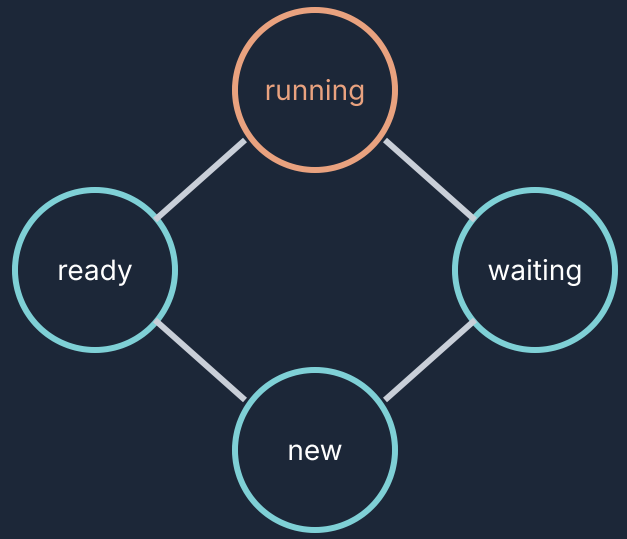
OPERATING SYSTEMS · LECTURE 3

Processes

States · PCB · scheduling · creation · IPC



Lecturer: Aidana Karibayeva

01 What is a process?

A **process** is a program **in execution** — the unit of work in a system. It needs CPU time, memory, files and I/O devices.

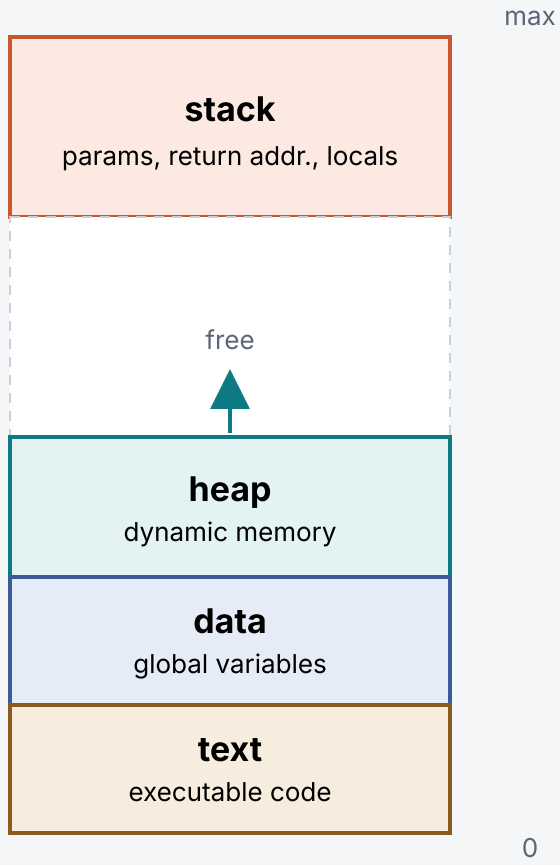
Program

passive: an executable file on disk (`prog.exe`, `a.out`)

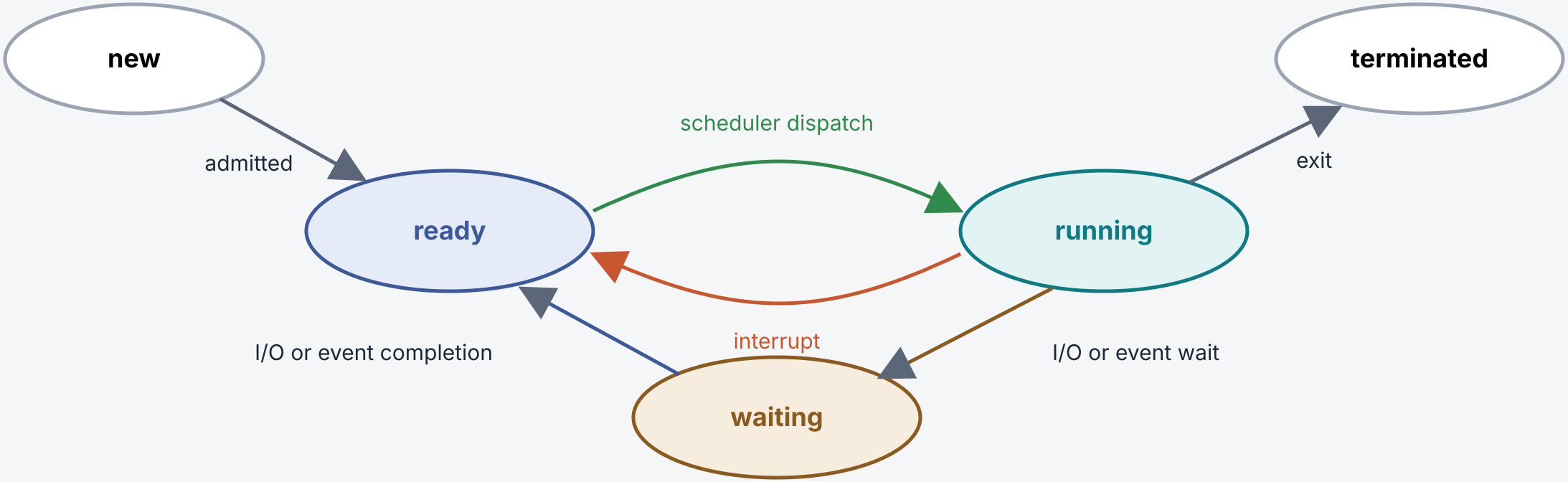
Process

active: the program loaded into memory, with a program counter and registers

> Two processes may run the **same program** — they are still two separate execution sequences. OS processes run system code, user processes run user code, all concurrently.

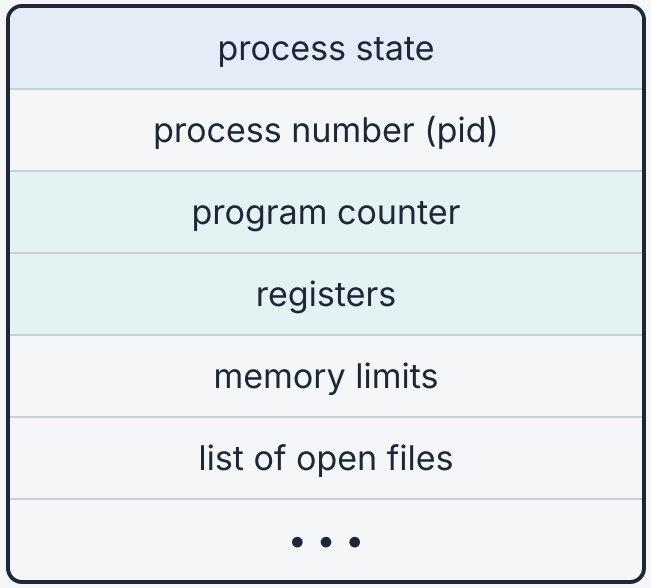


02 Process states



> Only **one** process can be **running** on a CPU core at a time; many can be ready or waiting.

03 Process Control Block (PCB)

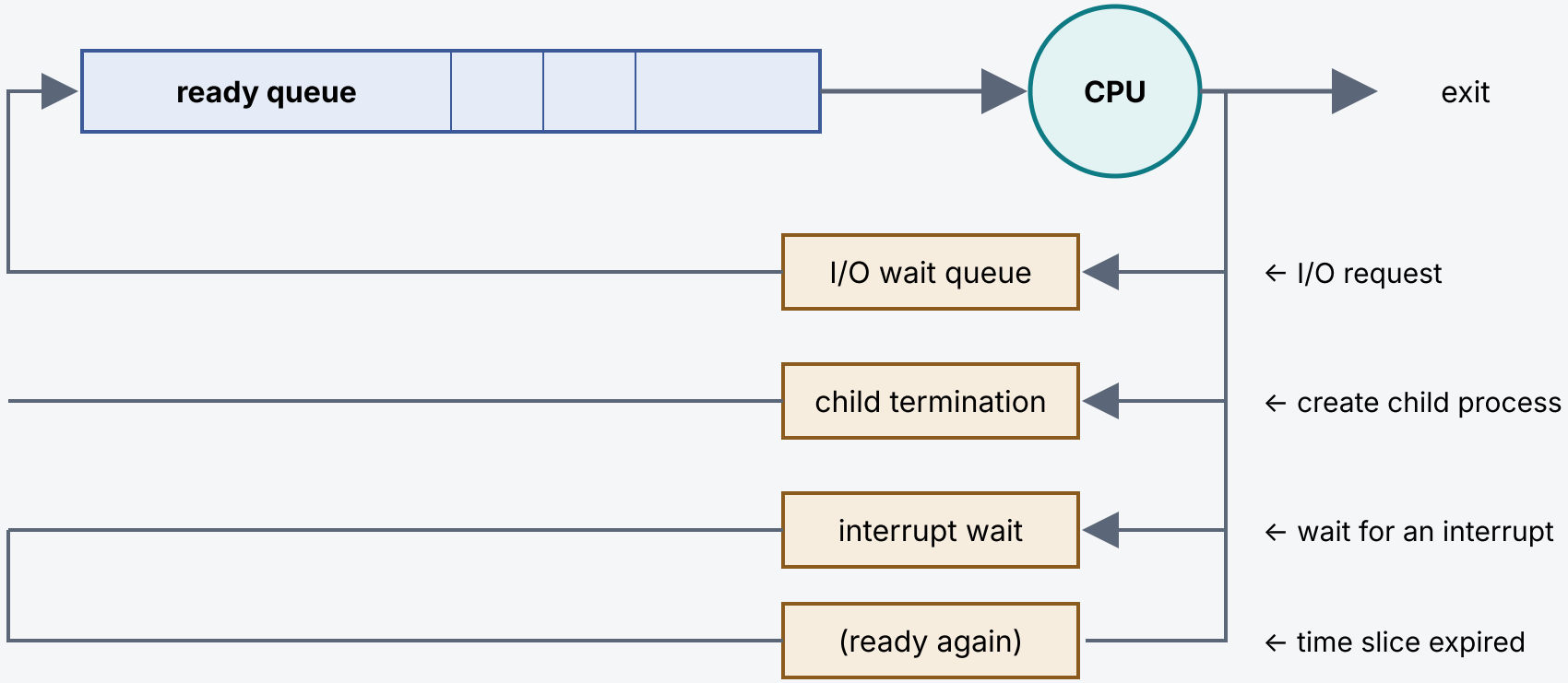


Process state	new, ready, running, waiting, halted...
Program counter	address of the next instruction
CPU registers	accumulators, index, stack pointers — saved on interrupt
CPU-scheduling info	priority, pointers to scheduling queues
Memory-management info	base/limit registers, page or segment tables
Accounting info	CPU time used, time limits, process numbers
I/O status info	allocated devices, open files

04

Process scheduling: queues

Multiprogramming keeps some process running at all times; **time sharing** switches the core often so users can interact.



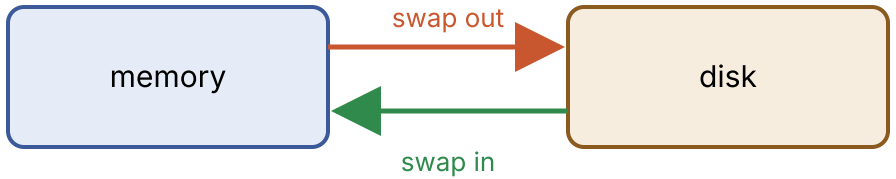
05 CPU scheduler and swapping

CPU scheduler

Picks a process from the **ready queue** and gives it a CPU core. Runs very often — at least every **100 ms**, usually more.

I/O-bound processes need the CPU for a few ms before waiting for I/O.

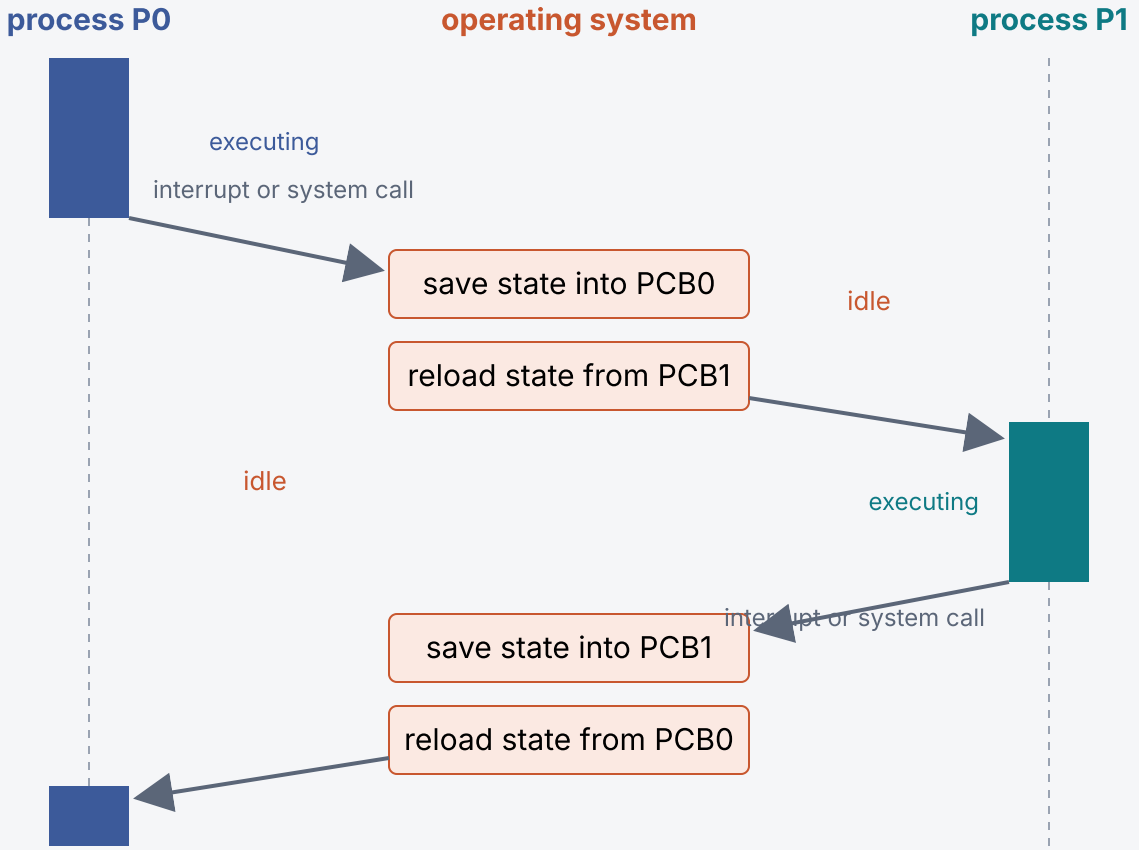
Swapping



Removes a process from memory to **lower the degree of multiprogramming**; later it continues where it stopped.

06

Context switch

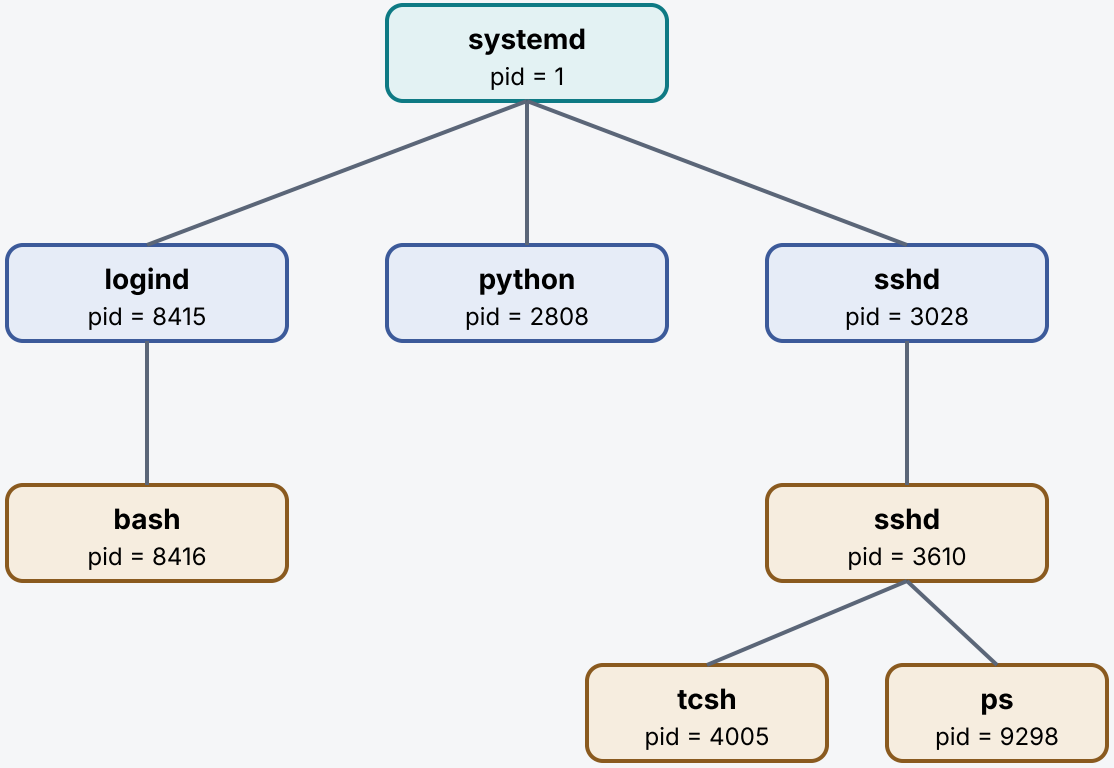


Switching the core to another process = **save** the current state into its PCB + **restore** the next one.

> Context-switch time is **pure overhead** — the system does no useful work while switching (a few microseconds).

07

Process creation: a tree of processes



A **parent** process creates **children**, which create their own — a tree. Each process has a unique **pid**.

```
$ ps -el
```

> Lists all active processes on UNIX / Linux.

08 Parent and child: the options

Resources

- child gets them from the OS, or
- a **subset** of the parent's resources

limits stop a process from overloading the system with children

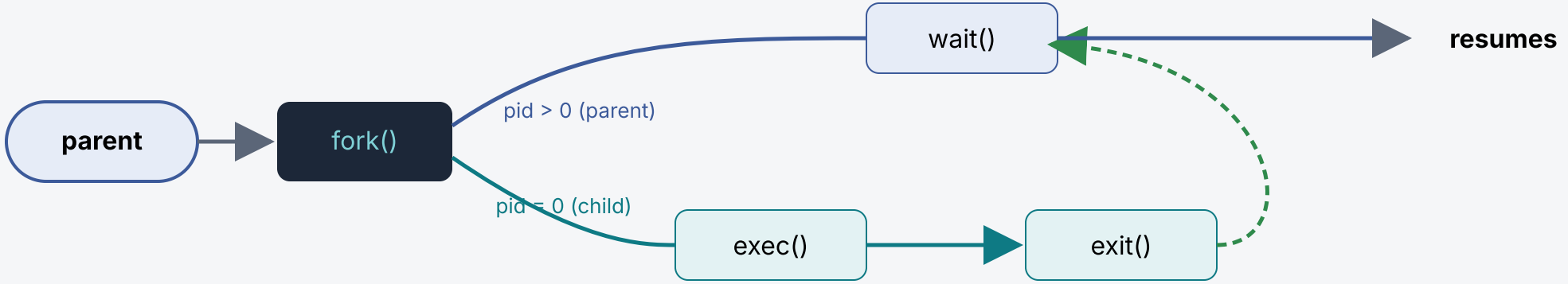
Execution

- parent runs **concurrently** with children, or
- parent **waits** until some/all children end

Address space

- child is a **duplicate** of the parent, or
- child **loads a new program**

09 Creating a process in UNIX: fork() and exec()




```c
pid_t pid = fork();
if (pid < 0) {          /* error */
   fprintf(stderr, "Fork Failed");
} else if (pid == 0) {  /* child */
   execlp("/bin/ls", "ls", NULL);
} else {                /* parent */
   wait(NULL);
   printf("Child Complete");
}
```

`fork()`	new process = **copy** of the parent's address space; both continue after fork
`exec()`	replaces the memory with a **new program**
`wait()`	parent waits for the child to finish

10 Creating a process in Windows

```
STARTUPINFO si;  PROCESS_INFORMATION pi;
ZeroMemory(&si, sizeof(si));
si.cb = sizeof(si);
ZeroMemory(&pi, sizeof(pi));

if (!CreateProcess(NULL,
      "C:\\WINDOWS\\system32\\mspaint.exe",
      NULL, NULL, FALSE, 0, NULL, NULL,
      &si, &pi))
   return -1;

WaitForSingleObject(pi.hProcess, INFINITE);
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
```

	UNIX	WINDOWS
Create	`fork()`	`CreateProcess()`
Child program	copy of parent, then `exec()`	new program loaded **at creation**
Parameters	none	at least 10
Wait	`wait()`	`WaitForSingleObject()`

11

Process termination

Normal end

last statement → `exit();` status returned to the parent through `wait()`. All memory, files and buffers are freed.

Parent may kill a child when

- it uses too many resources
- its task is no longer needed
- the parent itself is exiting

Cascading termination

some systems end **all children** when the parent ends

Zombie

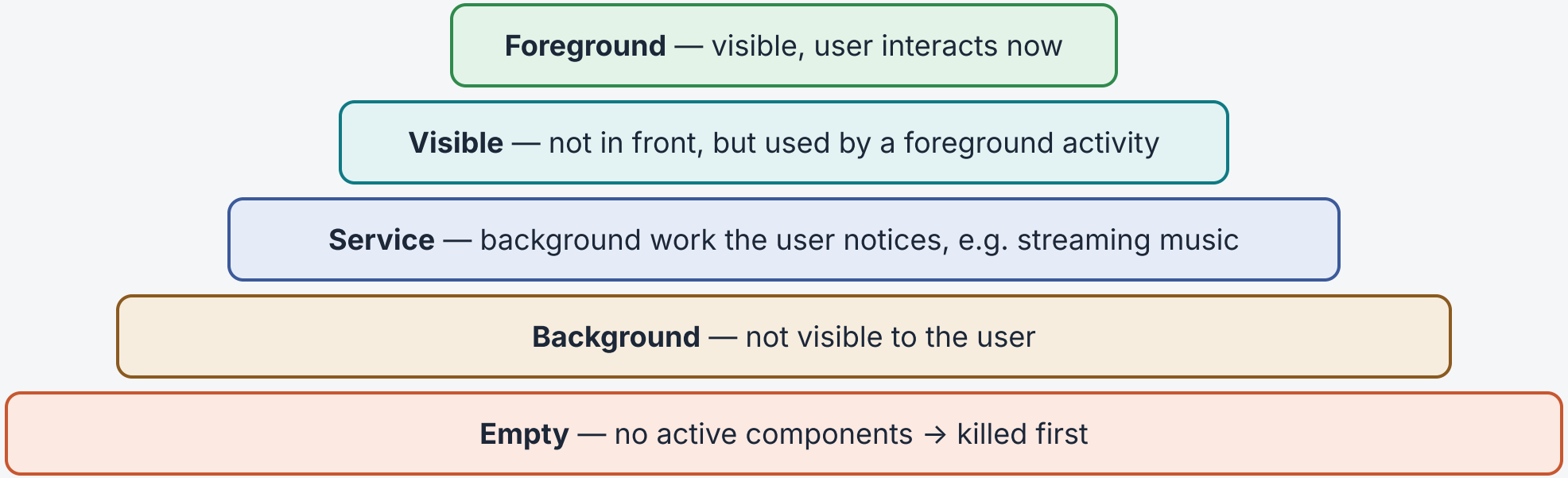
child ended, but parent has not called `wait()` yet — its entry stays in the process table

Orphan

parent ended without `wait();` Linux gives the child to **systemd / init**, which collects it

12 Android: importance hierarchy

Mobile systems may kill processes to free memory — **least important first**.



13 Cooperating processes

Independent

shares no data with other processes

Cooperating

can affect or be affected by others — needs **IPC**

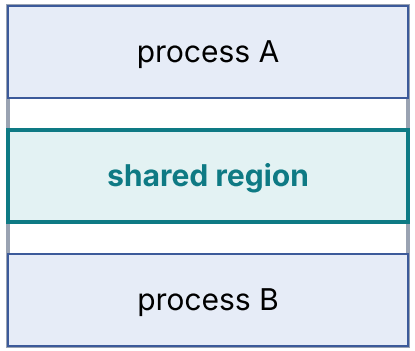
Information sharing
e.g. copy and paste

Computation speedup
subtasks run in parallel on many cores

Modularity
system split into separate processes

14 IPC: shared memory and message passing

Shared memory



region lives in the creator's address space; others **attach** to it. Producer–consumer with a buffer. Fast, but processes synchronise themselves.

Message passing

Direct	`send(P, message)` `receive(Q, message)`
Indirect	via a **mailbox (port)**: `send(A, message)`

No shared address space — useful in **distributed** systems, e.g. an Internet chat. A link is set up automatically between a pair of processes.

15 Summary

TOPIC	KEY IDEA
Process	program in execution: text, data, heap, stack
States	new → ready ⇄ running → terminated; running → waiting → ready
PCB	everything the OS knows about a process
Scheduling	ready and wait queues; CPU scheduler; swapping; context switch
Creation	process tree; UNIX `fork / exec / wait`; Windows `CreateProcess`
Termination	`exit()`; cascading; zombie and orphan
IPC	shared memory or message passing