void) {
7      int x = 10;
8      pid_t pid = fork();
9      if (pid < 0) { perror("fork"); exit(1);
   }
10     if (pid == 0) {
11         x = x + 5;
12         printf("child: x=%d\n", x);
13         return 0;
14     }
15     wait(NULL);
▶      printf("parent: x=%d\n", x);
17     return 0;
18 }
```

VARIABLES

`parent.x = 10` `parent.pid = 1235`

OUTPUT

```
child: x=15
parent: x=10
```

parent · 1234	child · 1235
running	reaped

The parent prints: its own `x` is still 10.

Code Trace: fork() + wait()

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int main(void) {
7      int x = 10;
8      pid_t pid = fork();
9      if (pid < 0) { perror("fork"); exit(1);
   }
10     if (pid == 0) {
11         x = x + 5;
12         printf("child: x=%d\n", x);
13         return 0;
14     }
15     wait(NULL);
16     printf("parent: x=%d\n", x);
▶      return 0;
18 }
```

VARIABLES

`parent.x = 10` `parent.pid = 1235`

OUTPUT

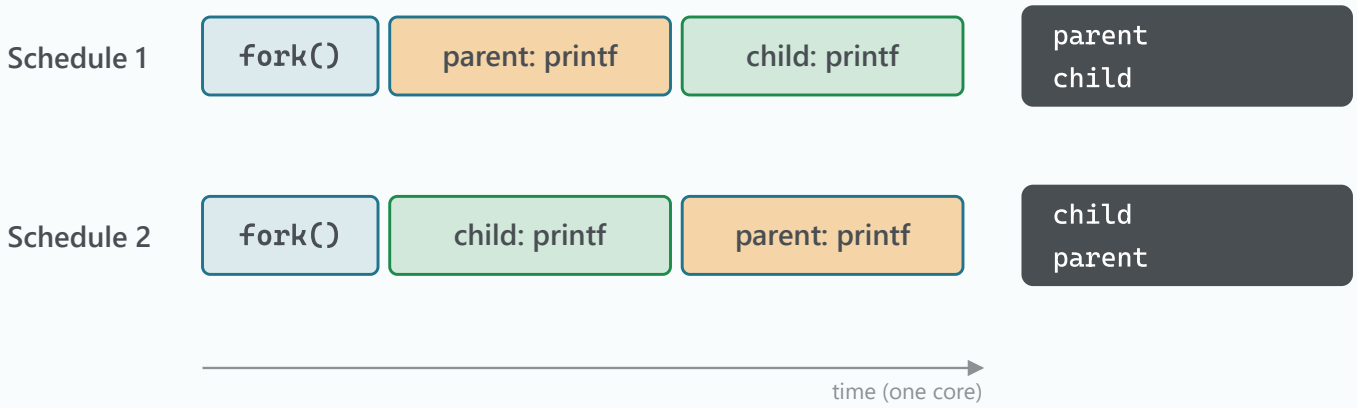
```
child: x=15
parent: x=10
```

parent · 1234	child · 1235
terminated	reaped

The parent terminates too. Thanks to `wait()`, the output order is **the same on every run**.

Without wait() the Output Order Is Undefined

```c
pid_t pid = fork();
if (pid == 0) printf("child\n");
else          printf("parent\n");
```



The **scheduler** decides which process runs first after fork; on multiple cores both may even run at once. If the program's correctness depends on this order, use `wait()` (or another form of synchronization).

The exec() Family



Call	Arguments	PATH search	Environment
`execl` / `execv`	list / array	no	inherited
`execlp` / `execvp`	list / array	yes	inherited
`execle` / `execve`	list / array	no	passed explicitly

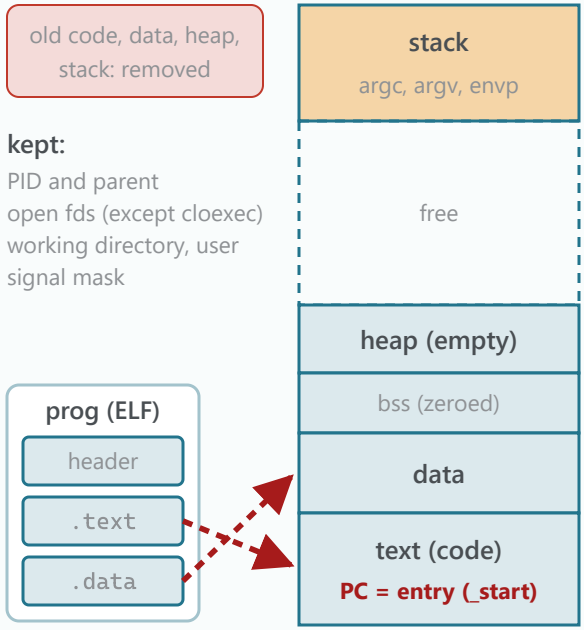
They all end up in the `execve()` system call. A successful exec **never returns**; if it returns, there was an error.

What Does exec Do in the Kernel?

1. Open the file; check permission and format (ELF header or `#!` line).

   If the file is missing or not executable, `exec` returns −1 and the old program carries on.

Step 1 / 6



mapping is lazy: a page is read on first access

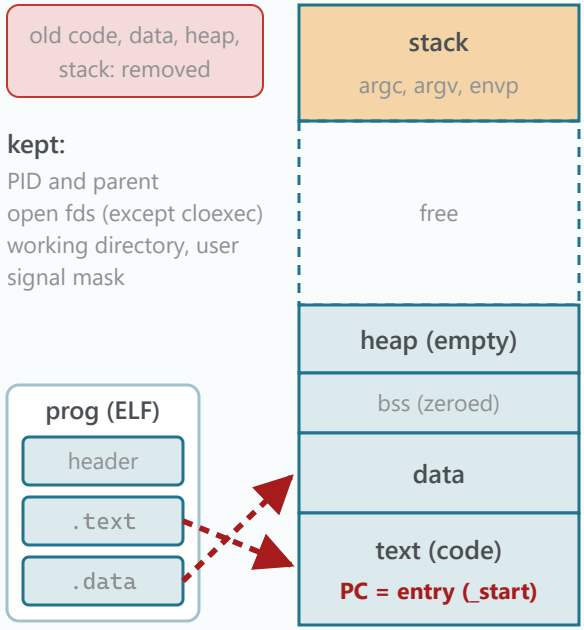
A program on disk is a passive file; exec turns it into the address space of a running process. On Linux the whole exec family ends up in `execve()`.

What Does exec Do in the Kernel?

1. Open the file; check permission and format (ELF header or `#!` line).

2. Release the old address space.

   The code, data, heap and stack mappings are removed; other threads in the process are terminated. From here on there is no program to return to.



mapping is lazy: a page is read on first access

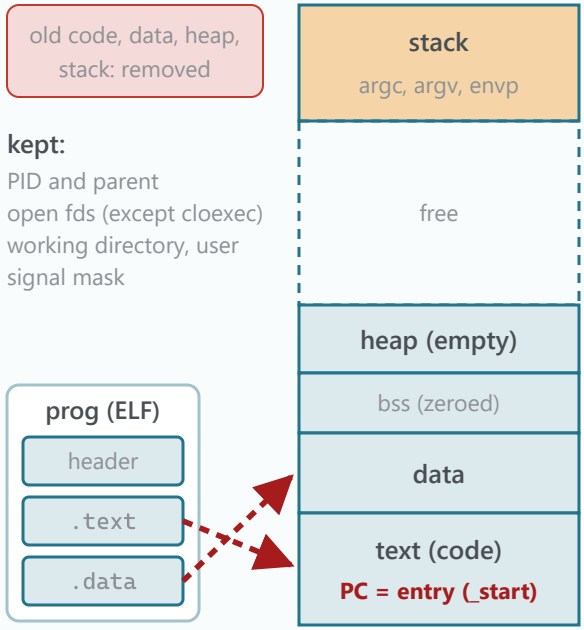
A program on disk is a passive file; exec turns it into the address space of a running process. On Linux the whole exec family ends up in `execve()`.

What Does exec Do in the Kernel?

1. Open the file; check permission and format (ELF header or `#!` line).
2. Release the old address space.
3. Close close-on-exec fds, reset signal handlers.

   Caught signals revert to the default action, since the handler code is no longer in memory; ignored signals stay ignored.

Step 3 / 6



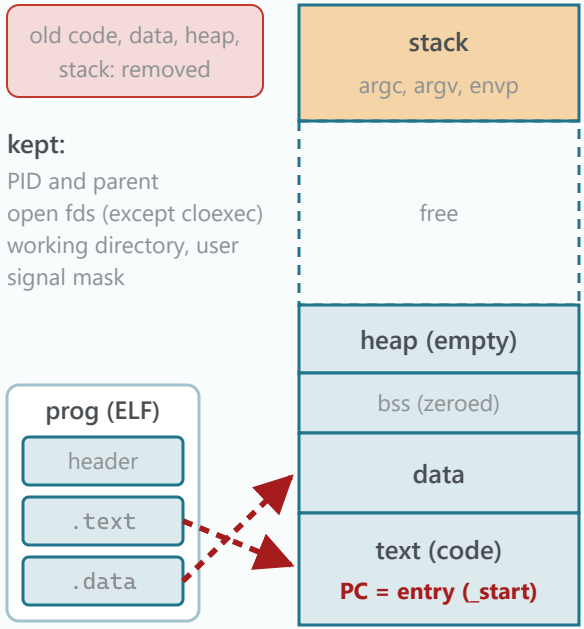
mapping is lazy: a page is read on first access

A program on disk is a passive file; exec turns it into the address space of a running process. On Linux the whole exec family ends up in `execve()`.

What Does exec Do in the Kernel?

1. Open the file; check permission and format (ELF header or `#!` line).
2. Release the old address space.
3. Close close-on-exec fds, reset signal handlers.
4. Map the code and data sections from the file.

   Nothing is copied: the first use of a page causes a page fault and that page is brought in from disk (lazy loading).



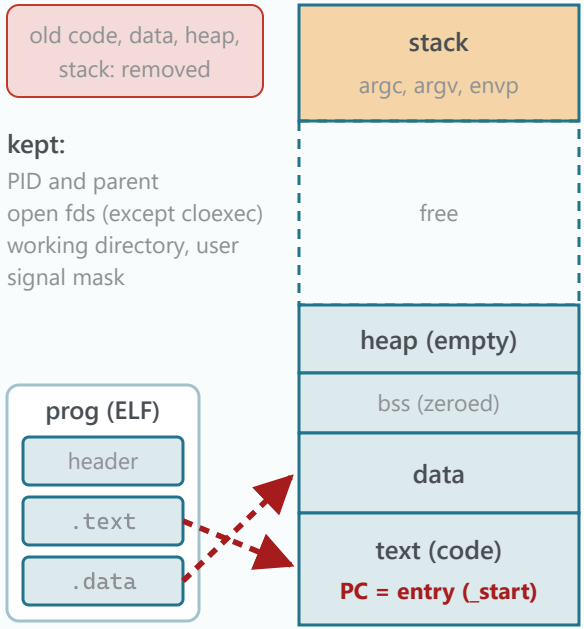
mapping is lazy: a page is read on first access

A program on disk is a passive file; exec turns it into the address space of a running process. On Linux the whole exec family ends up in `execve()`.

What Does exec Do in the Kernel?

1. Open the file; check permission and format (ELF header or `#!` line).
2. Release the old address space.
3. Close close-on-exec fds, reset signal handlers.
4. Map the code and data sections from the file.
5. Allocate zeroed memory for bss, heap and stack.
   The argv and envp strings, the pointer arrays to them and argc are placed at the top of the stack.

Step 5 / 6



mapping is lazy: a page is read on first access

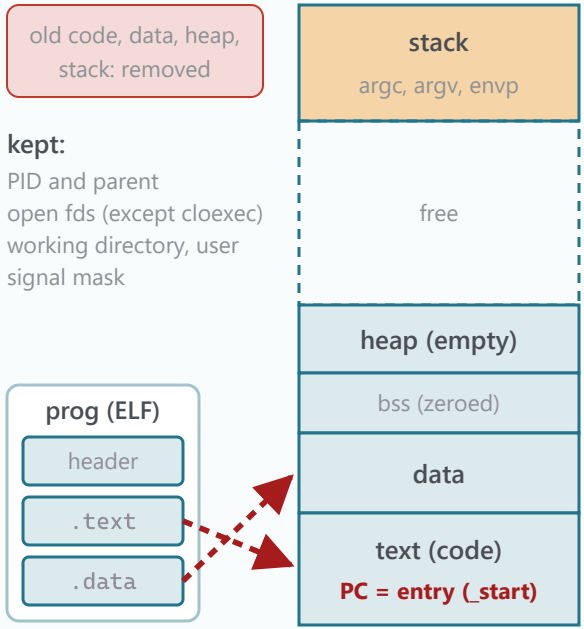
A program on disk is a passive file; exec turns it into the address space of a running process. On Linux the whole exec family ends up in `execve()`.

What Does exec Do in the Kernel?

1. Open the file; check permission and format (ELF header or `#!` line).
2. Release the old address space.
3. Close close-on-exec fds, reset signal handlers.
4. Map the code and data sections from the file.
5. Allocate zeroed memory for bss, heap and stack.
6. PC = entry point; return to user mode.

`_start` sets up the C runtime and calls `main(argc, argv, envp)`. The PID never changed.

Step 6 / 6



mapping is lazy: a page is read on first access

A program on disk is a passive file; exec turns it into the address space of a running process. On Linux the whole exec family ends up in `execve()`.

Code Trace: fork + exec + waitpid

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int main(void) {
▶    pid_t pid = fork();
8    if (pid < 0) { perror("fork"); exit(1); }
9    if (pid == 0) {
10     execlp("ls", "ls", "-l", NULL);
11     perror("execlp");  /* error */
12     exit(1);
13   }
14   int st;
15   if (waitpid(pid, &st, 0) < 0) {
   perror("waitpid"); exit(1); }
16   if (WIFEXITED(st)) printf("done: %d\n",
   WEXITSTATUS(st));
17   return 0;
18 }
```

VARIABLES

`parent.pid = 1235` `child.pid = 0`

OUTPUT

parent · 1234
fork() → 1235

child · 1235
fork() → 0

`fork()` : two processes are running the same `a.out` image. Had `pid < 0`, line 8 would print the error and exit.

Step 1 / 6

Code Trace: fork + exec + waitpid

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void) {
  pid_t pid = fork();
  if (pid < 0) { perror("fork"); exit(1); }
  if (pid == 0) {
    execlp("ls", "ls", "-l", NULL);
    perror("execlp");  /* error */
    exit(1);
  }
  int st;
▶ if (waitpid(pid, &st, 0) < 0) { perror("waitpid"); exit(1); }
  if (WIFEXITED(st)) printf("done: %d\n", WEXITSTATUS(st));
  return 0;
}
```

VARIABLES

`parent.pid = 1235` `child.pid = 0`

OUTPUT





The parent skips the child branch and blocks in `waitpid(1235, ...)`.

Code Trace: fork + exec + waitpid

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void) {
  pid_t pid = fork();
  if (pid < 0) { perror("fork"); exit(1); }
  if (pid == 0) {
    execlp("ls", "ls", "-l", NULL);
    perror("execlp");  /* error */
    exit(1);
  }
  int st;
  if (waitpid(pid, &st, 0) < 0) { perror("waitpid"); exit(1); }
  if (WIFEXITED(st)) printf("done: %d\n", WEXITSTATUS(st));
  return 0;
}
```

VARIABLES

`parent.pid = 1235` `child = ls -l`

OUTPUT





The child calls `execlp` : `ls` is found on PATH and the address space is **replaced** by `/usr/bin/ls` . The PID is still 1235.

Code Trace: fork + exec + waitpid

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <unistd.h>
 4 #include <sys/wait.h>
 5
 6 int main(void) {
 7   pid_t pid = fork();
 8   if (pid < 0) { perror("fork"); exit(1); }
 9   if (pid == 0) {
▶      execlp("ls", "ls", "-l", NULL);
11     perror("execlp");  /* error */
12     exit(1);
13   }
14   int st;
15   if (waitpid(pid, &st, 0) < 0) {
   perror("waitpid"); exit(1); }
16   if (WIFEXITED(st)) printf("done: %d\n",
   WEXITSTATUS(st));
17   return 0;
18 }
```

VARIABLES

`parent.pid = 1235` `child = ls -l`

OUTPUT

```
(ls -l output)
```

parent · 1234	child · 1235
waiting	ls running

The child is now the `ls` program; lines 11–12 **never run** (exec succeeded). It writes the listing to the inherited stdout.

Code Trace: fork + exec + waitpid

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int main(void) {
7    pid_t pid = fork();
8    if (pid < 0) { perror("fork"); exit(1); }
9    if (pid == 0) {
10     execlp("ls", "ls", "-l", NULL);
11     perror("execlp");  /* error */
12     exit(1);
13   }
14   int st;
▶    if (waitpid(pid, &st, 0) < 0) {
perror("waitpid"); exit(1); }
16   if (WIFEXITED(st)) printf("done: %d\n",
WEXITSTATUS(st));
17   return 0;
18 }
```

VARIABLES

`parent.pid = 1235` `st = 0 (encoded)`

OUTPUT

```
(ls -l output)
```

parent · 1234
woken

child · 1235
output: 0

`ls` ends with `exit(0)`; `waitpid` returns the child's PID (no error) and `st` carries the exit status.

Code Trace: fork + exec + waitpid

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void) {
  pid_t pid = fork();
  if (pid < 0) { perror("fork"); exit(1); }
  if (pid == 0) {
    execlp("ls", "ls", "-l", NULL);
    perror("execlp");  /* error */
    exit(1);
  }
  int st;
  if (waitpid(pid, &st, 0) < 0) { perror("waitpid"); exit(1); }
  if (WIFEXITED(st)) printf("done: %d\n", WEXITSTATUS(st));
  return 0;
}
```

VARIABLES

`parent.pid = 1235` `WEXITSTATUS = 0`

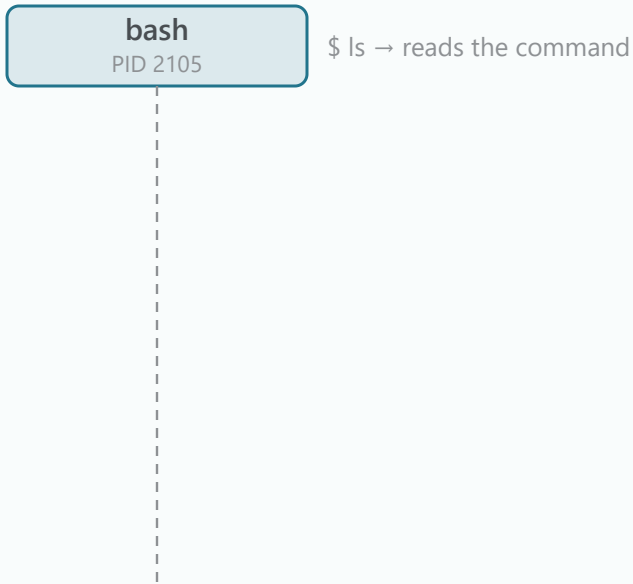
OUTPUT

```
(ls -l output)
done: 0
```

parent · 1234	child · 1235
running	reaped

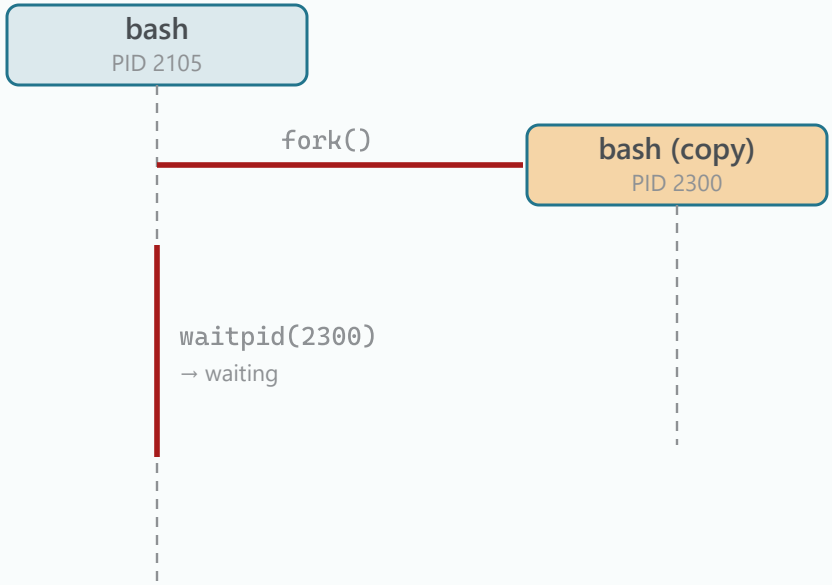
The child exited normally (`WIFEXITED` is true): `WEXITSTATUS(st)` = 0 is printed.

The Shell Loop: fork → exec → wait



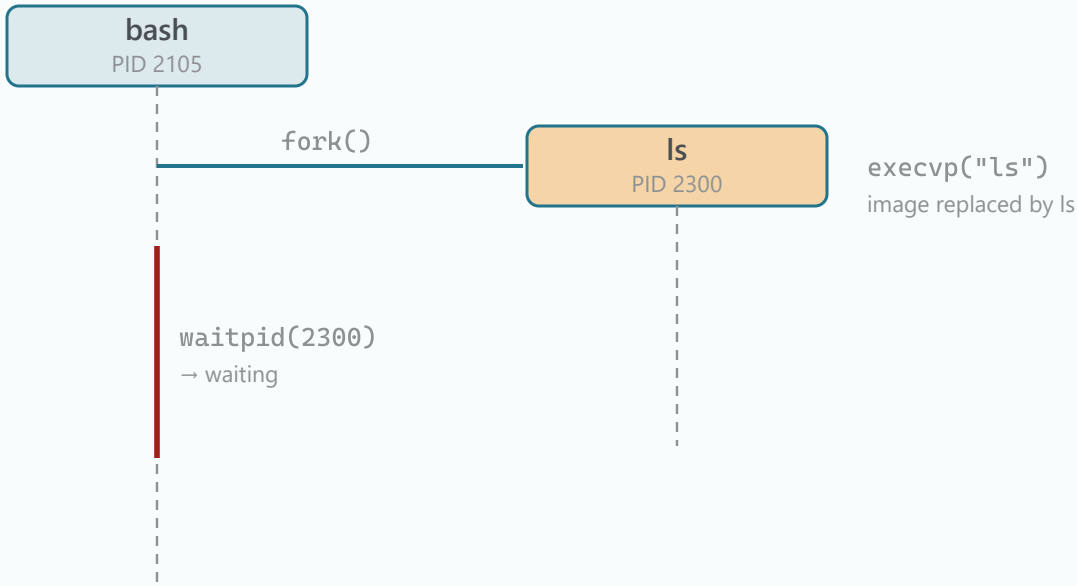
The shell shows a prompt and reads the `ls` command.

The Shell Loop: fork → exec → wait



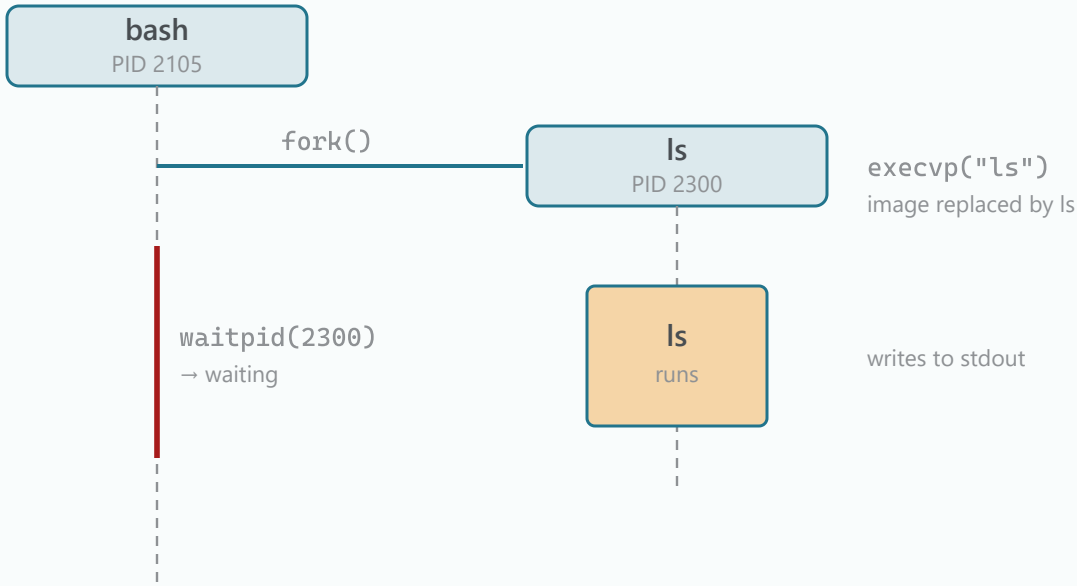
`fork()`: a copy of the shell (PID 2300) is created; the parent starts waiting with `waitpid`.

The Shell Loop: fork → exec → wait



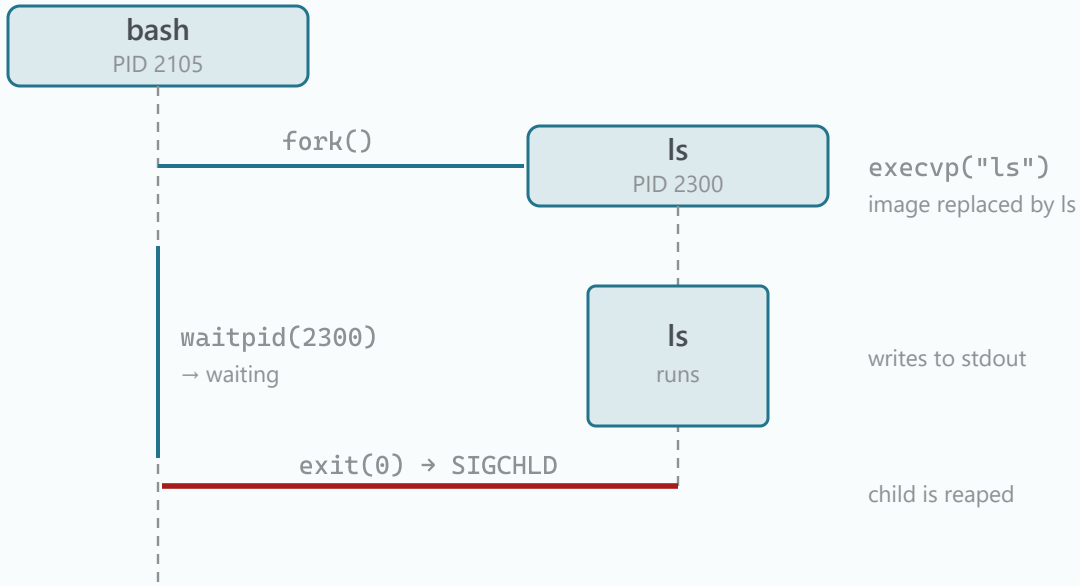
The child calls `execvp("ls", argv)`; its image is replaced by `ls`.

The Shell Loop: fork → exec → wait



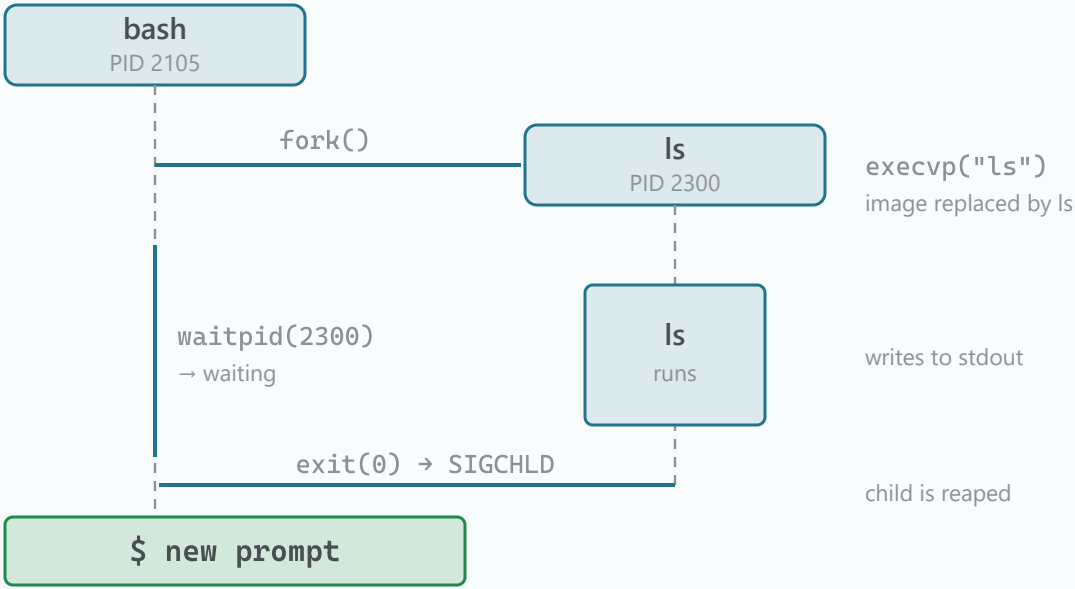
`ls` runs and writes to the terminal stdout it inherited.

The Shell Loop: fork → exec → wait



`ls` exits; the kernel notifies the parent with SIGCHLD and `waitpid` reaps the child.

The Shell Loop: fork → exec → wait



The shell shows a new prompt. If the command ends with `&`, the shell continues without waiting (a background job).

Why Are fork and exec Separate? Output Redirection

```
1  int main(void) {
▶    pid_t pid = fork();
3    if (pid == 0) {    /* child */
4      close(1);        /* stdout freed */
5      int fd = open("count.txt",
6               O_WRONLY | O_CREAT | O_TRUNC, 0644);
7      /* lowest free number: fd == 1 */
8      execlp("wc", "wc", "-l", "notes.txt", NULL);
9      perror("execlp");
10     exit(1);
11   }
12   waitpid(pid, NULL, 0);
13   printf("done\n");
14   return 0;
15 }
```

VARIABLES

`parent.pid = 1500`

`child.pid = 1501`

OUTPUT

child's fd table (copy of the parent's)

0 → tty	1 → tty	2 → tty

parent: 0, 1, 2 → terminal (unchanged)

`fork()` : the child **copies** the parent's fd table; 0, 1, 2 are the terminal.

Step 1 / 6

This is how the shell does `> count.txt` : between fork and exec the child sets up its environment (fds, directory, priority). Windows's `CreateProcess` takes these as parameters.

Why Are fork and exec Separate? Output Redirection

```
1  int main(void) {
2    pid_t pid = fork();
3    if (pid == 0) {    /* child */
4      close(1);        /* stdout freed */
5      int fd = open("count.txt",
6               O_WRONLY | O_CREAT | O_TRUNC, 0644);
7      /* lowest free number: fd == 1 */
8      execlp("wc", "wc", "-l", "notes.txt", NULL);
9      perror("execlp");
10     exit(1);
11   }
▶  waitpid(pid, NULL, 0);
13   printf("done\n");
14   return 0;
15 }
```

VARIABLES

`parent.pid = 1500`

`child.pid = 1501`

OUTPUT

child's fd table

0 → tty	1 → tty	2 → tty

parent: 0, 1, 2 → terminal (unchanged)

The parent waits in `waitpid`; the next steps change only the child's table.

This is how the shell does `> count.txt`: between fork and exec the child sets up its environment (fds, directory, priority). Windows's `CreateProcess` takes these as parameters.

Why Are fork and exec Separate? Output Redirection

```
1  int main(void) {
2    pid_t pid = fork();
3    if (pid == 0) {    /* child */
▶      close(1);        /* stdout freed */
5      int fd = open("count.txt",
6               O_WRONLY | O_CREAT | O_TRUNC, 0644);
7      /* lowest free number: fd == 1 */
8      execlp("wc", "wc", "-l", "notes.txt", NULL);
9      perror("execlp");
10     exit(1);
11   }
12   waitpid(pid, NULL, 0);
13   printf("done\n");
14   return 0;
15 }
```

VARIABLES

`parent.pid = 1500`

`child.pid = 1501`

OUTPUT

child's fd table



parent: 0, 1, 2 → terminal (unchanged)

`close(1)` : entry 1 in the child's table is freed. The parent's stdout is still the terminal.

Step 3 / 6

This is how the shell does `> count.txt` : between fork and exec the child sets up its environment (fds, directory, priority). Windows's `CreateProcess` takes these as parameters.

Why Are fork and exec Separate? Output Redirection

```
1  int main(void) {
2    pid_t pid = fork();
3    if (pid == 0) {    /* child */
4      close(1);        /* stdout freed */
▶      int fd = open("count.txt",
▶               O_WRONLY | O_CREAT | O_TRUNC, 0644);
7      /* lowest free number: fd == 1 */
8      execlp("wc", "wc", "-l", "notes.txt", NULL);
9      perror("execlp");
10     exit(1);
11   }
12   waitpid(pid, NULL, 0);
13   printf("done\n");
14   return 0;
15 }
```

VARIABLES

`parent.pid = 1500`

`child.pid = 1501` `fd = 1`

OUTPUT

child's fd table

0 → tty	1 → count.txt	2 → tty

parent: 0, 1, 2 → terminal (unchanged)

`open` always returns the **lowest free** number: 1. The child's stdout is now the file.

This is how the shell does `> count.txt` : between fork and exec the child sets up its environment (fds, directory, priority). Windows's `CreateProcess` takes these as parameters.

Why Are fork and exec Separate? Output Redirection

```
1  int main(void) {
2    pid_t pid = fork();
3    if (pid == 0) {    /* child */
4      close(1);        /* stdout freed */
5      int fd = open("count.txt",
6               O_WRONLY | O_CREAT | O_TRUNC, 0644);
7      /* lowest free number: fd == 1 */
▶      execlp("wc", "wc", "-l", "notes.txt", NULL);
9      perror("execlp");
10     exit(1);
11   }
12   waitpid(pid, NULL, 0);
13   printf("done\n");
14   return 0;
15 }
```

VARIABLES

`parent.pid = 1500` `child = wc -l`

OUTPUT

child = wc · fd table kept

0 → tty	1 → count.txt	2 → tty

parent: 0, 1, 2 → terminal (unchanged)

`exec` replaces code and data but **keeps** the fd table: unaware, wc writes to fd 1, that is, to the file.

This is how the shell does `> count.txt`: between fork and exec the child sets up its environment (fds, directory, priority). Windows's `CreateProcess` takes these as parameters.

Why Are fork and exec Separate? Output Redirection

```
1  int main(void) {
2    pid_t pid = fork();
3    if (pid == 0) {    /* child */
4      close(1);        /* stdout freed */
5      int fd = open("count.txt",
6               O_WRONLY | O_CREAT | O_TRUNC, 0644);
7      /* lowest free number: fd == 1 */
8      execlp("wc", "wc", "-l", "notes.txt", NULL);
9      perror("execlp");
10     exit(1);
11   }
12   waitpid(pid, NULL, 0);
▶    printf("done\n");
14   return 0;
15 }
```

VARIABLES

`parent.pid = 1500`

OUTPUT

```
done
```

child done; file holds: 42 notes.txt

0 → tty	1 → count.txt	2 → tty

$ cat count.txt → 42 notes.txt

wc finishes and `waitpid` returns; the parent writes to the terminal: only **done** appears on screen.

Step 6 / 6

This is how the shell does `> count.txt`: between fork and exec the child sets up its environment (fds, directory, priority). Windows's `CreateProcess` takes these as parameters.

Process Termination

Cause	Kind	Example
Normal exit	voluntary	`return 0`, `exit(0)`; the compiler finished its job
Error exit	voluntary	file not found → `exit(1)`
Fatal error	involuntary	invalid memory access (SIGSEGV), divide by zero (SIGFPE)
Killed by another process	involuntary	`kill(pid, SIGTERM)`, `TerminateProcess()` on Windows

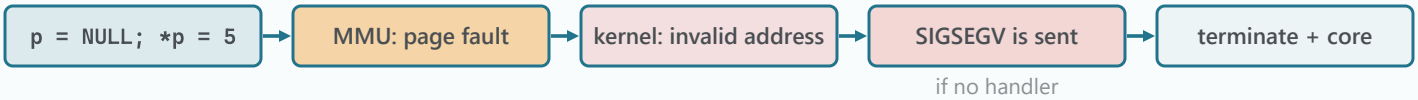
- `exit(status)`: the C library flushes buffers, runs the `atexit` handlers, then makes the `_exit` system call.
- `abort()`: the calling process sends itself SIGABRT and ends abnormally (a core dump may be produced). To terminate another process, use `kill()`.
- The kernel reclaims the process's memory, open files and other resources; the exit status is kept until the parent calls `wait()`.
- Some systems terminate the children when the parent terminates (*cascading termination*); in UNIX the children live on.

Fatal Errors

Error	Signal
Invalid/unauthorized memory access	SIGSEGV
Integer divide by zero	SIGFPE
Illegal instruction	SIGILL
Alignment / bus error	SIGBUS
Writing to a closed pipe	SIGPIPE

- The OS catches the hardware exception and sends the process the corresponding **signal**; if unhandled, the process terminates.
- The process is cleared from memory and its resources are released.
- A **core dump** can be taken: inspect the cause with `coredumpctl` or `gdb a.out core` (crash-dump analysis).

Leaking memory or not releasing resources is not fatal in itself; the process keeps running. If memory runs out, the Linux OOM killer may terminate a process with **SIGKILL**.



Termination by Signal: kill, pkill, killall

Signal	No	Meaning
SIGTERM	15	finish politely; the process can catch it and clean up (the default for `kill`)
SIGKILL	9	force termination; cannot be caught or ignored, no cleanup
SIGINT	2	`Ctrl+C` in the terminal
SIGTSTP	20	`Ctrl+Z` in the terminal (suspend); `fg` / `bg` resume it
SIGSTOP / SIGCONT	19 / 18	stop (cannot be caught) / continue

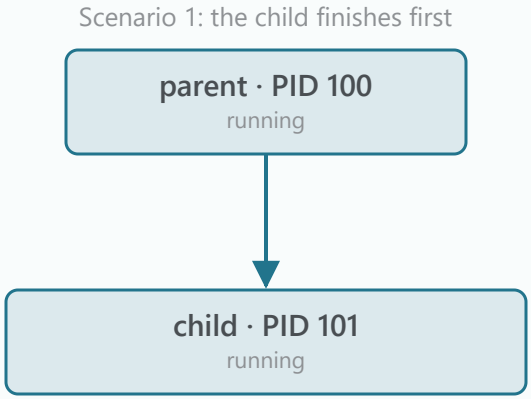
Numbers are for x86/ARM Linux; portable code uses the names, not the numbers.

```
$ ps -e | grep writer
 1234 ?  00:00:03 writer
$ kill 1234       # SIGTERM
$ kill -9 1234    # SIGKILL
$ pkill writer    # by name
$ killall writer
```

- SIGTERM first; SIGKILL if there is no response.
- Sending a signal requires being the same user or privileged (root).

PROCESS TERMINATION

Zombie and Orphan Processes



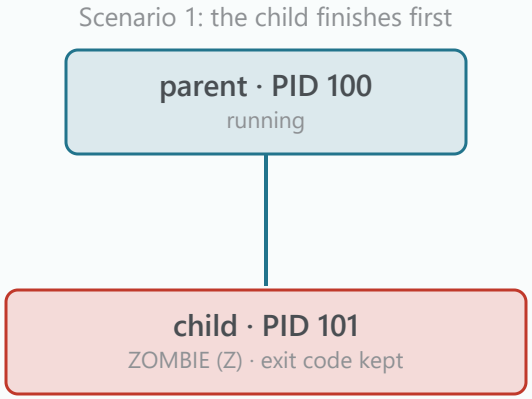
Parent (100) and child (101) are running.

Step 1 / 6

A zombie uses no resources but holds a PID; a long-lived server that never calls `wait` can pile up thousands of zombies and exhaust the PIDs. Fix: `wait` / `waitpid` or a SIGCHLD handler.

PROCESS TERMINATION

Zombie and Orphan Processes

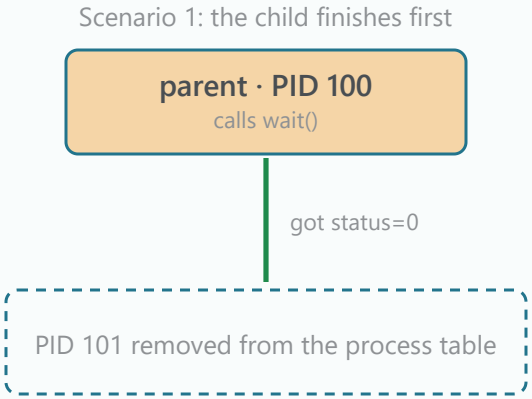


The child finished with `exit()`, but the parent has not called `wait()` yet: the child is a **zombie**. Its memory is freed; its PID and exit status wait in the process table (`Z` / <defunct> in `ps`).

Step 2 / 6

A zombie uses no resources but holds a PID; a long-lived server that never calls `wait` can pile up thousands of zombies and exhaust the PIDs. Fix: `wait` / `waitpid` or a SIGCHLD handler.

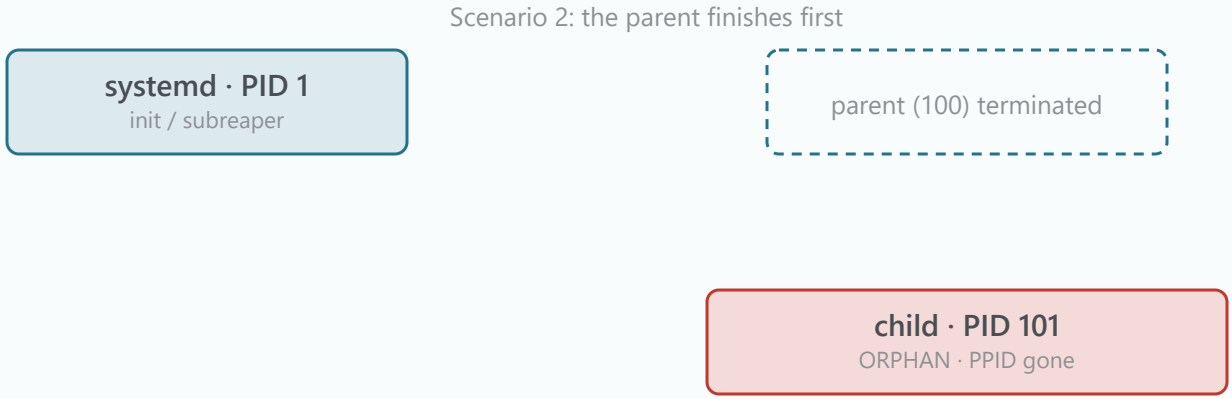
Zombie and Orphan Processes



The parent calls `wait()` and collects the exit status; the kernel removes the entry (**reaping**). No zombie remains.

A zombie uses no resources but holds a PID; a long-lived server that never calls `wait` can pile up thousands of zombies and exhaust the PIDs. Fix: `wait` / `waitpid` or a SIGCHLD handler.

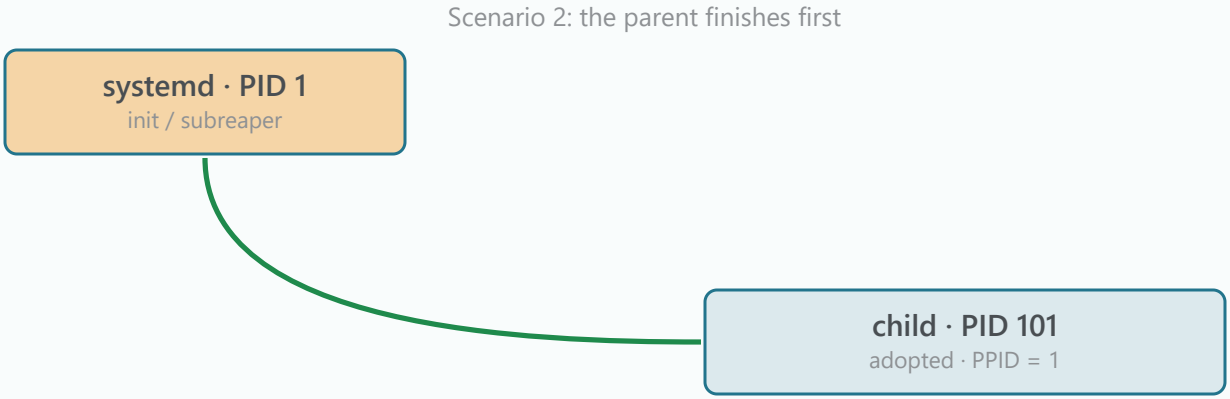
Zombie and Orphan Processes



Scenario 2: the parent (100) terminated before the child. The child (101) is still running: an **orphan** process.

A zombie uses no resources but holds a PID; a long-lived server that never calls `wait` can pile up thousands of zombies and exhaust the PIDs. Fix: `wait` / `waitpid` or a SIGCHLD handler.

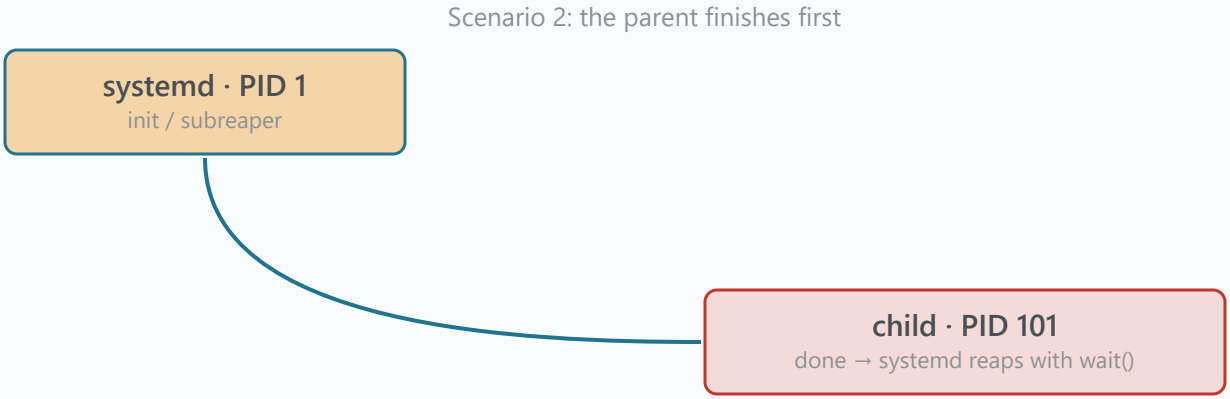
Zombie and Orphan Processes



The kernel reparents the orphan to `init` / `systemd` (PID 1) or a designated subreaper process; `getppid()` now returns 1.

A zombie uses no resources but holds a PID; a long-lived server that never calls `wait` can pile up thousands of zombies and exhaust the PIDs. Fix: `wait` / `waitpid` or a SIGCHLD handler.

Zombie and Orphan Processes

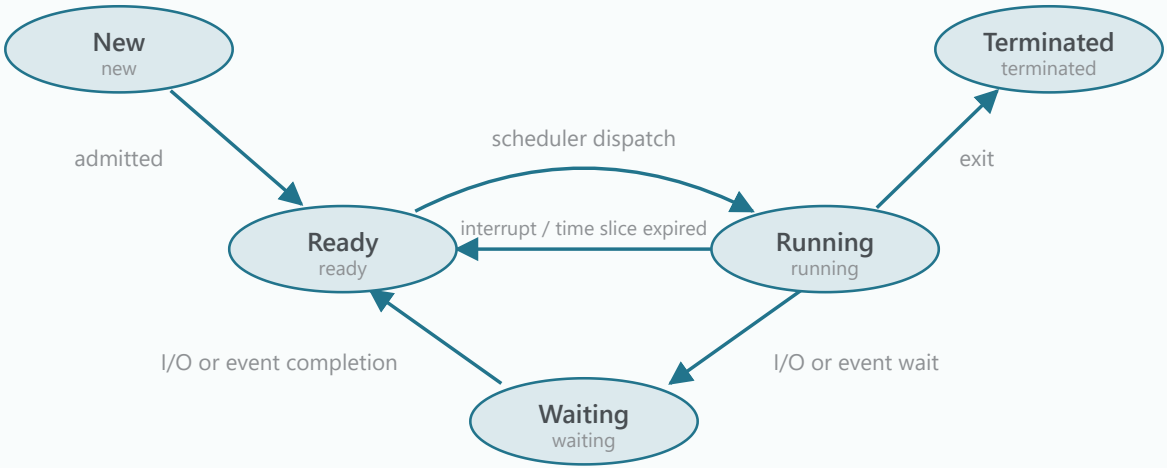


When the orphan finishes, its new parent reaps it with `wait()`, so zombies do not pile up.

A zombie uses no resources but holds a PID; a long-lived server that never calls `wait` can pile up thousands of zombies and exhaust the PIDs. Fix: `wait` / `waitpid` or a SIGCHLD handler.

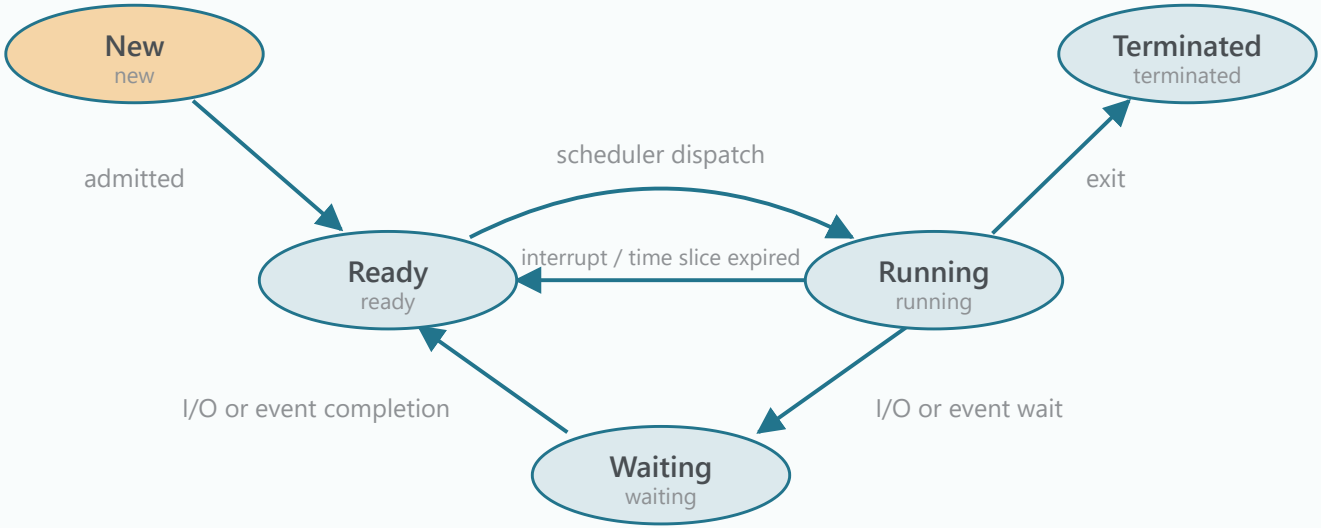
Process States (the Five-State Model)

New	Ready	Running	Waiting	Terminated
being created	waiting for the CPU	instructions are executing	(blocked) waiting for an event	finished execution



Only one process can be *running* on a core at any instant; many are ready or waiting.

State Transitions Step by Step

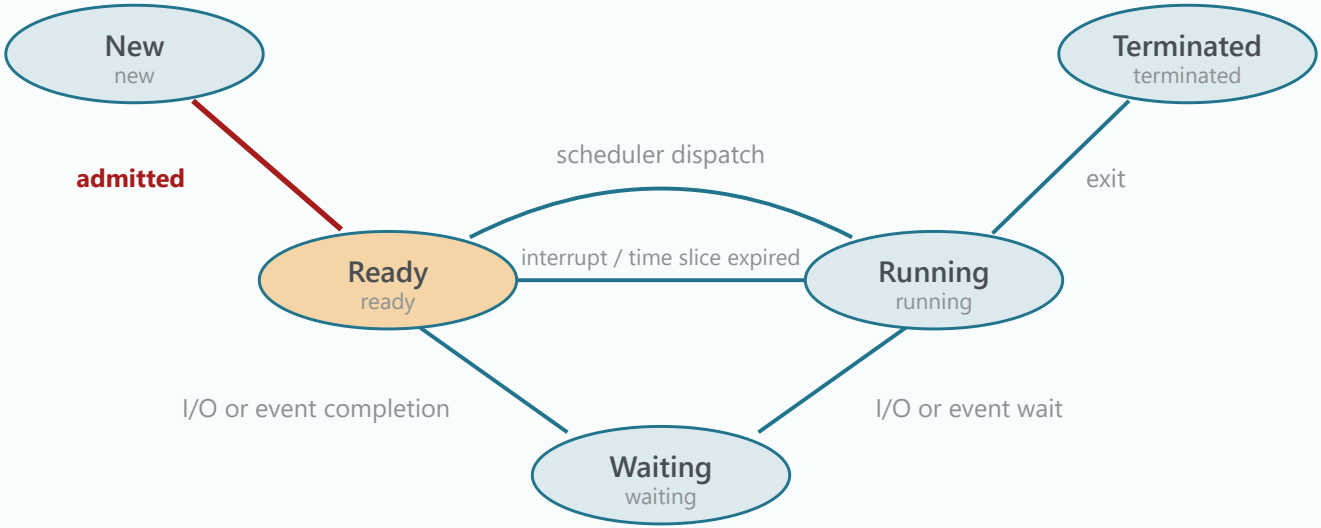


The user started a program: the PCB is being created and memory allocated → **New**.

Step 1 / 9

There is **no** Waiting → Running transition: a process whose event completes first enters the ready queue.

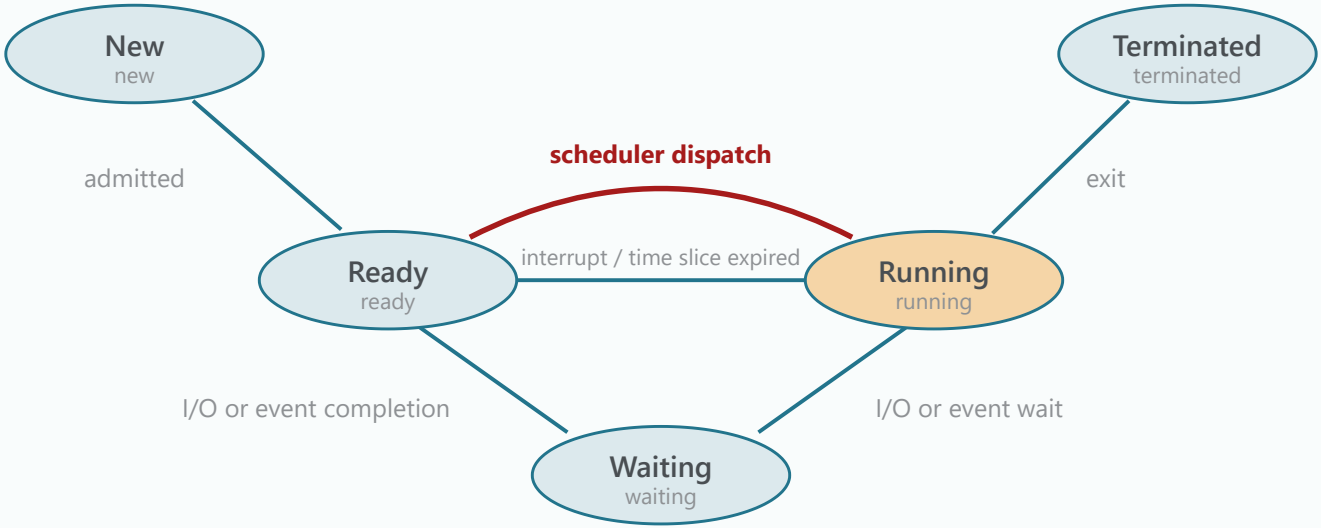
State Transitions Step by Step



The kernel admitted the process and put it in the ready queue → **Ready**.

There is **no** Waiting → Running transition: a process whose event completes first enters the ready queue.

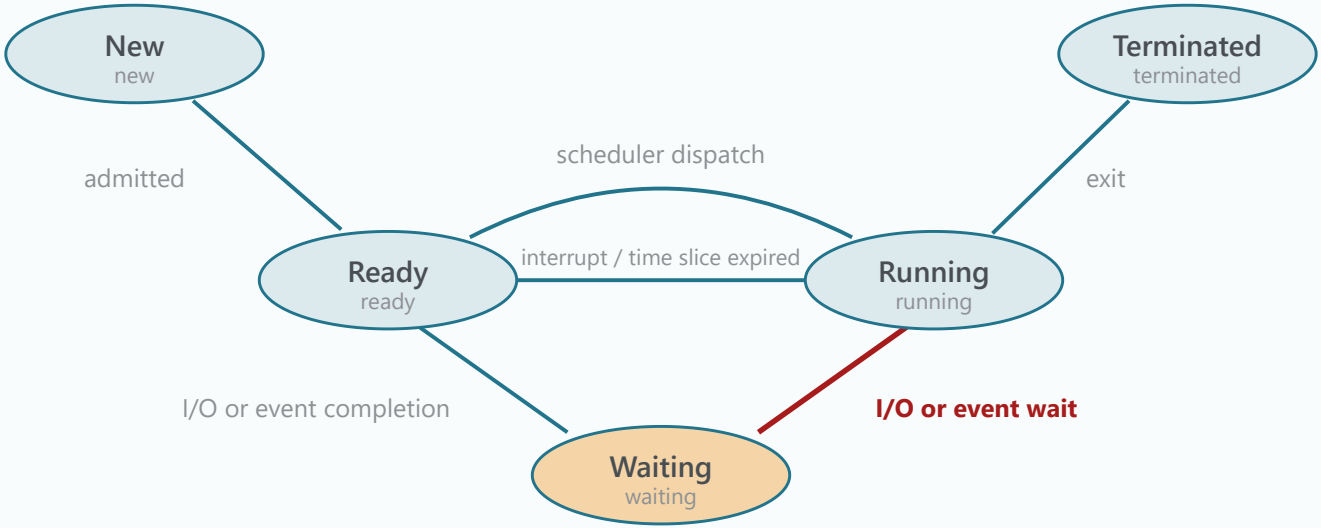
State Transitions Step by Step



The scheduler picked this process and its context was loaded → **Running**.

There is **no** Waiting → Running transition: a process whose event completes first enters the ready queue.

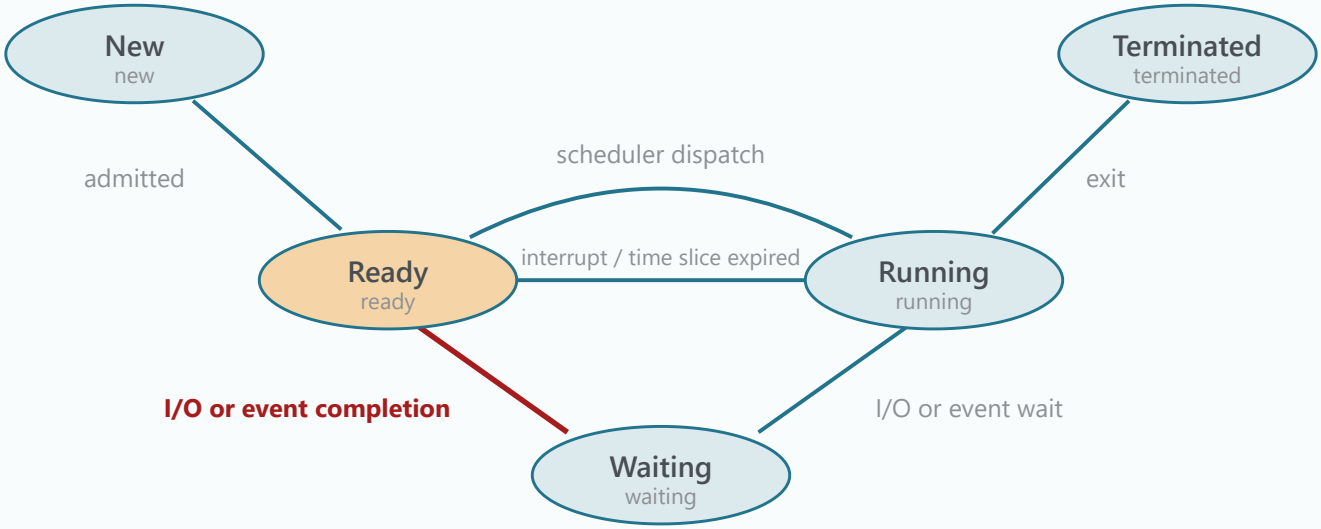
State Transitions Step by Step



The process asked to `read()` from disk; it gives up the CPU until the data arrives → **Waiting**.

There is **no** Waiting → Running transition: a process whose event completes first enters the ready queue.

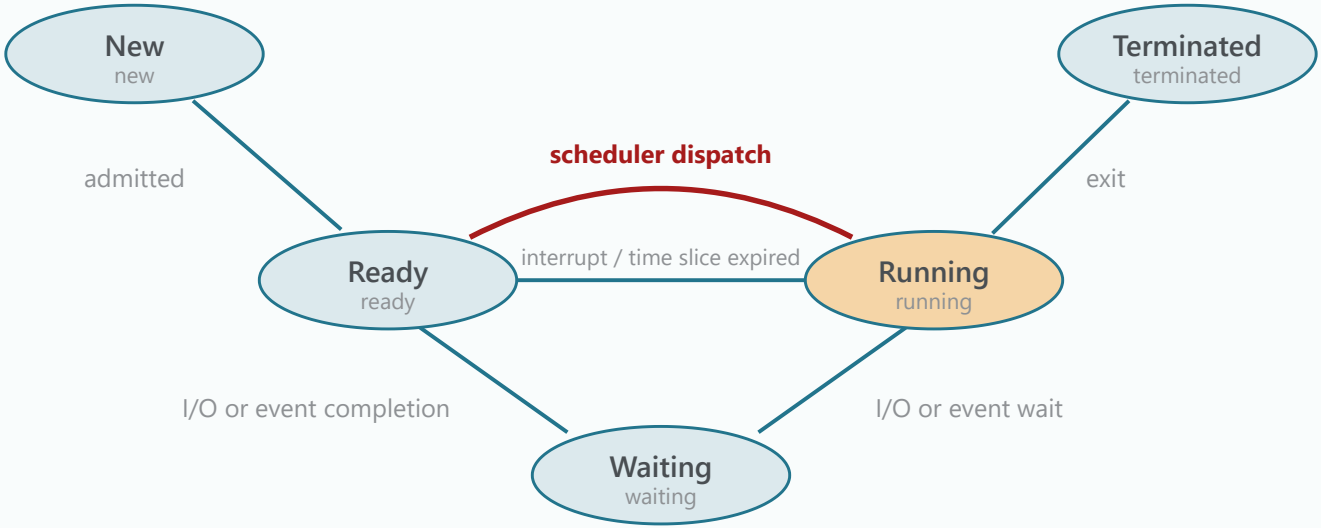
State Transitions Step by Step



Disk interrupt: the data is ready. The process does not run immediately; it returns to the ready queue → **Ready**.

There is **no** Waiting → Running transition: a process whose event completes first enters the ready queue.

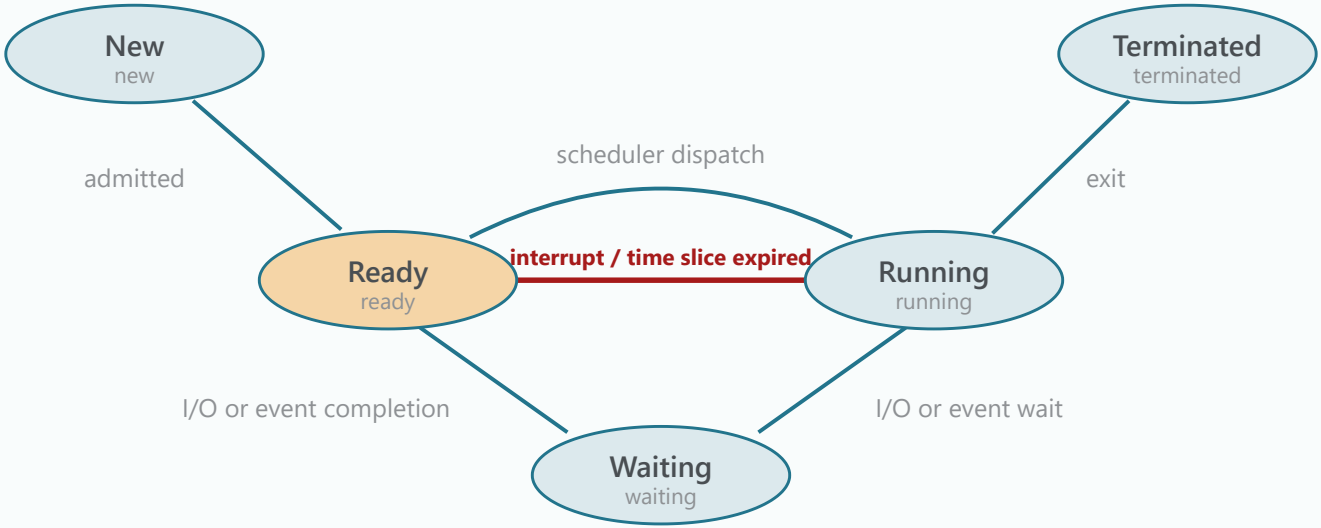
State Transitions Step by Step



Picked again → **Running**.

There is **no** Waiting → Running transition: a process whose event completes first enters the ready queue.

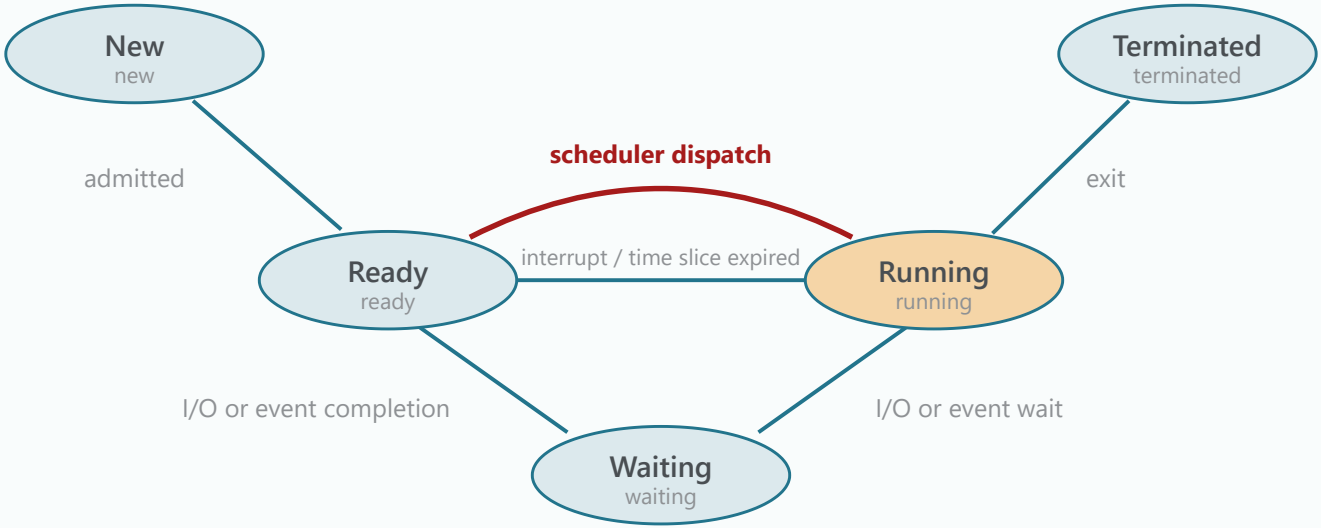
State Transitions Step by Step



Timer interrupt: the time slice expired (preemptive scheduling) → **Ready**.

There is **no** Waiting → Running transition: a process whose event completes first enters the ready queue.

State Transitions Step by Step

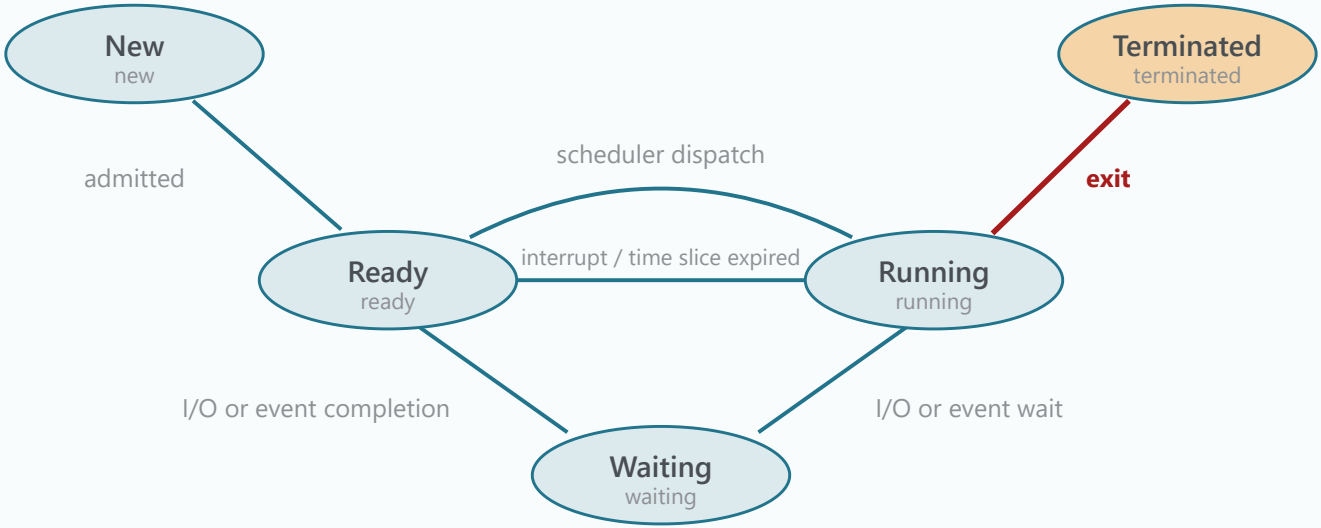


Picked once more → **Running**.

Step 8 / 9

There is **no** Waiting → Running transition: a process whose event completes first enters the ready queue.

State Transitions Step by Step



It called `exit()` → **Terminated**; a zombie until the parent calls `wait()`.

There is **no** Waiting → Running transition: a process whose event completes first enters the ready queue.

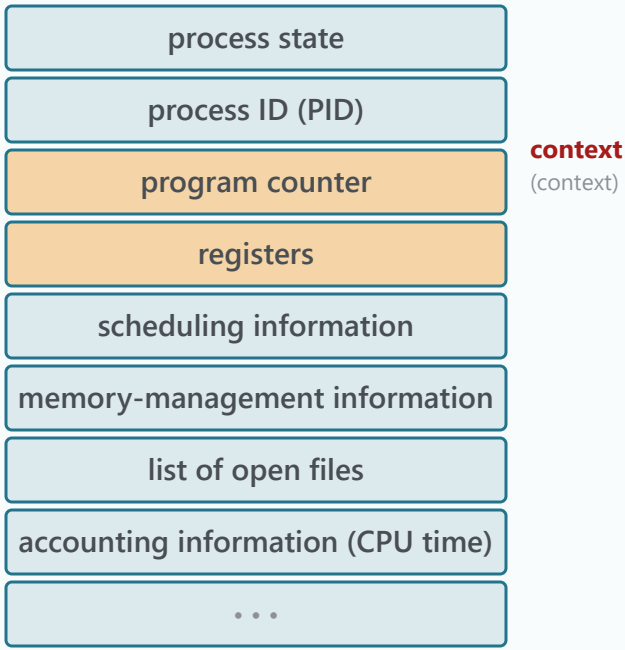
Process States on Linux

ps STAT	Meaning	In the model
`R`	running or ready to run	Running + Ready
`S`	interruptible sleep (waiting for an event)	Waiting
`D`	uninterruptible sleep (usually disk I/O)	Waiting
`T`	stopped (SIGSTOP, debugger)	suspended
`Z`	zombie	Terminated (not reaped)
`I`	idle kernel thread	—

```
$ ps -eo pid,stat,comm
  PID STAT COMMAND
    1 Ss   systemd
  890 Ss   sshd
 2105 Ss   bash
 2241 R+   ps
 2307 Z    writer <defunct>
```

Extra letters: `s` session leader, `+` foreground group. Linux does not distinguish ready from running; both are `R`.

The Process Control Block (PCB)



- The kernel keeps a **PCB** (process control block) for each process; everything that must be known about the process is here.
- The structure formed by all the PCBs is the **process table** (Tanenbaum's table lists the fields of these entries).
- When a process leaves the CPU, its program counter and registers are saved in the PCB; when it returns, they are loaded from there. This information is the **context**.
- PCBs live in kernel memory; user processes cannot access them directly.

PCB Fields (after Tanenbaum)

Process management	Memory management	File management
registers · program counter · program status word (PSW) · stack pointer · process state · priority · scheduling parameters · PID · parent process · process group · signals · start time · CPU time used · children's CPU time · time of next alarm	pointers to the text, data and stack segments; today a page-table pointer and a list of memory regions	root directory · working directory · file descriptors · user ID (UID) · group ID (GID)

PSW

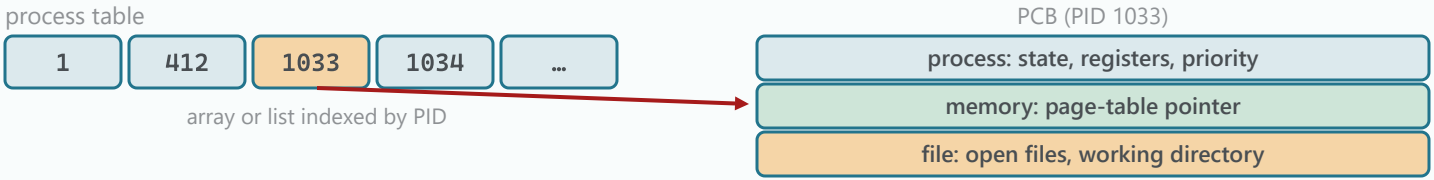
Status flags (zero, overflow…), execution mode (user/kernel), interrupt enables.

Process group

Processes managed together: a shell pipeline forms one group, and Ctrl+C goes to all of them.

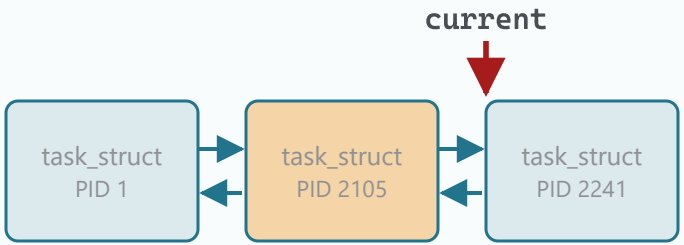
Next alarm

The delivery time of a SIGALRM requested with `alarm()`.



The PCB on Linux: task_struct

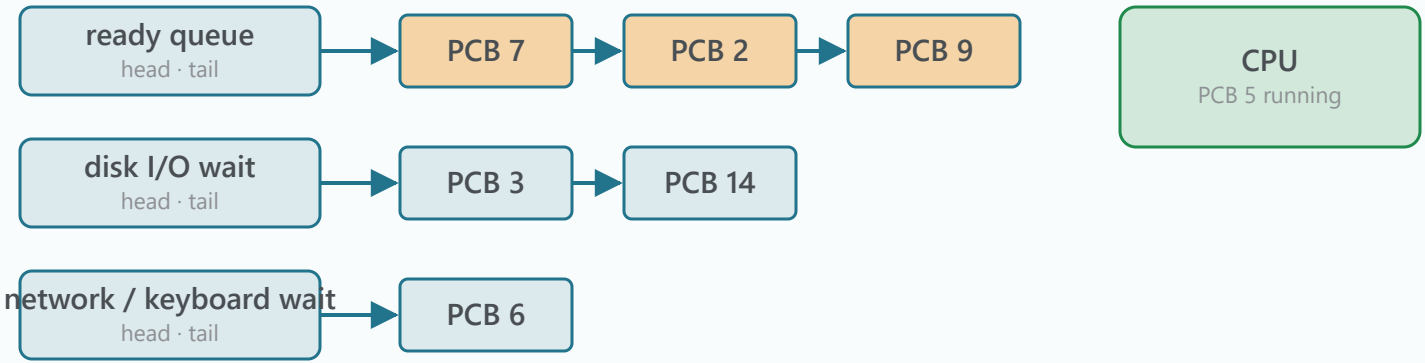
```c
struct task_struct {          /* <linux/sched.h> */
    pid_t pid;                    /* process ID */
    unsigned int __state;         /* process state */
    int prio;                     /* priority */
    struct sched_entity se;       /* CFS/EEVDF info */
    struct task_struct *parent;   /* parent process */
    struct list_head children;    /* children */
    struct files_struct *files;   /* open files */
    struct mm_struct *mm;         /* address space */
    /* ... hundreds more fields ... */
};
```



tasks: doubly linked list

- All tasks are on a doubly linked list; `current` points to the task running on that core.
- The `long state` / `time_slice` fields of older kernels have moved into `__state` and `sched_entity` in current kernels.

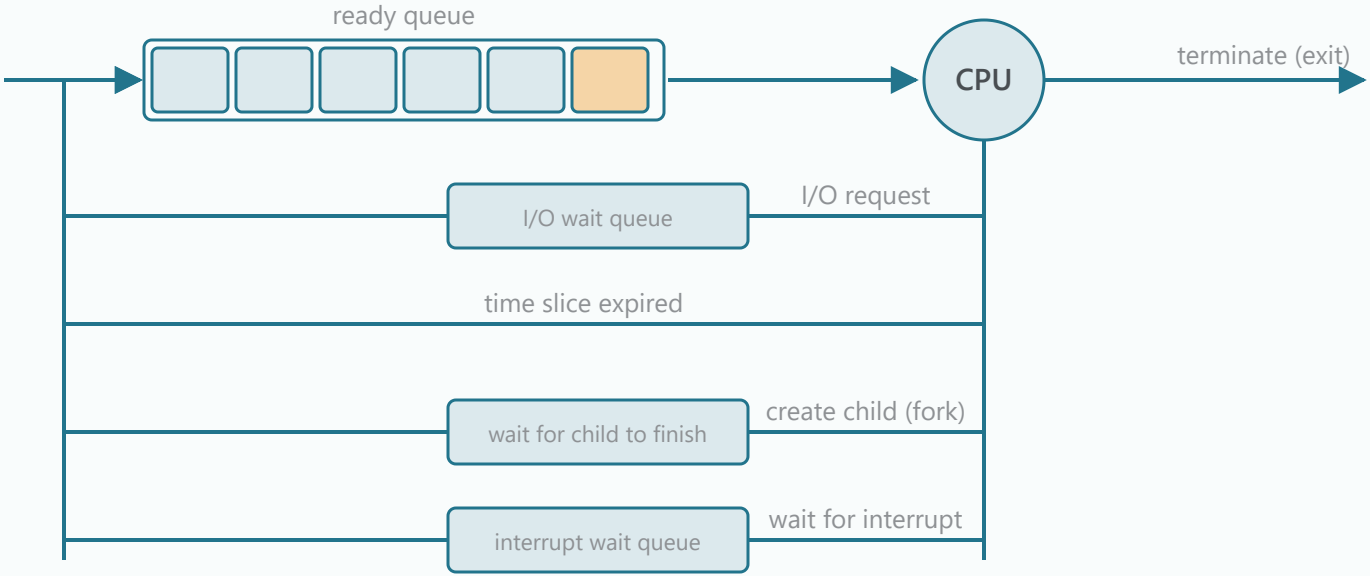
Ready and Wait Queues



queues are kept as linked lists of PCBs

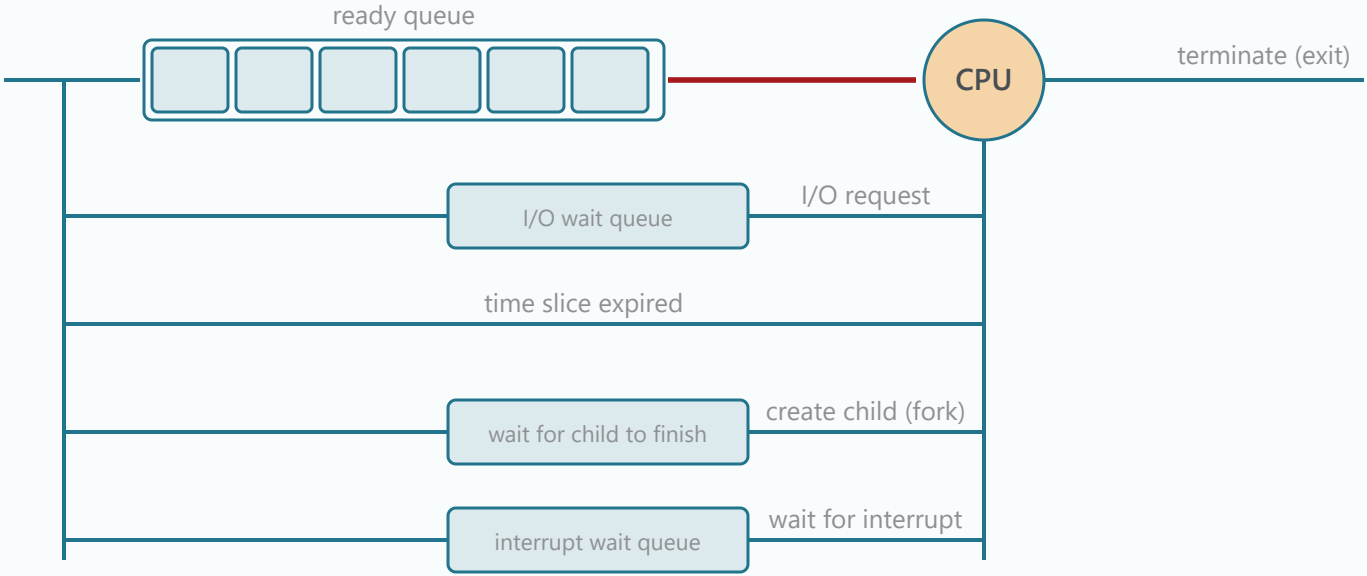
- The **scheduler** picks which ready process gets the CPU next; the goal is to keep the CPU busy and respond quickly to users.
- **Ready queue:** processes in memory waiting to execute. **Wait queues:** processes waiting for a specific event (I/O completion, a child terminating...); there is a separate queue for each kind of event.
- Linux gives each core its own runqueue; CFS/EEVDF keeps it in a red-black tree.

Queueing Diagram: A Process's Journey



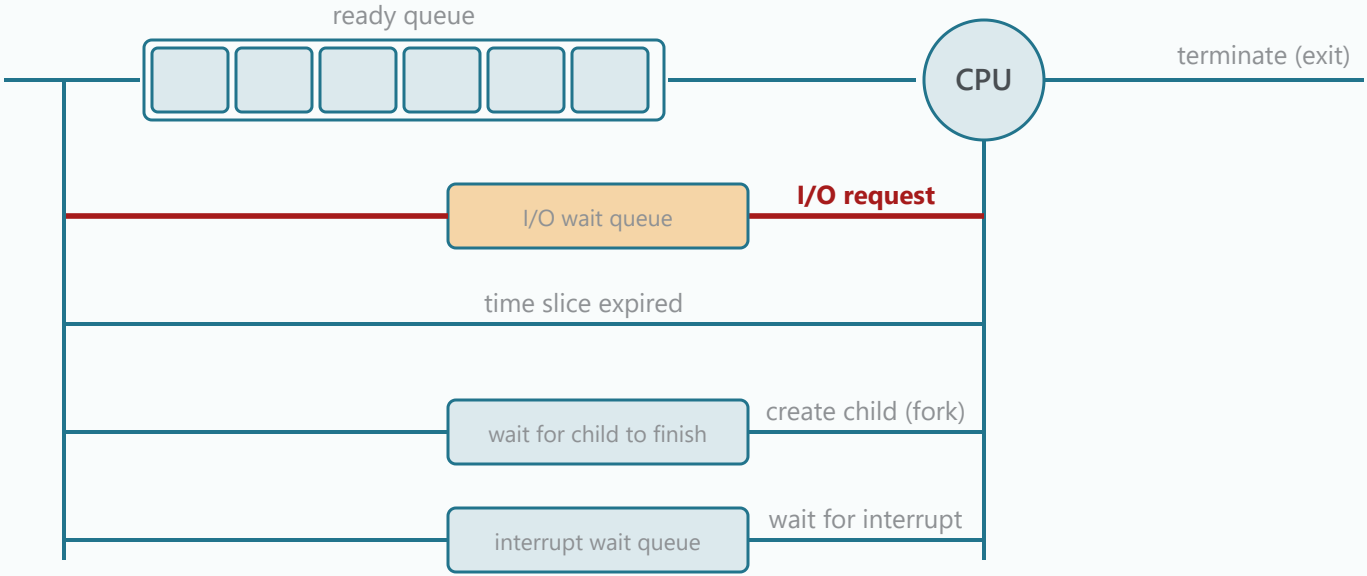
A new process is added to the tail of the ready queue and waits its turn.

Queueing Diagram: A Process's Journey



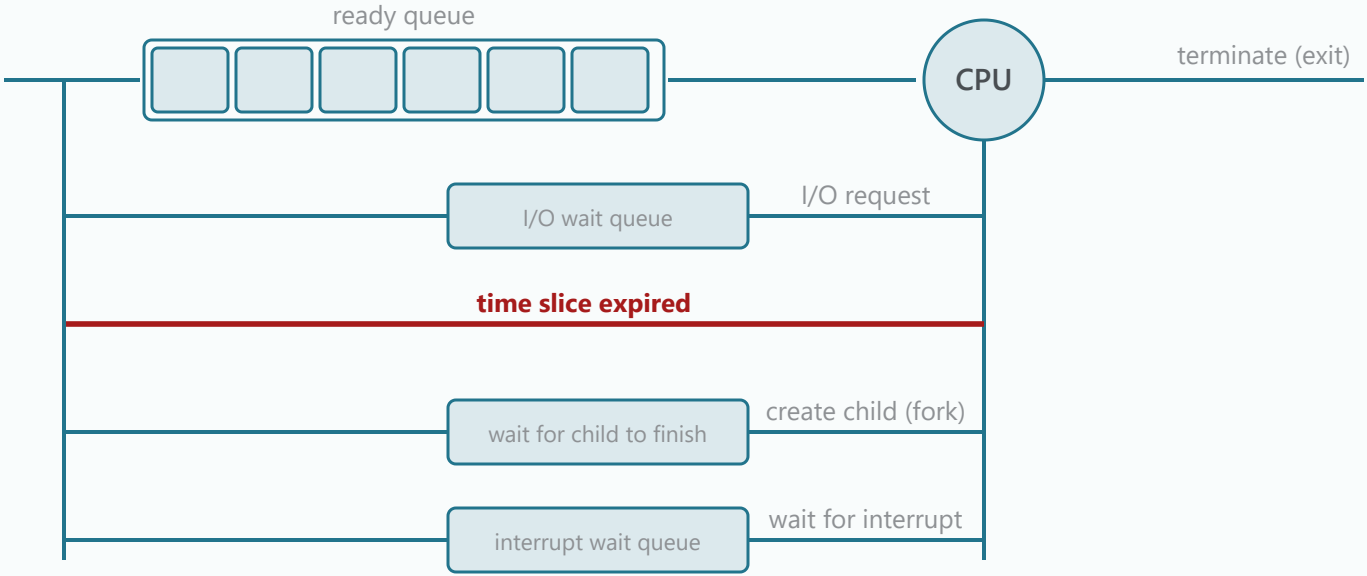
The scheduler dispatches it: the process runs on the CPU.

Queueing Diagram: A Process's Journey



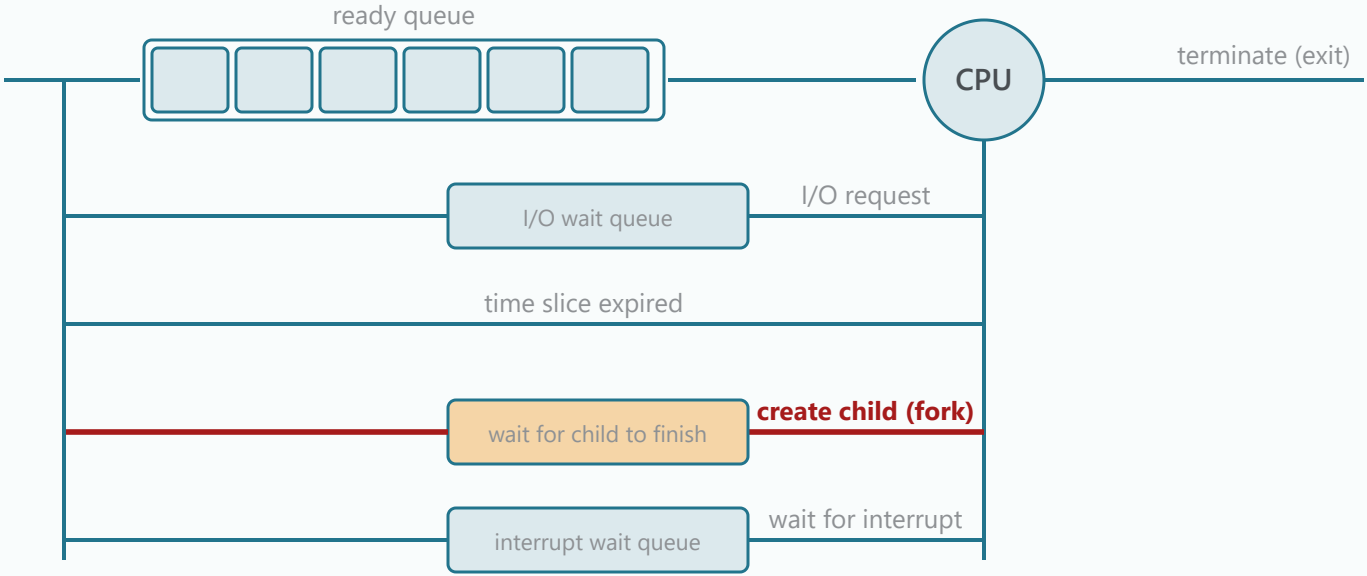
It made an I/O request: it moves to the I/O wait queue and returns to the ready queue when the I/O completes.

Queueing Diagram: A Process's Journey



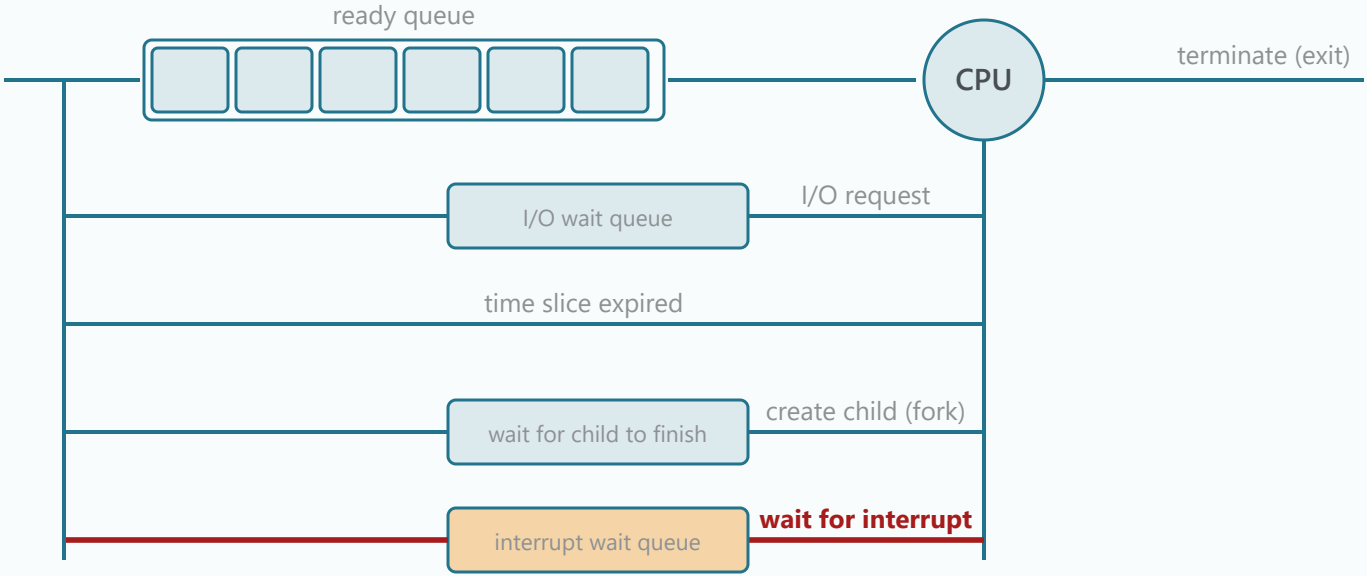
Time slice expired: a timer interrupt takes it off the CPU and it goes straight back to the ready queue.

Queueing Diagram: A Process's Journey



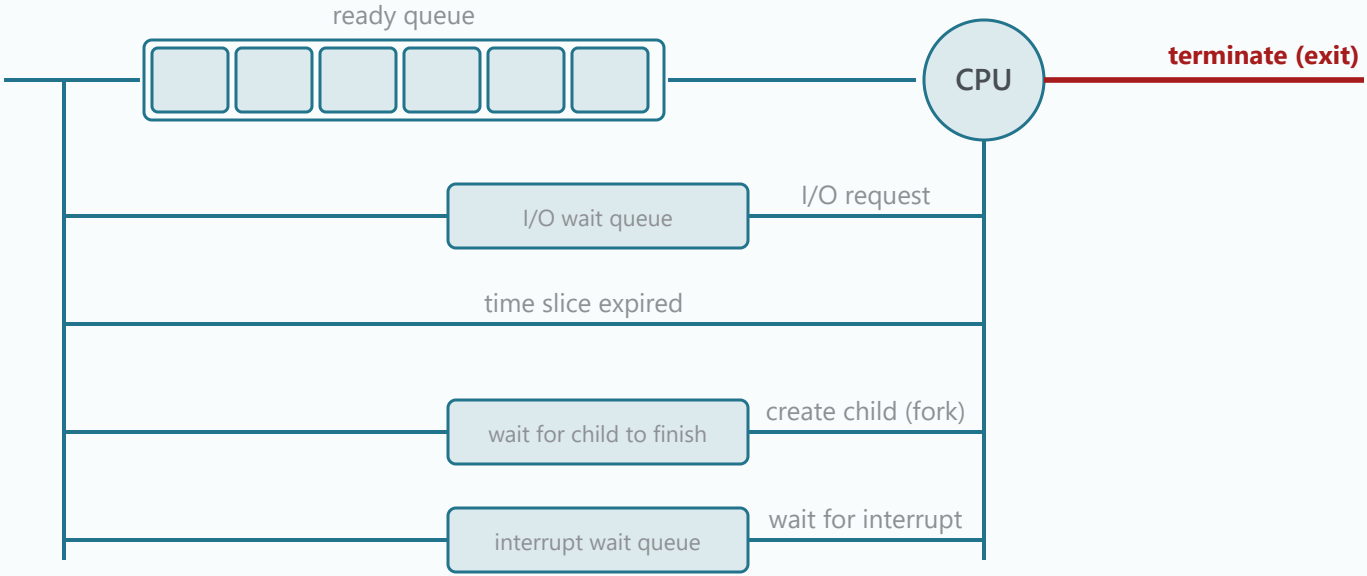
It created a child and called `wait()` : it waits until the child finishes.

Queueing Diagram: A Process's Journey



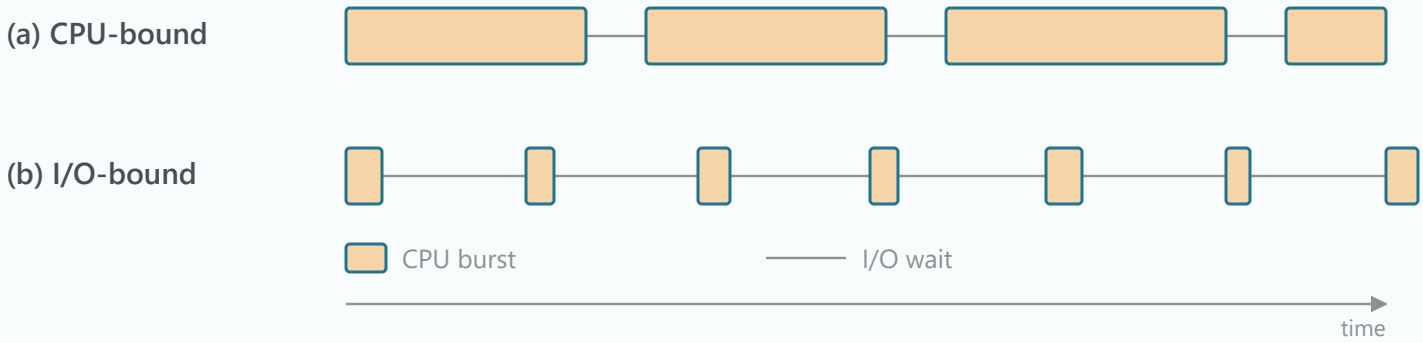
It waits for an interrupt/event (e.g. `sleep()`, a network packet).

Queueing Diagram: A Process's Journey



Its work is done: it leaves the system with `exit()`; its PCB is removed when the parent calls `wait`.

CPU-Bound and I/O-Bound Processes



CPU-bound

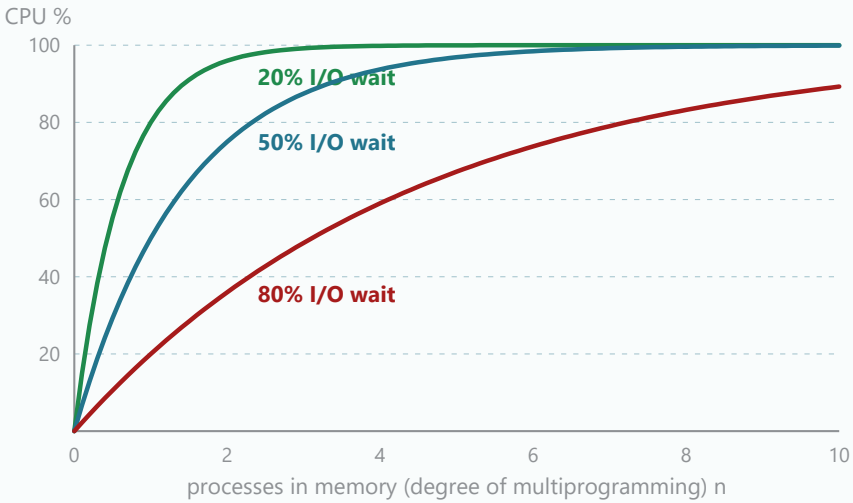
Long computation bursts, infrequent I/O: compiling, video encoding, scientific computing.

I/O-bound

Short CPU bursts, frequent waits: editors, web servers, database clients.

- Because CPUs improve much faster than disks, processes are becoming more and more **I/O-bound**.
- An I/O-bound process that becomes ready should get the CPU at once (fast context switch); a good mix of both kinds is needed.

Modeling Multiprogramming



Suppose a process spends a fraction **p** of its time waiting for I/O. With **n** independent processes in memory, the probability that all are waiting at once is $\mathbf{p^n}$:

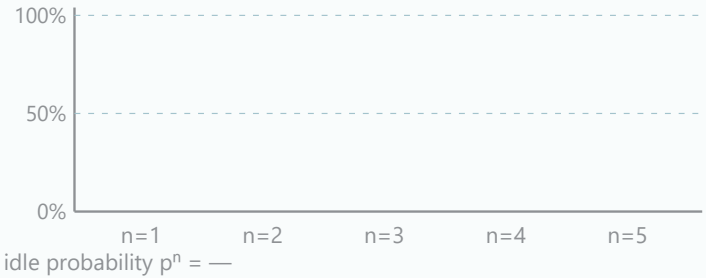
$$\textbf{CPU utilization} = 1 - p^n$$

E.g. p = 0.8: n = 1 → 20%, n = 4 → 59%, n = 10 → 89%.

A rough model: it assumes the processes are independent.

Code Trace: CPU Utilization 1 – p^n

```
▶ double p = 0.8;       /* I/O fraction */
▶ double pn = 1.0;
3 for (int n = 1; n <= 5; n++) {
4   pn = pn * p;        /* all waiting */
5   double util = 1 - pn;
6   printf("n=%d %.0f%%\n", n, util * 100);
7 }
```



idle probability p^n = —

Each process spends **80%** of its time waiting for I/O. The CPU is idle only when **all** processes in memory are waiting at once.

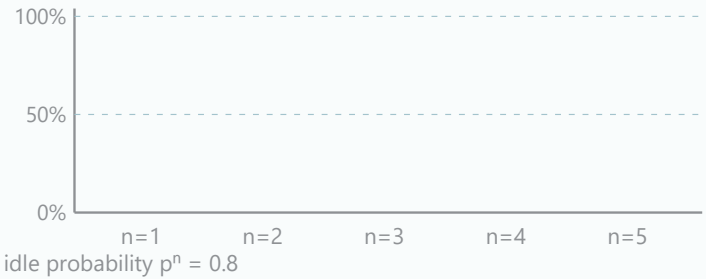
Step 1 / 11

VARIABLES

`p = 0.8` `pn = 1.0`

Code Trace: CPU Utilization 1 – p^n

```
1 double p = 0.8;       /* I/O fraction */
2 double pn = 1.0;
▶ for (int n = 1; n <= 5; n++) {
▶   pn = pn * p;        /* all waiting */
5   double util = 1 - pn;
6   printf("n=%d %.0f%%\n", n, util * 100);
7 }
```



idle probability $p^n = 0.8$

n = 1: a single process; the probability that the CPU is idle is **$p^n = 0.8$**.

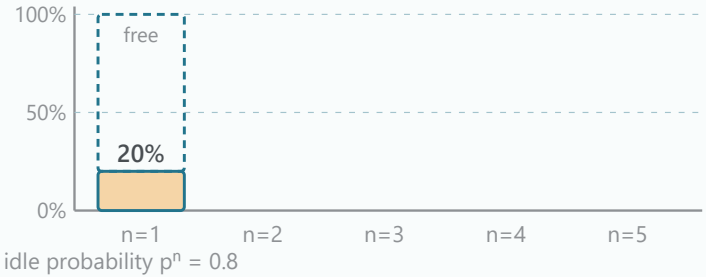
Step 2 / 11

VARIABLES

`p = 0.8` `pn = 0.8` `n = 1`

Code Trace: CPU Utilization 1 − p^n

```
1 double p = 0.8;       /* I/O fraction */
2 double pn = 1.0;
3 for (int n = 1; n <= 5; n++) {
4   pn = pn * p;        /* all waiting */
▶   double util = 1 - pn;
▶   printf("n=%d %.0f%%\n", n, util * 100);
7 }
```



idle probability p^n = 0.8

util = 1 − 0.8 = **20%**: on a uniprogrammed system the CPU is idle four-fifths of the time.

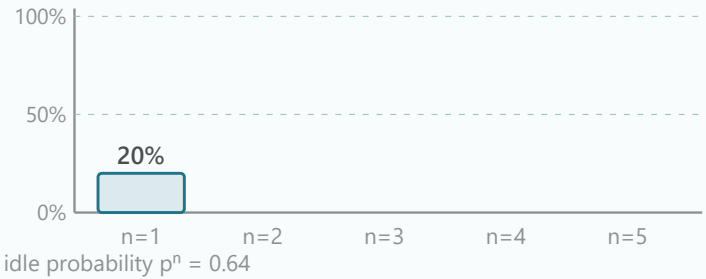
Step 3 / 11

VARIABLES

`p = 0.8` `pn = 0.8` `n = 1` `util = 0.2`

Code Trace: CPU Utilization 1 – p^n

```
1 double p = 0.8;       /* I/O fraction */
2 double pn = 1.0;
▶ for (int n = 1; n <= 5; n++) {
▶   pn = pn * p;        /* all waiting */
5   double util = 1 - pn;
6   printf("n=%d %.0f%%\n", n, util * 100);
7 }
```



idle probability p^n = 0.64

n = 2: the probability that all wait at once is 0.8^2 = **0.64**.

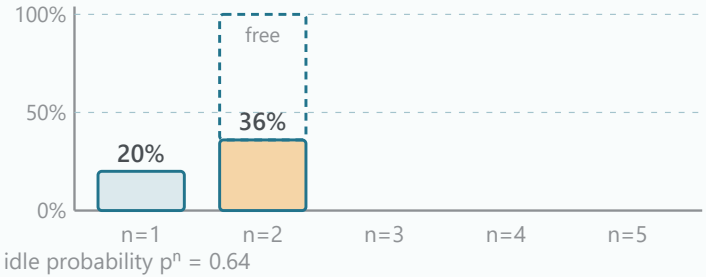
Step 4 / 11

VARIABLES

`p = 0.8` `pn = 0.64` `n = 2`

Code Trace: CPU Utilization 1 – p^n

```
1 double p = 0.8;       /* I/O fraction */
2 double pn = 1.0;
3 for (int n = 1; n <= 5; n++) {
4   pn = pn * p;        /* all waiting */
▶   double util = 1 - pn;
▶   printf("n=%d %.0f%%\n", n, util * 100);
7 }
```



idle probability p^n = 0.64

util = **36%**.

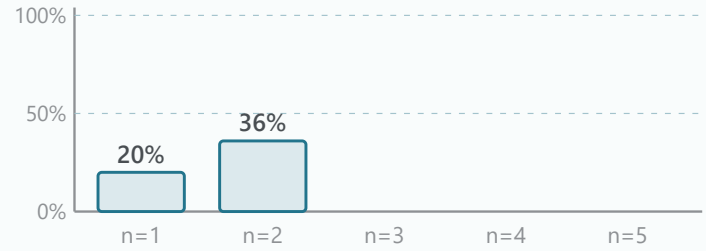
Step 5 / 11

VARIABLES

`p = 0.8` `pn = 0.64` `n = 2` `util = 0.36`

Code Trace: CPU Utilization 1 − p^n

```
1 double p = 0.8;       /* I/O fraction */
2 double pn = 1.0;
▶ for (int n = 1; n <= 5; n++) {
▶   pn = pn * p;        /* all waiting */
5   double util = 1 - pn;
6   printf("n=%d %.0f%%\n", n, util * 100);
7 }
```



idle probability p^n = 0.512

n = 3: the probability that all wait at once is 0.8^3 = **0.512**.

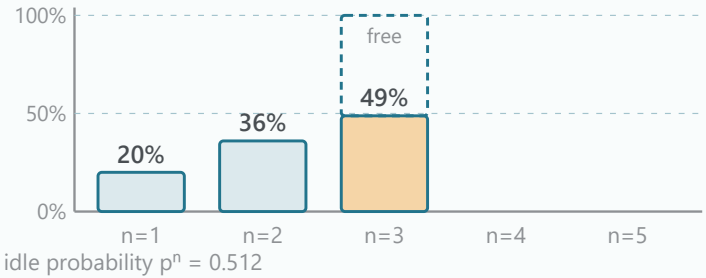
Step 6 / 11

VARIABLES

`p = 0.8` `pn = 0.512` `n = 3`

Code Trace: CPU Utilization 1 – p^n

```
1 double p = 0.8;       /* I/O fraction */
2 double pn = 1.0;
3 for (int n = 1; n <= 5; n++) {
4   pn = pn * p;        /* all waiting */
▶   double util = 1 - pn;
▶   printf("n=%d %.0f%%\n", n, util * 100);
7 }
```



idle probability p^n = 0.512

util = **48.8%**.

Step 7 / 11

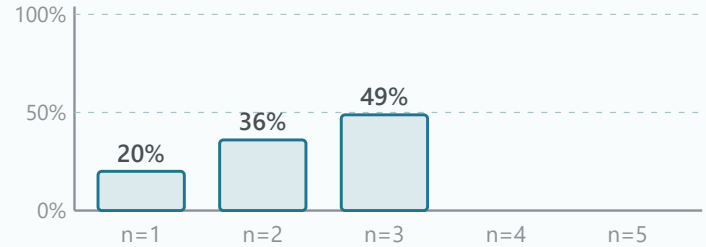
VARIABLES

`p = 0.8` `pn = 0.512` `n = 3`

`util = 0.488`

Code Trace: CPU Utilization 1 – p^n

```
1 double p = 0.8;       /* I/O fraction */
2 double pn = 1.0;
▶ for (int n = 1; n <= 5; n++) {
▶   pn = pn * p;        /* all waiting */
5   double util = 1 - pn;
6   printf("n=%d %.0f%%\n", n, util * 100);
7 }
```



idle probability p^n = 0.4096

n = 4: the probability that all wait at once is 0.8^4 = **0.4096**.

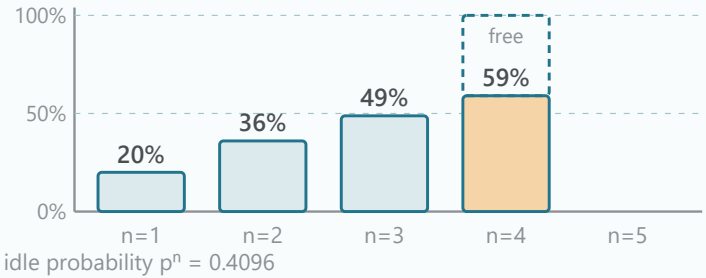
Step 8 / 11

VARIABLES

`p = 0.8` `pn = 0.4096` `n = 4`

Code Trace: CPU Utilization 1 – p^n

```
1 double p = 0.8;       /* I/O fraction */
2 double pn = 1.0;
3 for (int n = 1; n <= 5; n++) {
4   pn = pn * p;        /* all waiting */
▶   double util = 1 - pn;
▶   printf("n=%d %.0f%%\n", n, util * 100);
7 }
```



idle probability p^n = 0.4096

util = **59.04%**.

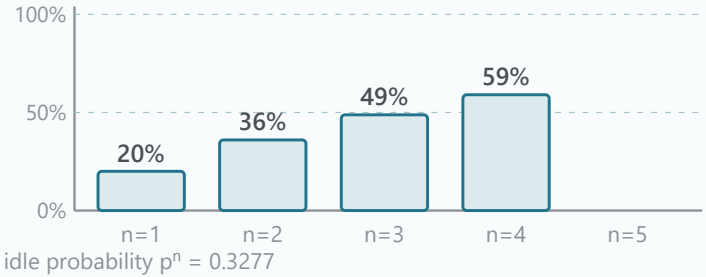
Step 9 / 11

VARIABLES

`p = 0.8` `pn = 0.4096` `n = 4` `util = 0.5904`

Code Trace: CPU Utilization 1 – p^n

```
1 double p = 0.8;       /* I/O fraction */
2 double pn = 1.0;
▶ for (int n = 1; n <= 5; n++) {
▶   pn = pn * p;        /* all waiting */
5   double util = 1 - pn;
6   printf("n=%d %.0f%%\n", n, util * 100);
7 }
```



idle probability p^n = 0.3277

n = 5: the probability that all wait at once is 0.8^5 = **0.32768**.

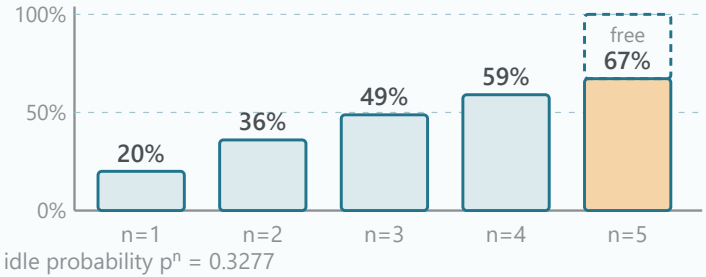
Step 10 / 11

VARIABLES

`p = 0.8` `pn = 0.32768` `n = 5`

Code Trace: CPU Utilization 1 – p^n

```
1 double p = 0.8;       /* I/O fraction */
2 double pn = 1.0;
3 for (int n = 1; n <= 5; n++) {
4   pn = pn * p;        /* all waiting */
▶   double util = 1 - pn;
▶   printf("n=%d %.0f%%\n", n, util * 100);
7 }
```



idle probability p^n = 0.3277

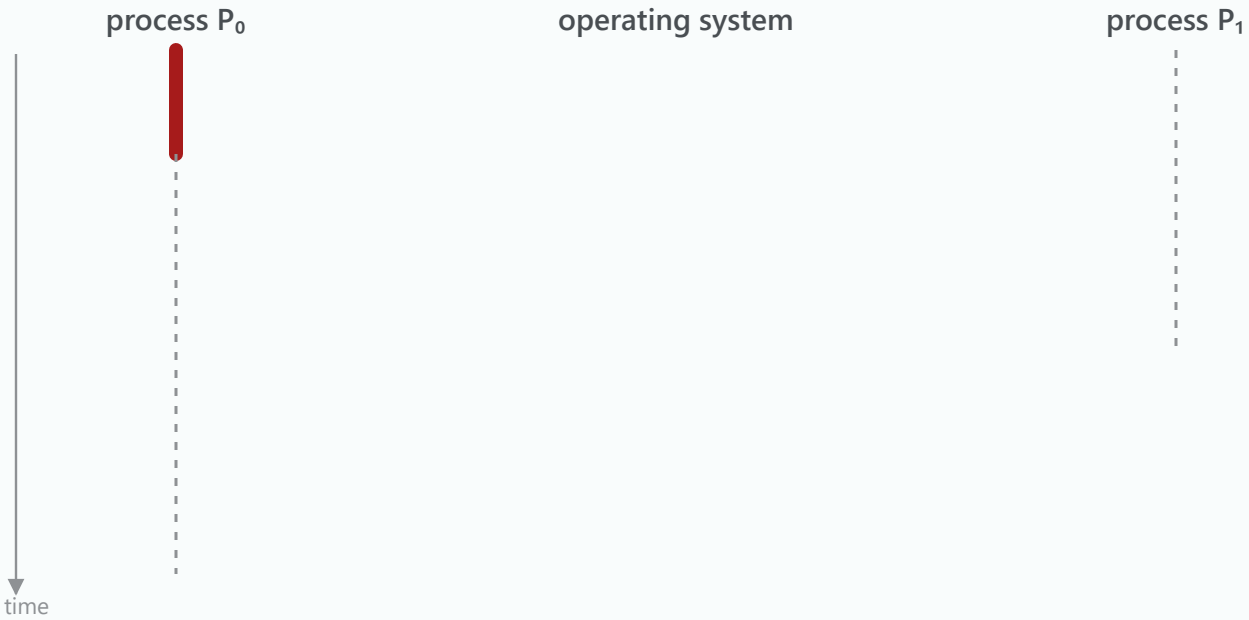
util = **67.23%**. Each new process adds **less** than the one before: an increase of 8 points.

Step 11 / 11

VARIABLES

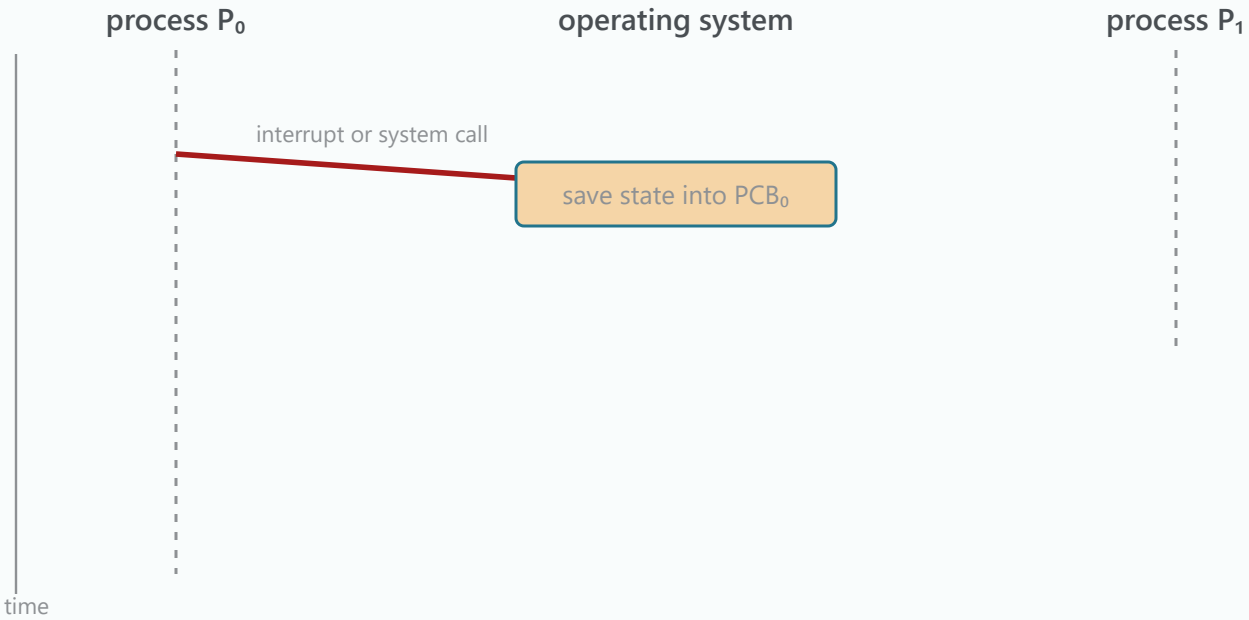
`p = 0.8` `pn = 0.32768` `n = 5` `util = 0.67232`

Context Switch



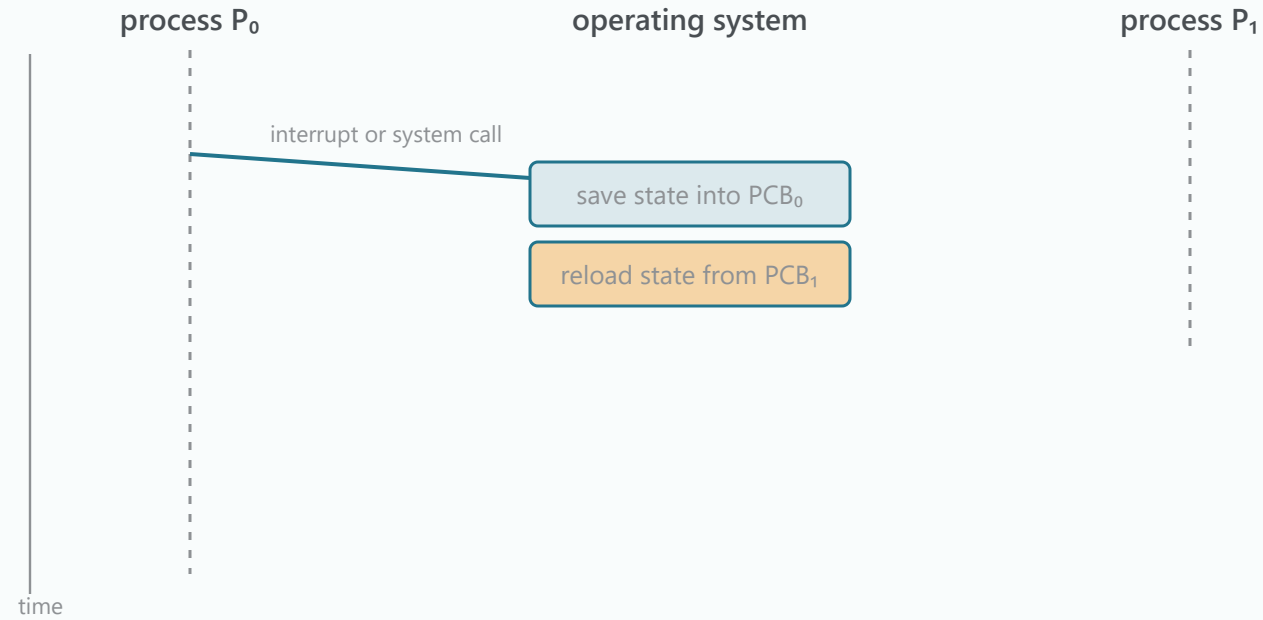
P_0 is running (thick line); P_1 waits in the ready queue.

Context Switch



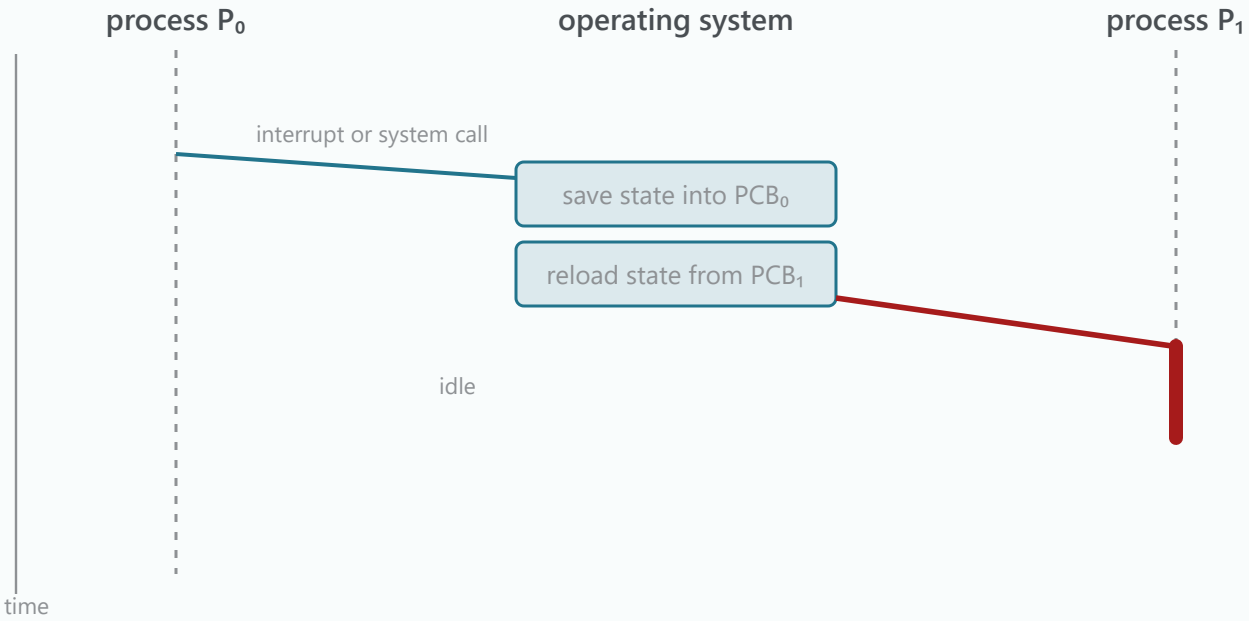
Interrupt or system call: the CPU enters kernel mode and P_0's program counter and registers are saved in **PCB_0**.

Context Switch



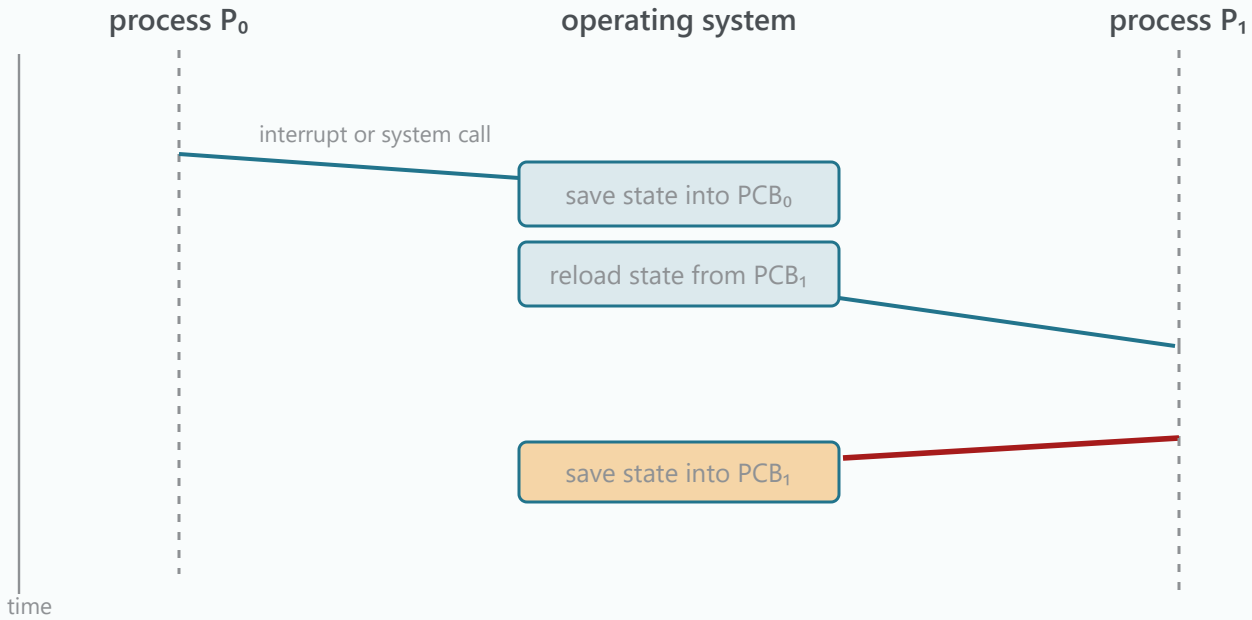
The scheduler picks P_1; P_1's context is loaded from **PCB_1** (the address space changes too).

Context Switch



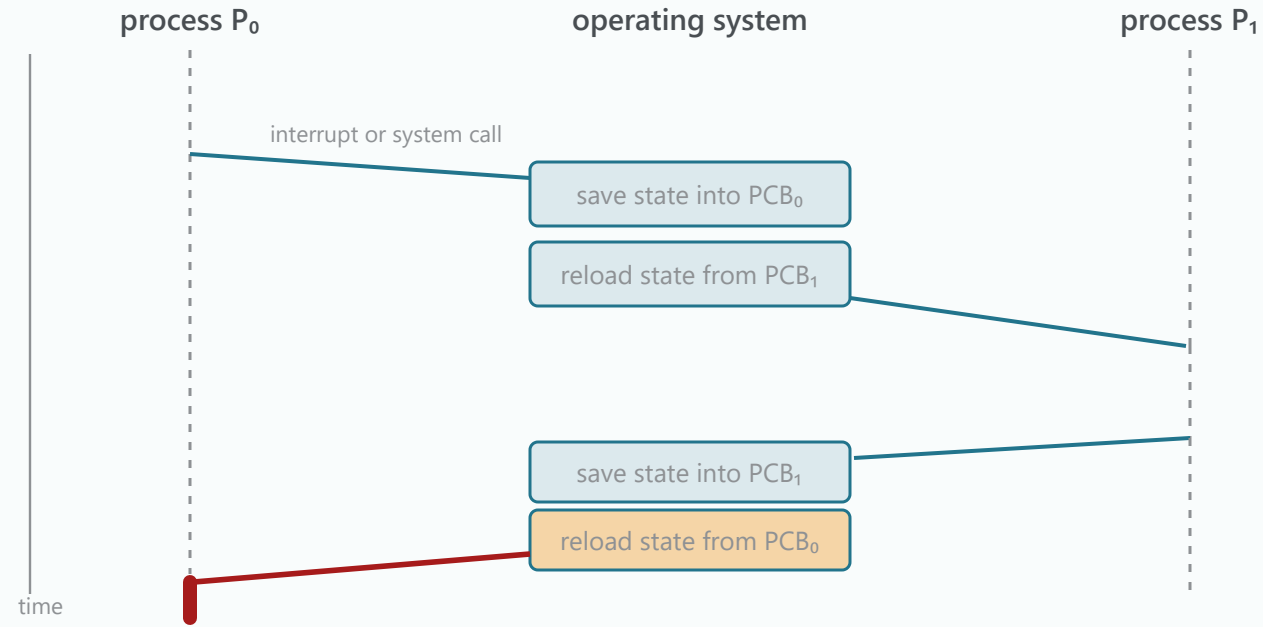
P_1 is running; P_0 is idle (ready or waiting).

Context Switch



Another interrupt: P_1's state is saved in PCB_1.

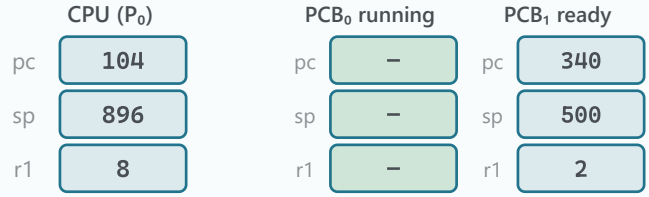
Context Switch



Reloaded from PCB_0; P_0 resumes where it left off. No user work is done in the gray parts in between: **overhead**.

Code Trace: Context Switch

```
1  struct pcb { int pc, sp, r1; char state; };
2  struct pcb P[2];   /* process table */
3  int pc, sp, r1;    /* CPU registers */
4  void context_switch(int from, int to) {
5    P[from].pc = pc;  P[from].sp = sp;  /* save */
6    P[from].r1 = r1;  P[from].state = 'R';
7    pc = P[to].pc;  sp = P[to].sp;  /* load */
8    r1 = P[to].r1;  P[to].state = 'X';
9  }
▶ /* interrupt */  context_switch(0, 1);
11 /* ... P1 runs ... */
12 /* interrupt */  context_switch(1, 0);
```



P_0 is running: the registers hold **P_0's** values. P_1's last state is saved in PCB_1. A timer interrupt arrives; the kernel decides to switch to P_1.

Step 1 / 11

VARIABLES

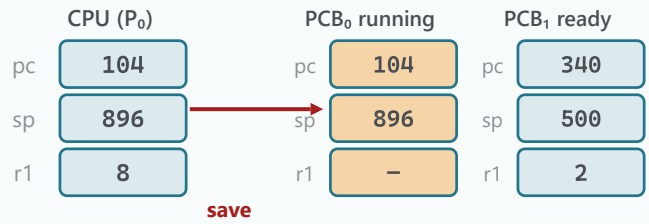
`pc = 104` `sp = 896` `r1 = 8`

CALL STACK

`interrupt handler [kernel]`

Code Trace: Context Switch

```
1  struct pcb { int pc, sp, r1; char state; };
2  struct pcb P[2];   /* process table */
3  int pc, sp, r1;    /* CPU registers */
4  void context_switch(int from, int to) {
▶    P[from].pc = pc;  P[from].sp = sp;  /* save */
6    P[from].r1 = r1;  P[from].state = 'R';
7    pc = P[to].pc;  sp = P[to].sp;  /* load */
8    r1 = P[to].r1;  P[to].state = 'X';
9  }
10 /* interrupt */  context_switch(0, 1);
11 /* ... P1 runs ... */
12 /* interrupt */  context_switch(1, 0);
```



Save: P_0's pc and sp are written to PCB_0. If they were lost, P_0 could not return to where it was.

Step 2 / 11

VARIABLES

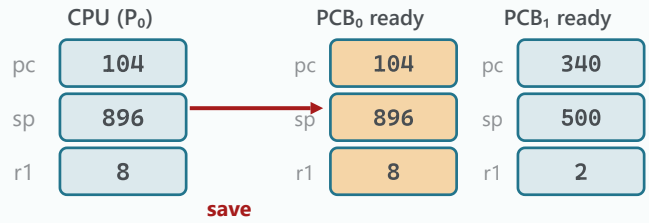
`pc = 104` `sp = 896` `r1 = 8`

CALL STACK

`context_switch(0, 1)`

Code Trace: Context Switch

```
1  struct pcb { int pc, sp, r1; char state; };
2  struct pcb P[2];   /* process table */
3  int pc, sp, r1;    /* CPU registers */
4  void context_switch(int from, int to) {
5    P[from].pc = pc;  P[from].sp = sp;  /* save */
▶    P[from].r1 = r1;  P[from].state = 'R';
7    pc = P[to].pc;  sp = P[to].sp;  /* load */
8    r1 = P[to].r1;  P[to].state = 'X';
9  }
10 /* interrupt */  context_switch(0, 1);
11 /* ... P1 runs ... */
12 /* interrupt */  context_switch(1, 0);
```



The general register is saved too; P_0 is now **ready** (R).

Step 3 / 11

VARIABLES

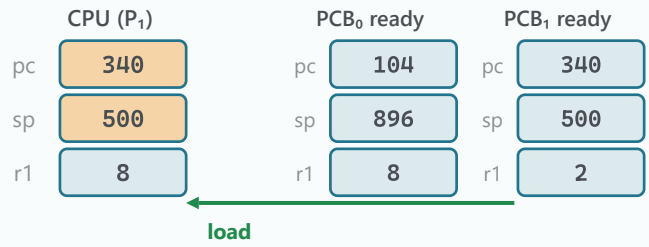
`pc = 104` `sp = 896` `r1 = 8`

CALL STACK

`context_switch(0, 1)`

Code Trace: Context Switch

```
1  struct pcb { int pc, sp, r1; char state; };
2  struct pcb P[2];   /* process table */
3  int pc, sp, r1;    /* CPU registers */
4  void context_switch(int from, int to) {
5    P[from].pc = pc;  P[from].sp = sp;  /* save */
6    P[from].r1 = r1;  P[from].state = 'R';
▶    pc = P[to].pc;  sp = P[to].sp;  /* load */
8    r1 = P[to].r1;  P[to].state = 'X';
9  }
10 /* interrupt */  context_switch(0, 1);
11 /* ... P1 runs ... */
12 /* interrupt */  context_switch(1, 0);
```



Load: P_1's pc and sp go into the CPU. From now on the CPU uses P_1's code and stack.

Step 4 / 11

VARIABLES

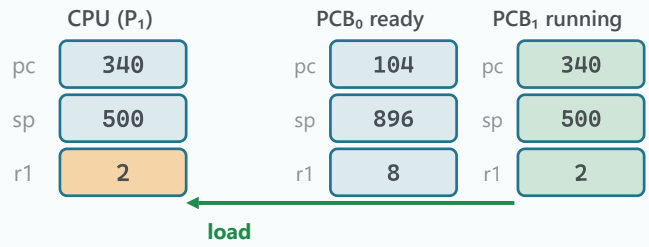
`pc = 340` `sp = 500` `r1 = 8`

CALL STACK

`context_switch(0, 1)`

Code Trace: Context Switch

```
1  struct pcb { int pc, sp, r1; char state; };
2  struct pcb P[2];   /* process table */
3  int pc, sp, r1;    /* CPU registers */
4  void context_switch(int from, int to) {
5    P[from].pc = pc;  P[from].sp = sp;  /* save */
6    P[from].r1 = r1;  P[from].state = 'R';
7    pc = P[to].pc;  sp = P[to].sp;  /* load */
▶    r1 = P[to].r1;  P[to].state = 'X';
9  }
10 /* interrupt */  context_switch(0, 1);
11 /* ... P1 runs ... */
12 /* interrupt */  context_switch(1, 0);
```



r1 = 2 is loaded; P_1 is **running** (X). No user work was done meanwhile: a context switch is pure **overhead**.

Step 5 / 11

VARIABLES

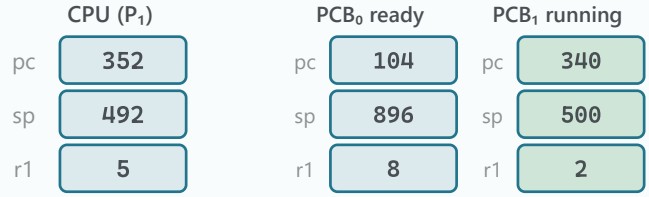
`pc = 340` `sp = 500` `r1 = 2`

CALL STACK

`context_switch(0, 1)`

Code Trace: Context Switch

```
1  struct pcb { int pc, sp, r1; char state; };
2  struct pcb P[2];   /* process table */
3  int pc, sp, r1;    /* CPU registers */
4  void context_switch(int from, int to) {
5    P[from].pc = pc;  P[from].sp = sp;  /* save */
6    P[from].r1 = r1;  P[from].state = 'R';
7    pc = P[to].pc;  sp = P[to].sp;  /* load */
8    r1 = P[to].r1;  P[to].state = 'X';
9  }
10 /* interrupt */  context_switch(0, 1);
▶ /* ... P1 runs ... */
12 /* interrupt */  context_switch(1, 0);
```



P_1 ran a few instructions: pc 340 → **352**, r1 = 5, sp = 492. PCB_1 is out of date now; the current values exist only in the CPU.

Step 6 / 11

VARIABLES

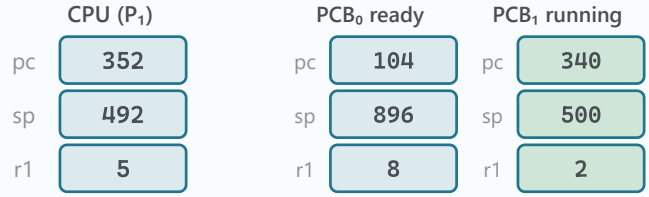
`pc = 352` `sp = 492` `r1 = 5`

CALL STACK

`P₁ user code`

Code Trace: Context Switch

```
1  struct pcb { int pc, sp, r1; char state; };
2  struct pcb P[2];   /* process table */
3  int pc, sp, r1;    /* CPU registers */
4  void context_switch(int from, int to) {
5    P[from].pc = pc;  P[from].sp = sp;  /* save */
6    P[from].r1 = r1;  P[from].state = 'R';
7    pc = P[to].pc;  sp = P[to].sp;  /* load */
8    r1 = P[to].r1;  P[to].state = 'X';
9  }
10 /* interrupt */  context_switch(0, 1);
11 /* ... P1 runs ... */
▶  /* interrupt */  context_switch(1, 0);
```



Another interrupt: the kernel decides to return to P_0.

Step 7 / 11

VARIABLES

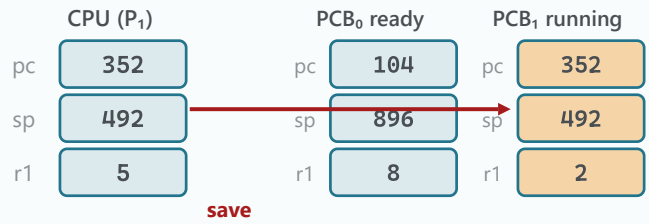
`pc = 352` `sp = 492` `r1 = 5`

CALL STACK

`interrupt handler [kernel]`

Code Trace: Context Switch

```
1  struct pcb { int pc, sp, r1; char state; };
2  struct pcb P[2];   /* process table */
3  int pc, sp, r1;    /* CPU registers */
4  void context_switch(int from, int to) {
▶    P[from].pc = pc;  P[from].sp = sp;  /* save */
6    P[from].r1 = r1;  P[from].state = 'R';
7    pc = P[to].pc;  sp = P[to].sp;  /* load */
8    r1 = P[to].r1;  P[to].state = 'X';
9  }
10 /* interrupt */  context_switch(0, 1);
11 /* ... P1 runs ... */
12 /* interrupt */  context_switch(1, 0);
```



P_1's pc and sp are saved in PCB_1.

Step 8 / 11

VARIABLES

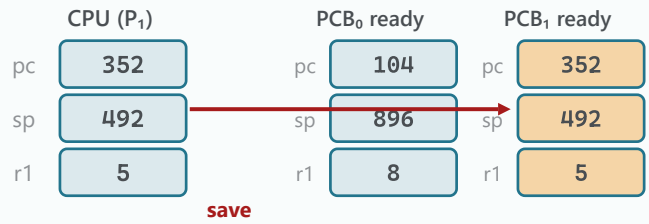
`pc = 352` `sp = 492` `r1 = 5`

CALL STACK

```
context_switch(1, 0)
```

Code Trace: Context Switch

```
1  struct pcb { int pc, sp, r1; char state; };
2  struct pcb P[2];   /* process table */
3  int pc, sp, r1;    /* CPU registers */
4  void context_switch(int from, int to) {
5    P[from].pc = pc;  P[from].sp = sp;  /* save */
▶    P[from].r1 = r1;  P[from].state = 'R';
7    pc = P[to].pc;  sp = P[to].sp;  /* load */
8    r1 = P[to].r1;  P[to].state = 'X';
9  }
10 /* interrupt */  context_switch(0, 1);
11 /* ... P1 runs ... */
12 /* interrupt */  context_switch(1, 0);
```



r1 is saved as well; P_1 is ready.

Step 9 / 11

VARIABLES

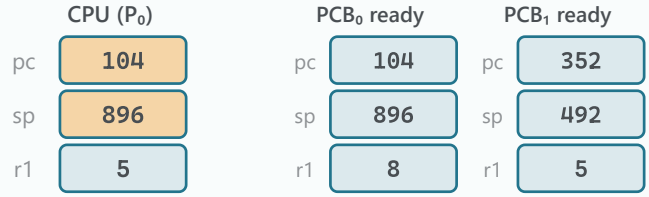
`pc = 352` `sp = 492` `r1 = 5`

CALL STACK

`context_switch(1, 0)`

Code Trace: Context Switch

```
1  struct pcb { int pc, sp, r1; char state; };
2  struct pcb P[2];   /* process table */
3  int pc, sp, r1;    /* CPU registers */
4  void context_switch(int from, int to) {
5    P[from].pc = pc;  P[from].sp = sp;  /* save */
6    P[from].r1 = r1;  P[from].state = 'R';
▶    pc = P[to].pc;  sp = P[to].sp;  /* load */
8    r1 = P[to].r1;  P[to].state = 'X';
9  }
10 /* interrupt */  context_switch(0, 1);
11 /* ... P1 runs ... */
12 /* interrupt */  context_switch(1, 0);
```



The values in PCB_0 are restored: pc = 104, sp = 896.

Step 10 / 11

VARIABLES

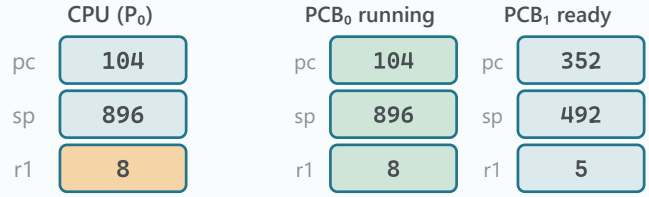
`pc = 104` `sp = 896` `r1 = 5`

CALL STACK

`context_switch(1, 0)`

Code Trace: Context Switch

```
1  struct pcb { int pc, sp, r1; char state; };
2  struct pcb P[2];   /* process table */
3  int pc, sp, r1;    /* CPU registers */
4  void context_switch(int from, int to) {
5    P[from].pc = pc;  P[from].sp = sp;  /* save */
6    P[from].r1 = r1;  P[from].state = 'R';
7    pc = P[to].pc;  sp = P[to].sp;  /* load */
▶    r1 = P[to].r1;  P[to].state = 'X';
9  }
10 /* interrupt */  context_switch(0, 1);
11 /* ... P1 runs ... */
12 /* interrupt */  context_switch(1, 0);
```



r1 = 8: P_0 continues exactly where it was interrupted, at **pc = 104**, without noticing. Real CPUs have dozens of registers, and the address space changes too.

Step 11 / 11

VARIABLES

`pc = 104` `sp = 896` `r1 = 8`

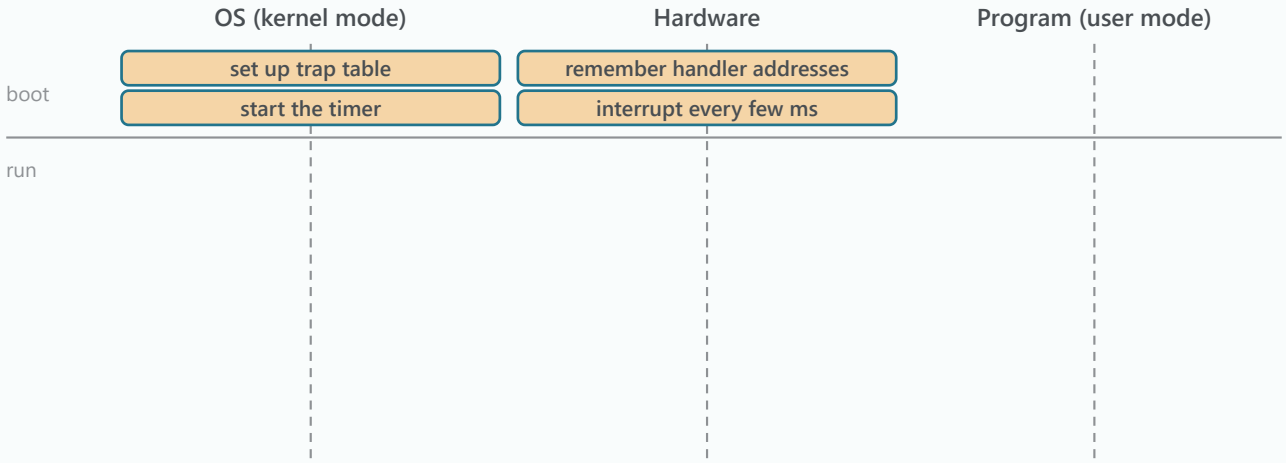
CALL STACK

`context_switch(1, 0)`

SCHEDULING

Limited Direct Execution

The program runs directly on the CPU; privileged work happens in the kernel, and the OS takes the CPU back **by force**.



At boot, in kernel mode: the OS tells the hardware which code runs on which event (the trap/interrupt table) and sets up the timer. Both are **privileged** instructions.

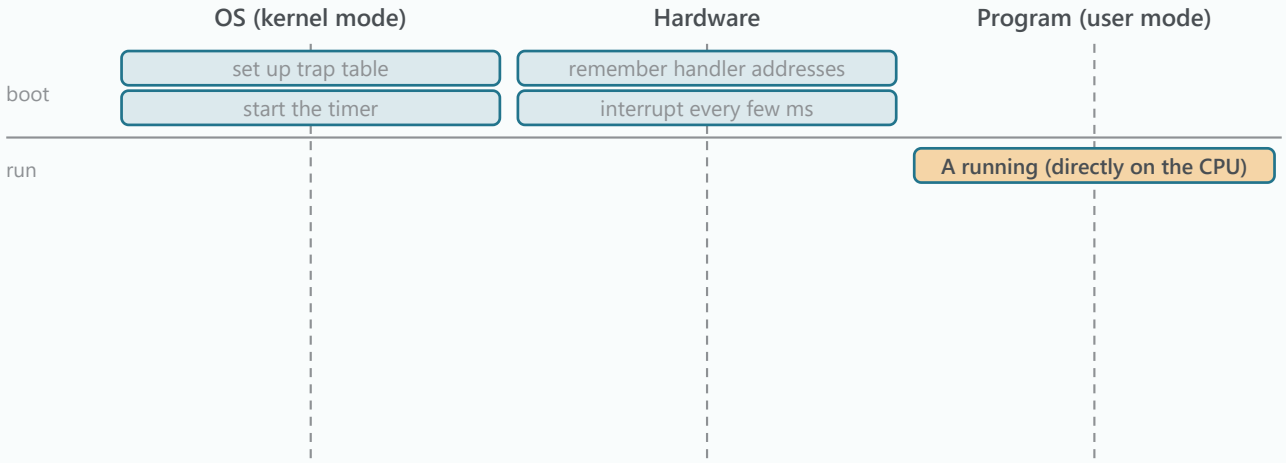
Step 1 / 5

Without the interrupt (**cooperative** multitasking, Windows 3.x), an infinite loop would hang the machine.

SCHEDULING

Limited Direct Execution

The program runs directly on the CPU; privileged work happens in the kernel, and the OS takes the CPU back **by force**.

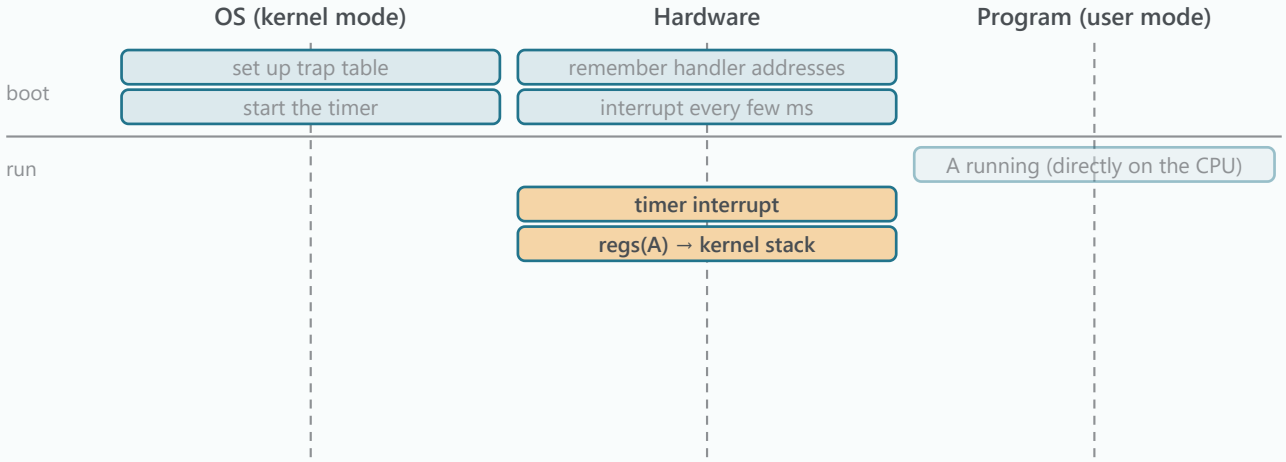


A runs **directly** on the CPU in user mode: no interpreter in between, full speed. For privileged work it must trap with a system call.

Without the interrupt (**cooperative** multitasking, Windows 3.x), an infinite loop would hang the machine.

Limited Direct Execution

The program runs directly on the CPU; privileged work happens in the kernel, and the OS takes the CPU back **by force**.

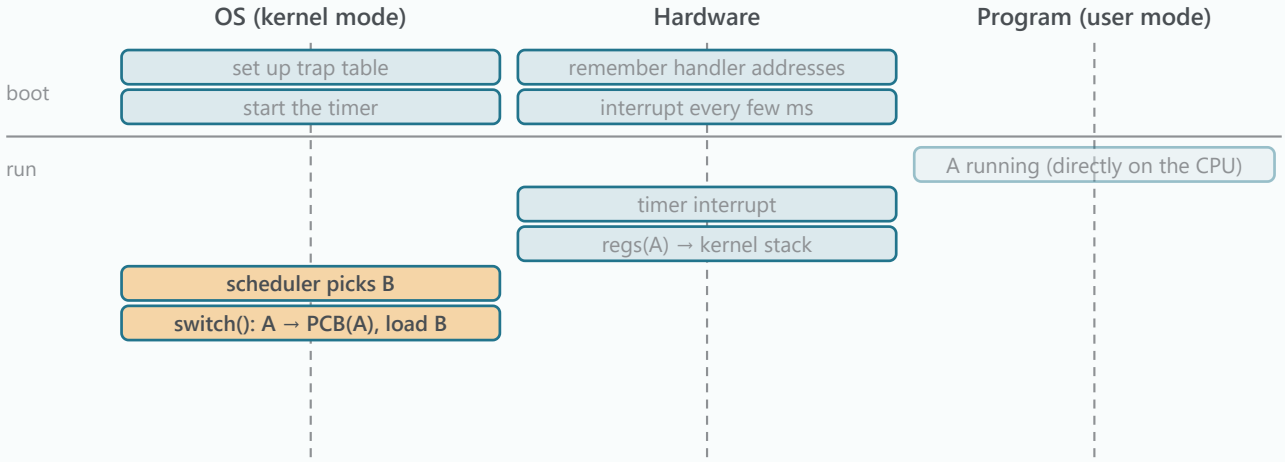


Even if A is in an infinite loop, the interrupt comes. The hardware saves A's registers, enters kernel mode and jumps to the OS handler: **the OS is in control**.

Without the interrupt (**cooperative** multitasking, Windows 3.x), an infinite loop would hang the machine.

Limited Direct Execution

The program runs directly on the CPU; privileged work happens in the kernel, and the OS takes the CPU back **by force**.

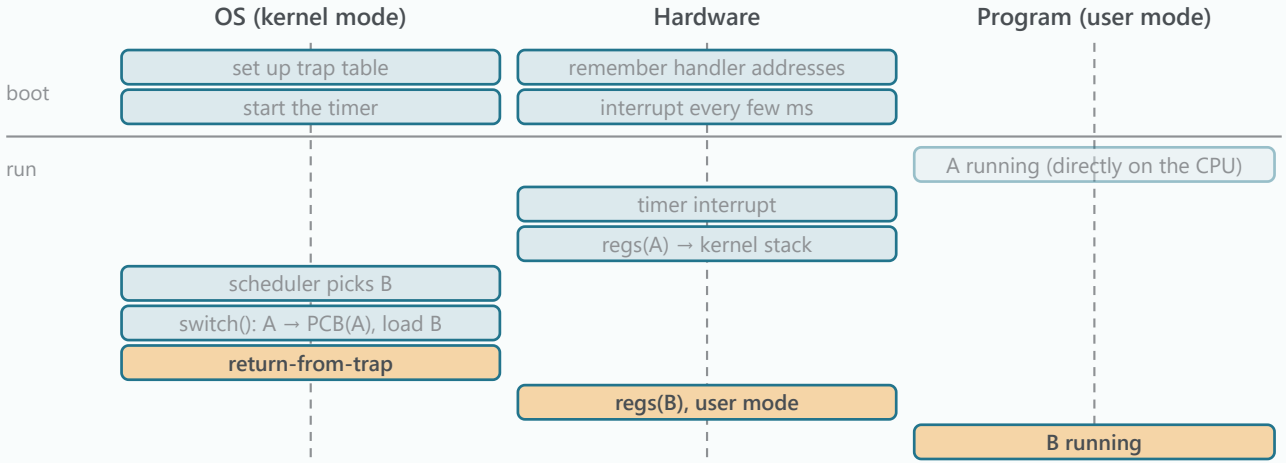


A's slice is up, so the OS decides to switch to B. `switch()` writes A's context to PCB(A), loads B's and moves to B's kernel stack.

Without the interrupt (**cooperative** multitasking, Windows 3.x), an infinite loop would hang the machine.

Limited Direct Execution

The program runs directly on the CPU; privileged work happens in the kernel, and the OS takes the CPU back **by force**.



Return-from-trap loads B's registers as if B were returning from where it was interrupted, and enters user mode: now **B** runs.

Without the interrupt (**cooperative** multitasking, Windows 3.x), an infinite loop would hang the machine.

The Cost of a Context Switch

- When the CPU switches to another process it must **save** the old process's state and **load** the new one's.
- This time is **pure overhead**: the system does no useful work meanwhile.
- The more complex the PCB and the OS, the longer a context switch takes.

Direct cost

Saving/loading registers, kernel code: typically **a few microseconds** on current Linux.

Indirect cost

Caches and the TLB “go cold”: the new process takes more cache misses at first; this often exceeds the direct cost.

Hardware support

Multiple register sets; address-space tags in the TLB (ASID / PCID) to avoid flushing.

Handling an Interrupt

What happens at the lowest level of the OS when an I/O interrupt arrives (Tanenbaum):

1. Hardware stacks the program counter, PSW, etc.

   The interrupt request is accepted at an instruction boundary; the CPU enters kernel mode.

Handling an Interrupt

What happens at the lowest level of the OS when an I/O interrupt arrives (Tanenbaum):

1. Hardware stacks the program counter, PSW, etc.

2. Hardware loads a new program counter from the interrupt vector.
   The vector is a table holding the handler address for each kind of interrupt.

Handling an Interrupt

What happens at the lowest level of the OS when an I/O interrupt arrives (Tanenbaum):

1. Hardware stacks the program counter, PSW, etc.

2. Hardware loads a new program counter from the interrupt vector.

3. An assembly routine saves the registers.
   The registers are written to the current process's PCB (or the kernel stack).

Handling an Interrupt

What happens at the lowest level of the OS when an I/O interrupt arrives (Tanenbaum):

1. Hardware stacks the program counter, PSW, etc.
2. Hardware loads a new program counter from the interrupt vector.
3. An assembly routine saves the registers.
4. An assembly routine sets up a new stack.

   The interrupt handler runs on the kernel stack.

Handling an Interrupt

What happens at the lowest level of the OS when an I/O interrupt arrives (Tanenbaum):

1. Hardware stacks the program counter, PSW, etc.
2. Hardware loads a new program counter from the interrupt vector.
3. An assembly routine saves the registers.
4. An assembly routine sets up a new stack.
5. The C interrupt service routine runs.
   It usually reads/buffers input and makes the waiting process ready.

Handling an Interrupt

What happens at the lowest level of the OS when an I/O interrupt arrives (Tanenbaum):

1. Hardware stacks the program counter, PSW, etc.
2. Hardware loads a new program counter from the interrupt vector.
3. An assembly routine saves the registers.
4. An assembly routine sets up a new stack.
5. The C interrupt service routine runs.
6. The scheduler picks the next process.

   If the woken process has higher priority, it may be chosen instead of the interrupted one.

Handling an Interrupt

What happens at the lowest level of the OS when an I/O interrupt arrives (Tanenbaum):

1. Hardware stacks the program counter, PSW, etc.
2. Hardware loads a new program counter from the interrupt vector.
3. An assembly routine saves the registers.
4. An assembly routine sets up a new stack.
5. The C interrupt service routine runs.
6. The scheduler picks the next process.
7. The C routine returns to the assembly code.

   What remains is loading the chosen process's context.

Handling an Interrupt

What happens at the lowest level of the OS when an I/O interrupt arrives (Tanenbaum):

1. Hardware stacks the program counter, PSW, etc.
2. Hardware loads a new program counter from the interrupt vector.
3. An assembly routine saves the registers.
4. An assembly routine sets up a new stack.
5. The C interrupt service routine runs.
6. The scheduler picks the next process.
7. The C routine returns to the assembly code.
8. The assembly routine starts the new process.

   Registers and the memory map are loaded and control returns to user mode.

Interprocess Communication

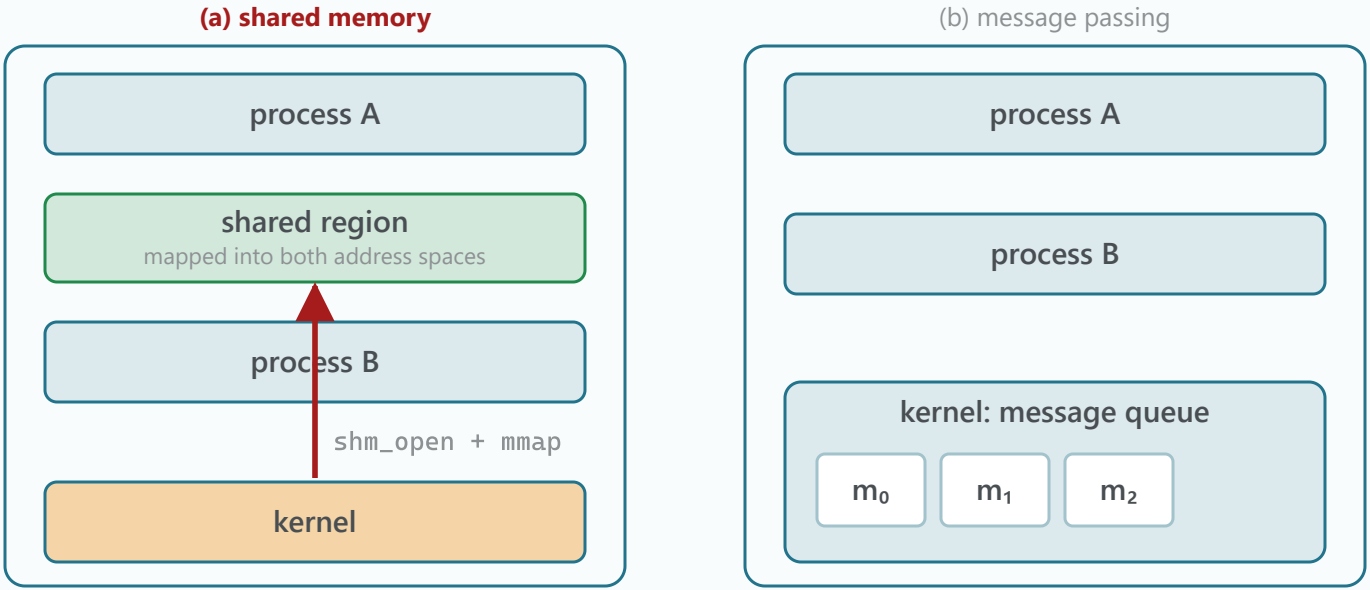
- Processes can be **independent** (unaffected by others) or **cooperating** (sharing data).
- Reasons to cooperate: information sharing, speedup (splitting the work), modularity, security and isolation (Chrome).
- Cooperating processes need an **IPC** (interprocess communication) mechanism.

Three basic questions

1. **How** does one process pass information to another?
2. How do we keep processes out of each other's way: how do we prevent **two bookings for the same seat**?
3. How do we ensure the **right order** when there are dependencies? (Don't fire before aiming; don't consume before producing.)

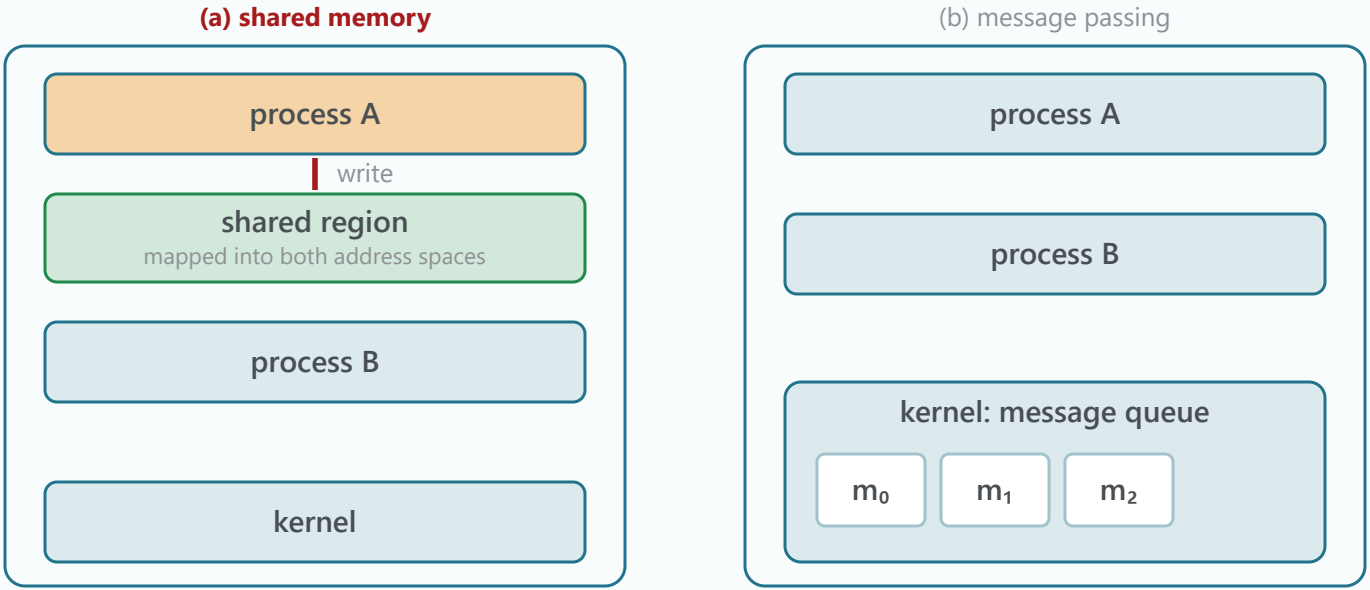
Two basic models: **shared memory** and **message passing**. The last two questions call for **synchronization**: uncontrolled access to shared data makes the result depend on execution order (a race condition). The same questions apply to threads.

Shared Memory and Message Passing



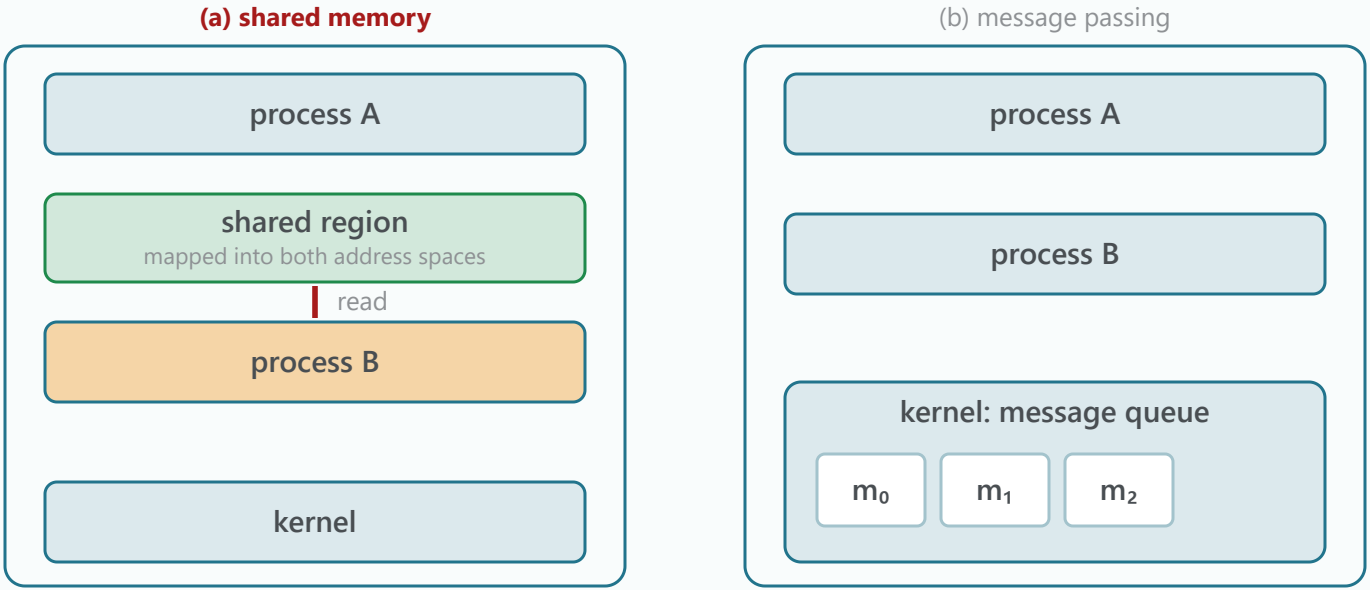
Shared memory: the kernel creates a region and maps it into both processes' address spaces (system calls only during setup).

Shared Memory and Message Passing



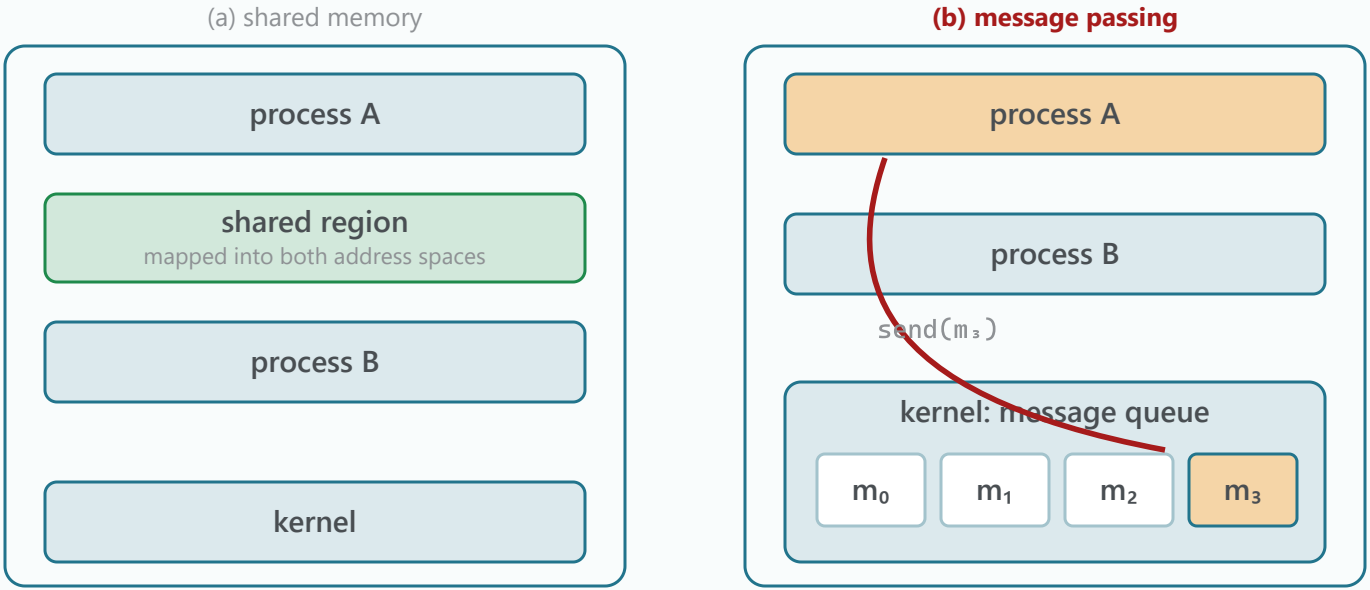
A writes directly into the region: the kernel is **not** involved, so this is very fast.

Shared Memory and Message Passing



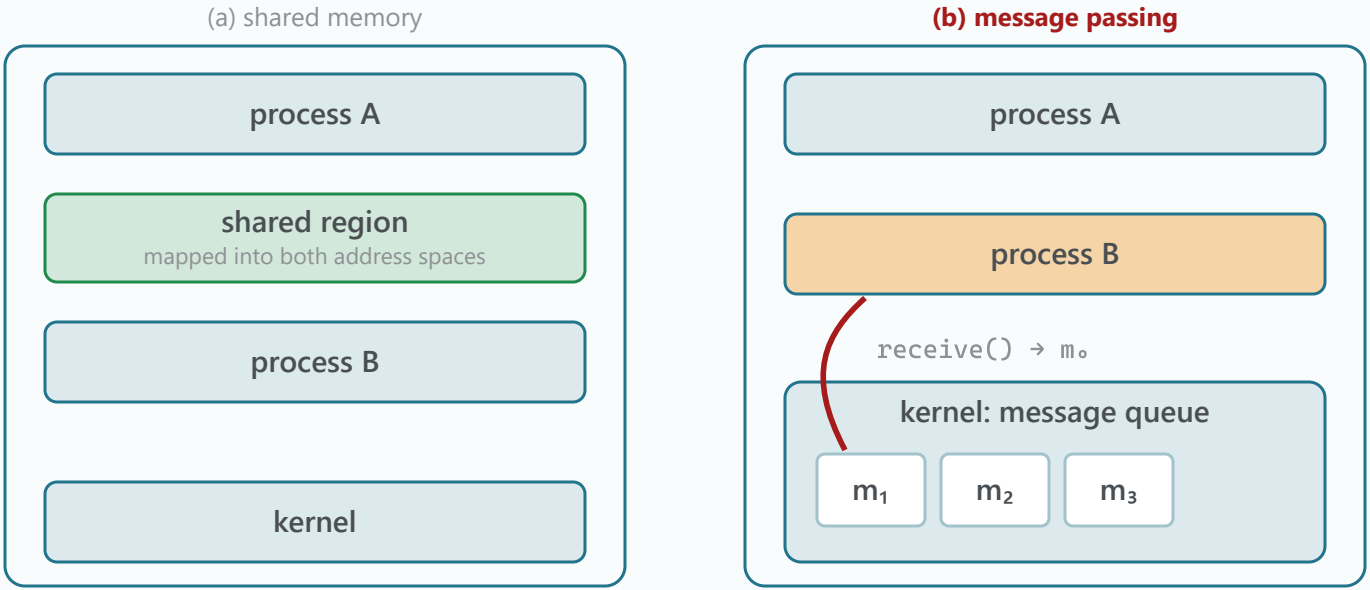
B reads the same data. But the processes must handle synchronization themselves (who writes when?).

Shared Memory and Message Passing



Message passing: A says `send(m₃)`; the kernel copies the message into its own queue.

Shared Memory and Message Passing



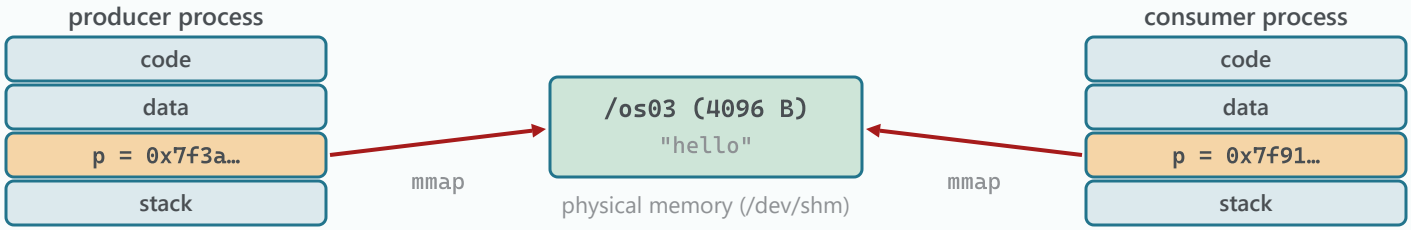
B takes the next message with `receive()`. Each message costs a system call and a copy, but the kernel provides synchronization, and it works over a network too.

POSIX Shared Memory

```c
/* producer */
int fd = shm_open("/os03", O_CREAT|O_RDWR, 0600);
ftruncate(fd, 4096);
char *p = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
               MAP_SHARED, fd, 0);
strcpy(p, "hello");     /* direct write */

/* consumer (another process) */
int fd = shm_open("/os03", O_RDONLY, 0);
char *p = mmap(NULL, 4096, PROT_READ,
               MAP_SHARED, fd, 0);
printf("%s\n", p);
shm_unlink("/os03");    /* remove */
```

- `shm_open` creates a named object (under `/dev/shm` on Linux).
- `ftruncate` sets the size, `mmap` maps it into the address space.
- After that it is ordinary memory access.
- If the consumer reads before the producer writes, it sees nothing: real code needs **synchronization** such as a semaphore.
- The older System V API: `shmget` / `shmat`.



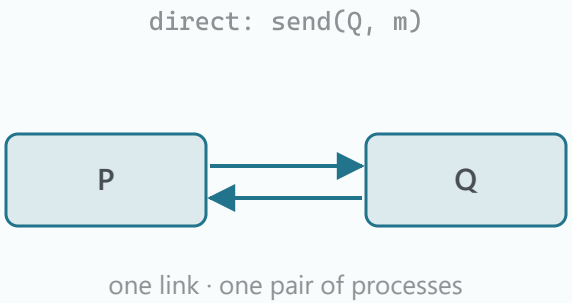
Message-Passing Design

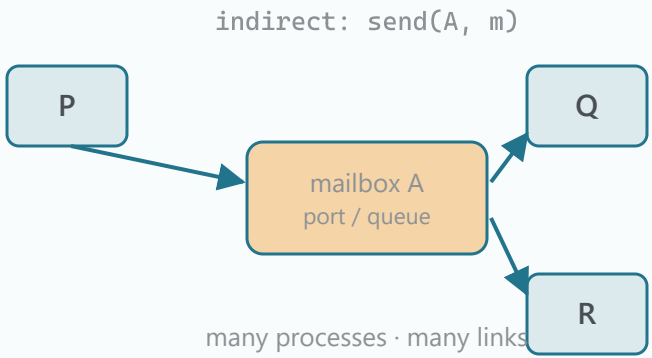
Two basic operations: `send(message)` and `receive(message)`. First a **communication link** must be set up.

Aspect	Options
Physical link	shared memory, hardware bus, network
Naming	**direct** (process name) · **indirect** (mailbox / port)
Synchronous / asynchronous	**blocking (synchronous)**: the sender waits until the message is received, the receiver until one arrives · **nonblocking (asynchronous)**: returns at once
Buffering	**zero capacity** (sender waits: rendezvous) · **bounded** (waits if the queue is full) · **unbounded** (never waits)

Blocking send + blocking receive = **rendezvous**: the producer–consumer problem solves itself. Examples: POSIX `mq_send` / `mq_receive`, Mach ports, Windows ALPC.

Direct and Indirect Communication





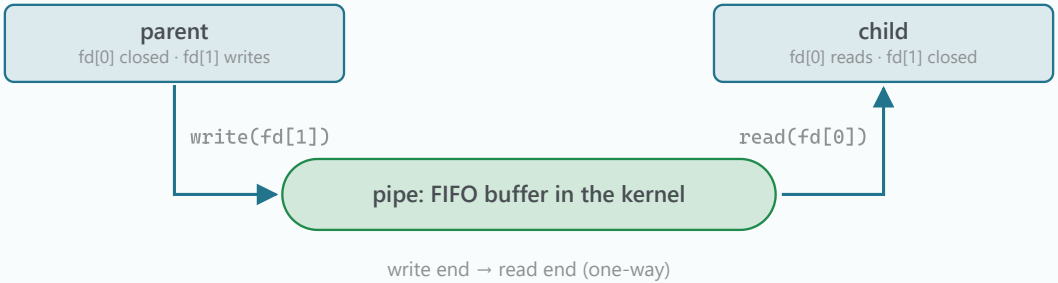
Direct

- `send(P, m)` : send to P; `receive(Q, m)` : receive from Q.
- The link is set up automatically and belongs to exactly one pair of processes.
- One link between each pair; usually bidirectional.

Indirect (mailbox)

- `send(A, m)` / `receive(A, m)` : through mailbox A.
- Each mailbox has a unique ID; processes sharing a mailbox can communicate.
- A link may involve many processes, and a pair may have many links; uni- or bidirectional.

Pipes



Design questions

- One-way (simplex) or two-way?
- If two-way, half or full duplex?
- Must the processes be parent and child?
- Can it be used over a network?

	Ordinary (anonymous)	**Named (FIFO)**
Access	only the creator and its children	any process that knows the file name
Direction	one-way	UNIX: half duplex; Windows: full duplex
Lifetime	ends when the processes close it	`mkfifo`; stays in the file system
Network	no	UNIX no; Windows yes

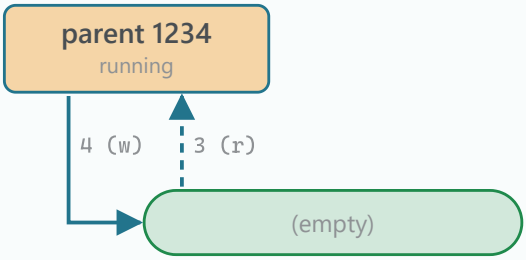
Code Trace: pipe() + fork()

```
1  int main(void) {
2    int fd[2];
3    char buf[32];
▶    pipe(fd);
5    pid_t pid = fork();
6    if (pid == 0) {    /* child: reads */
7      close(fd[1]);
8      ssize_t n = read(fd[0], buf, sizeof buf -
   1);
9      buf[n] = '\0';
10     printf("child got: %s\n", buf);
11     close(fd[0]);
12     return 0;
13   }
14   close(fd[0]);      /* parent: writes */
15   write(fd[1], "welcome", 7);
16   close(fd[1]);
17   wait(NULL);
18   return 0;
19 }
```

VARIABLES

`fd[0] = 3` `fd[1] = 4`

OUTPUT



`pipe(fd)` : the kernel opens an empty buffer; read end `fd[0]=3` , write end `fd[1]=4` (0–2 are stdin/stdout/stderr).

Code Trace: pipe() + fork()

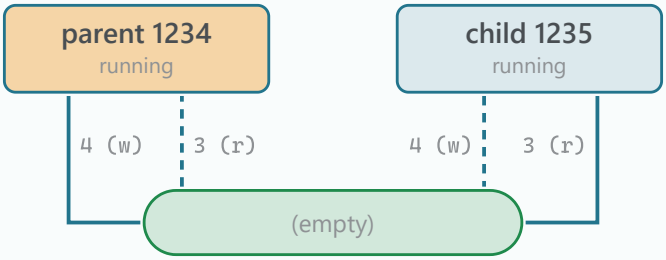
```
1  int main(void) {
2    int fd[2];
3    char buf[32];
4    pipe(fd);
▶    pid_t pid = fork();
6    if (pid == 0) {    /* child: reads */
7      close(fd[1]);
8      ssize_t n = read(fd[0], buf, sizeof buf -
   1);
9      buf[n] = '\0';
10     printf("child got: %s\n", buf);
11     close(fd[0]);
12     return 0;
13   }
14   close(fd[0]);      /* parent: writes */
15   write(fd[1], "welcome", 7);
16   close(fd[1]);
17   wait(NULL);
18   return 0;
19 }
```

VARIABLES

`fd[0] = 3` `fd[1] = 4`

`parent.pid = 1235` `child.pid = 0`

OUTPUT



`fork()` : the child inherits the descriptors; both ends are now open in both processes.

Code Trace: pipe() + fork()

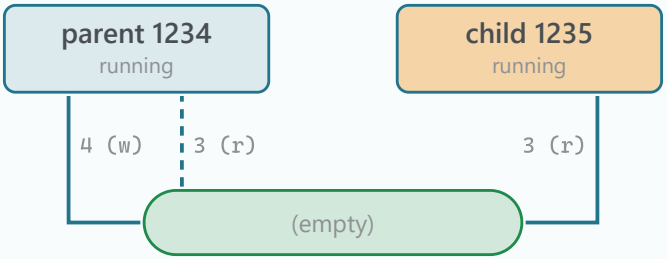
```
1  int main(void) {
2    int fd[2];
3    char buf[32];
4    pipe(fd);
5    pid_t pid = fork();
6    if (pid == 0) {    /* child: reads */
▶      close(fd[1]);
8      ssize_t n = read(fd[0], buf, sizeof buf -
   1);
9      buf[n] = '\0';
10     printf("child got: %s\n", buf);
11     close(fd[0]);
12     return 0;
13   }
14   close(fd[0]);      /* parent: writes */
15   write(fd[1], "welcome", 7);
16   close(fd[1]);
17   wait(NULL);
18   return 0;
19 }
```

VARIABLES

`fd[0] = 3` `fd[1] = 4`

`parent.pid = 1235` `child.pid = 0`

OUTPUT



The child closes the write end it will not use.

Code Trace: pipe() + fork()

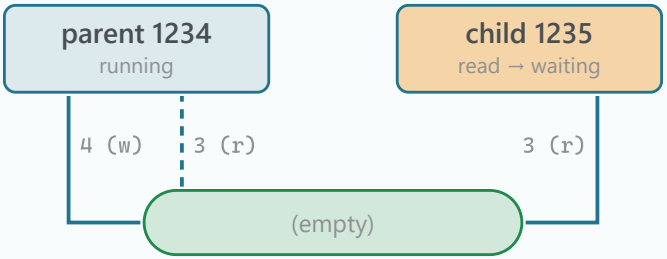
```
1  int main(void) {
2    int fd[2];
3    char buf[32];
4    pipe(fd);
5    pid_t pid = fork();
6    if (pid == 0) {   /* child: reads */
7      close(fd[1]);
▶      ssize_t n = read(fd[0], buf, sizeof buf -
1);
9      buf[n] = '\0';
10     printf("child got: %s\n", buf);
11     close(fd[0]);
12     return 0;
13   }
14   close(fd[0]);     /* parent: writes */
15   write(fd[1], "welcome", 7);
16   close(fd[1]);
17   wait(NULL);
18   return 0;
19 }
```

VARIABLES

`fd[0] = 3` `fd[1] = 4`

`parent.pid = 1235` `child.pid = 0`

OUTPUT



The pipe is empty: `read` **blocks** the child.

Code Trace: pipe() + fork()

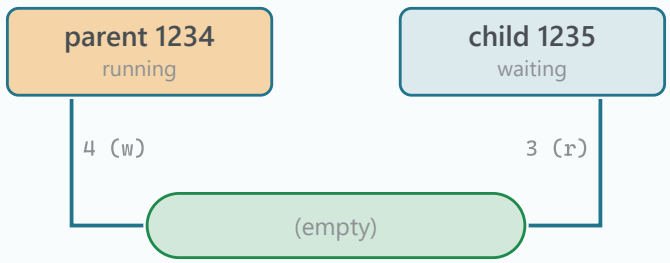
```
1  int main(void) {
2    int fd[2];
3    char buf[32];
4    pipe(fd);
5    pid_t pid = fork();
6    if (pid == 0) {    /* child: reads */
7      close(fd[1]);
8      ssize_t n = read(fd[0], buf, sizeof buf -
   1);
9      buf[n] = '\0';
10     printf("child got: %s\n", buf);
11     close(fd[0]);
12     return 0;
13   }
▶  close(fd[0]);      /* parent: writes */
15   write(fd[1], "welcome", 7);
16   close(fd[1]);
17   wait(NULL);
18   return 0;
19 }
```

VARIABLES

`fd[0] = 3` `fd[1] = 4`

`parent.pid = 1235` `child.pid = 0`

OUTPUT



The parent closes the read end.

Code Trace: pipe() + fork()

```
1  int main(void) {
2    int fd[2];
3    char buf[32];
4    pipe(fd);
5    pid_t pid = fork();
6    if (pid == 0) {    /* child: reads */
7      close(fd[1]);
8      ssize_t n = read(fd[0], buf, sizeof buf -
   1);
9      buf[n] = '\0';
10     printf("child got: %s\n", buf);
11     close(fd[0]);
12     return 0;
13   }
14   close(fd[0]);      /* parent: writes */
▶    write(fd[1], "welcome", 7);
16   close(fd[1]);
17   wait(NULL);
18   return 0;
19 }
```

VARIABLES

`fd[0] = 3` `fd[1] = 4`

`parent.pid = 1235` `child.pid = 0`

OUTPUT



The parent writes 7 bytes; writes of ≤ PIPE_BUF are not split (atomic). The waiting reader wakes up.

Code Trace: pipe() + fork()

```
1  int main(void) {
2    int fd[2];
3    char buf[32];
4    pipe(fd);
5    pid_t pid = fork();
6    if (pid == 0) {    /* child: reads */
7      close(fd[1]);
8      ssize_t n = read(fd[0], buf, sizeof buf -
   1);
9      buf[n] = '\0';
10     printf("child got: %s\n", buf);
11     close(fd[0]);
12     return 0;
13   }
14   close(fd[0]);      /* parent: writes */
15   write(fd[1], "welcome", 7);
▶    close(fd[1]);
▶    wait(NULL);
18   return 0;
19 }
```

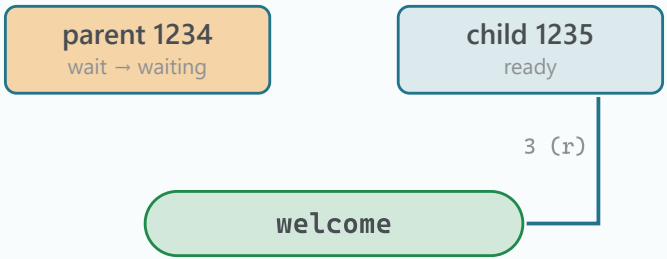
VARIABLES

`fd[0] = 3` `fd[1] = 4`

`parent.pid = 1235` `child.pid = 0`

OUTPUT





The parent closes the write end (so the reader can later see EOF) and waits with `wait`.

Code Trace: pipe() + fork()

```
1  int main(void) {
2    int fd[2];
3    char buf[32];
4    pipe(fd);
5    pid_t pid = fork();
6    if (pid == 0) {    /* child: reads */
7      close(fd[1]);
▶      ssize_t n = read(fd[0], buf, sizeof buf -
1);
9      buf[n] = '\0';
10     printf("child got: %s\n", buf);
11     close(fd[0]);
12     return 0;
13   }
14   close(fd[0]);      /* parent: writes */
15   write(fd[1], "welcome", 7);
16   close(fd[1]);
17   wait(NULL);
18   return 0;
19 }
```

VARIABLES

`n = 7` `buf = "welcome"` `child.pid = 0`

OUTPUT



`read` returns 7; the data is copied from the buffer into the child's `buf` array.

Code Trace: pipe() + fork()

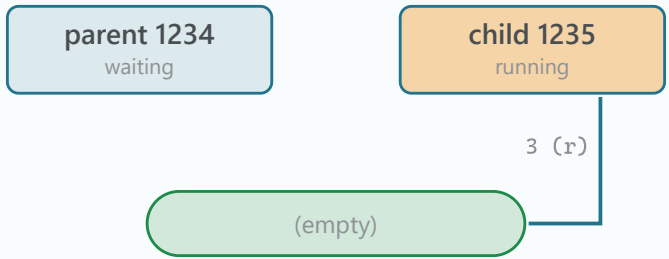
```
1  int main(void) {
2    int fd[2];
3    char buf[32];
4    pipe(fd);
5    pid_t pid = fork();
6    if (pid == 0) {    /* child: reads */
7      close(fd[1]);
8      ssize_t n = read(fd[0], buf, sizeof buf -
   1);
▶      buf[n] = '\0';
▶      printf("child got: %s\n", buf);
11     close(fd[0]);
12     return 0;
13   }
14   close(fd[0]);      /* parent: writes */
15   write(fd[1], "welcome", 7);
16   close(fd[1]);
17   wait(NULL);
18   return 0;
19 }
```

VARIABLES

`n = 7` `buf = "welcome\0"` `child.pid = 0`

OUTPUT

```
child got: welcome
```



A terminating `'\0'` is added and the string is printed.

Code Trace: pipe() + fork()

```
 1 int main(void) {
 2   int fd[2];
 3   char buf[32];
 4   pipe(fd);
 5   pid_t pid = fork();
 6   if (pid == 0) {    /* child: reads */
 7     close(fd[1]);
 8     ssize_t n = read(fd[0], buf, sizeof buf -
   1);
 9     buf[n] = '\0';
10     printf("child got: %s\n", buf);
11     close(fd[0]);
12     return 0;
13   }
14   close(fd[0]);      /* parent: writes */
15   write(fd[1], "welcome", 7);
16   close(fd[1]);
▶  wait(NULL);
▶  return 0;
19 }
```

VARIABLES

`parent.pid = 1235`

OUTPUT

```
child got: welcome
```

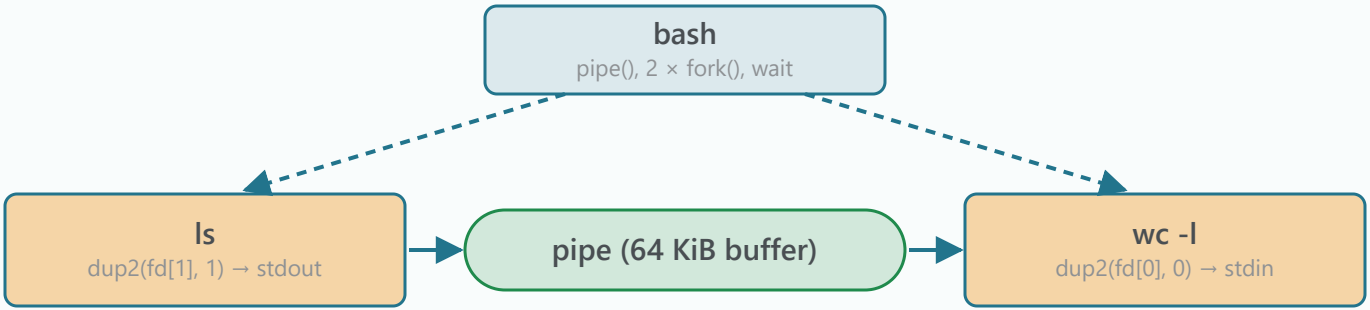
parent 1234
running

child 1235
done

(empty)

The child closes the read end and exits; the parent returns from `wait` and finishes. Since only the child prints, the output is deterministic.

A Shell Pipeline: ls | wc -l

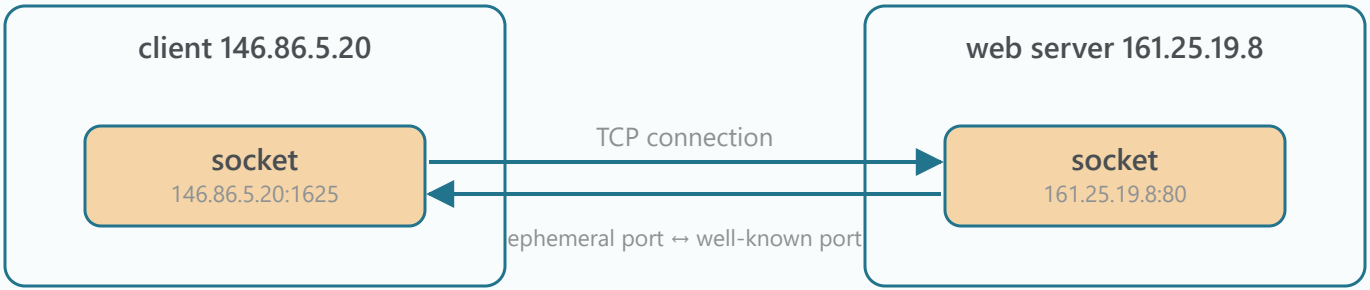


both processes must close the ends they don't use, or wc never sees EOF

```
int fd[2]; pipe(fd);
if (fork() == 0) { dup2(fd[1], 1); close(fd[0]); close(fd[1]);
                   execlp("ls", "ls", NULL); }
if (fork() == 0) { dup2(fd[0], 0); close(fd[0]); close(fd[1]);
                   execlp("wc", "wc", "-l", NULL); }
close(fd[0]); close(fd[1]); wait(NULL); wait(NULL);
```

Error checks omitted for brevity. `dup2(old, new)` makes the new descriptor a copy of the old one; programs work with stdin/stdout, so they never notice the pipe.

Sockets



- **Socket:** an endpoint for communication, identified by IP address + port (`161.25.19.8:80`).
- Communication takes place between **a pair of sockets**; the port identifies the service on the machine.
- Ports below 1024 are reserved for **well-known** services (SSH 22, HTTP 80, HTTPS 443); binding them as a server requires privilege.
- `127.0.0.1` (IPv6: `::1`) is the loopback address: processes on the same machine.
- **TCP** is a connection-oriented, ordered, reliable byte stream; **UDP** is connectionless datagrams.
- On the same machine, **UNIX domain sockets** (`AF_UNIX`) are faster; Docker, systemd and X/Wayland use them.

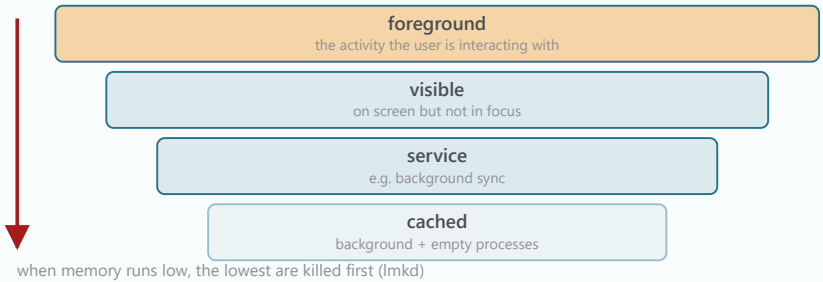
Multitasking on Mobile Systems

iOS / iPadOS

- Early versions (2007) allowed only one user app at a time; iOS 4 (2010) brought limited multitasking.
- Only permitted work runs in the background: audio playback, location, VoIP, `BGTaskScheduler` tasks; everything else is suspended.

Android

- Allows background processes; long work continues in a **service** component even while the UI is suspended.
- Background limits since Android 8: a **foreground service** with a notification, or `WorkManager`, for long work.



The limits exist because of scarce memory and battery. The “background” and “empty” processes of older docs are now grouped as **cached**; when memory runs low, `lmkd` kills from the bottom up.

Summary

Process

A program in execution: text, data, heap, stack + CPU context + resources. The kernel tracks it with a PCB.

Life cycle

New → Ready ⇄ Running → Waiting → Ready … → Terminated. The exit status is collected with `wait`.

UNIX API

`fork` copies (COW), `exec` replaces the image, `wait` waits and reaps, `exit` ends.

Scheduling machinery

Ready/wait queues; a context switch is pure overhead.

IPC

Shared memory is fast but needs synchronization; message passing, pipes and sockets go through the kernel.

Mobile multitasking

Because of scarce memory and battery, background processes are suspended and killed if needed.

SUMMARY

Review Questions

1. How many processes in total does a program running `fork(); fork();` end up with? And `for (i = 0; i < 3; i++) fork();`?
2. What is the output of `printf("A"); fork(); printf("B\n");` when stdout is redirected to a *file*? Why?
3. Can a process go directly from Waiting to Running?
4. What is the difference between a zombie and an orphan? Which causes more trouble for the system?
5. If `exec` succeeds, when does the line after it run?
6. What is the CPU utilization for p = 0.5 and n = 3?
7. In the pipe example, what would happen if the parent did not `close(fd[1])` and the child read until EOF?

▶ **Show answers**

References

- A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10th ed., Wiley, 2018 — Chapter 3: Processes.
- A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4th ed. (5th ed. 2023), Pearson — Chapter 2: Processes and Threads.
- R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), pages.cs.wisc.edu/~remzi/OSTEP — Processes, Process API.
- Linux man pages: `fork(2)`, `execve(2)`, `wait(2)`, `pipe(7)`, `signal(7)`, `shm_overview(7)`.
- The Chromium Projects, “Multi-process Architecture” and “Site Isolation”; Android Developers, “Processes and app lifecycle”.

The End

Chapter 3: Processes

Previous: Chapter 2: Operating-System Structures · Next: Chapter 4: Threads