Processes

Operating Systems
Baochun Li
University of Toronto

Running multiple programs

Why do we need to run multiple programs concurrently?

Called “multiprogramming”

Because it increases CPU utilization

I/O intensive programs are waiting for I/O most of the time, it would be good to keep the CPU busy with other tasks

Multiprogramming and time sharing

Multiprogramming: accommodating multiple processes in one physical address space

- Each process can be **I/O bound** or **CPU bound**
- It would be good to have a mix of **I/O bound** and **CPU bound** processes
- The goal is to increase **CPU utilization**
- A scheduler decides which process to execute

Time sharing (or “multitasking”): switching back-and-forth across processes very quickly — called “context switch”

- The goal is to reduce latency when a user interacts with the computer

The process abstraction

Program is a passive entity

- Usually stored in an executable file in a file system
- Contains instructions and static data values

Process is a program in execution (or a "running program")

But what constitutes a process?

We need to understand its execution context (environment)

- What a program can read or update when it's running

One obvious component: Memory

- The memory that a process can address is called its **address space**

What else?

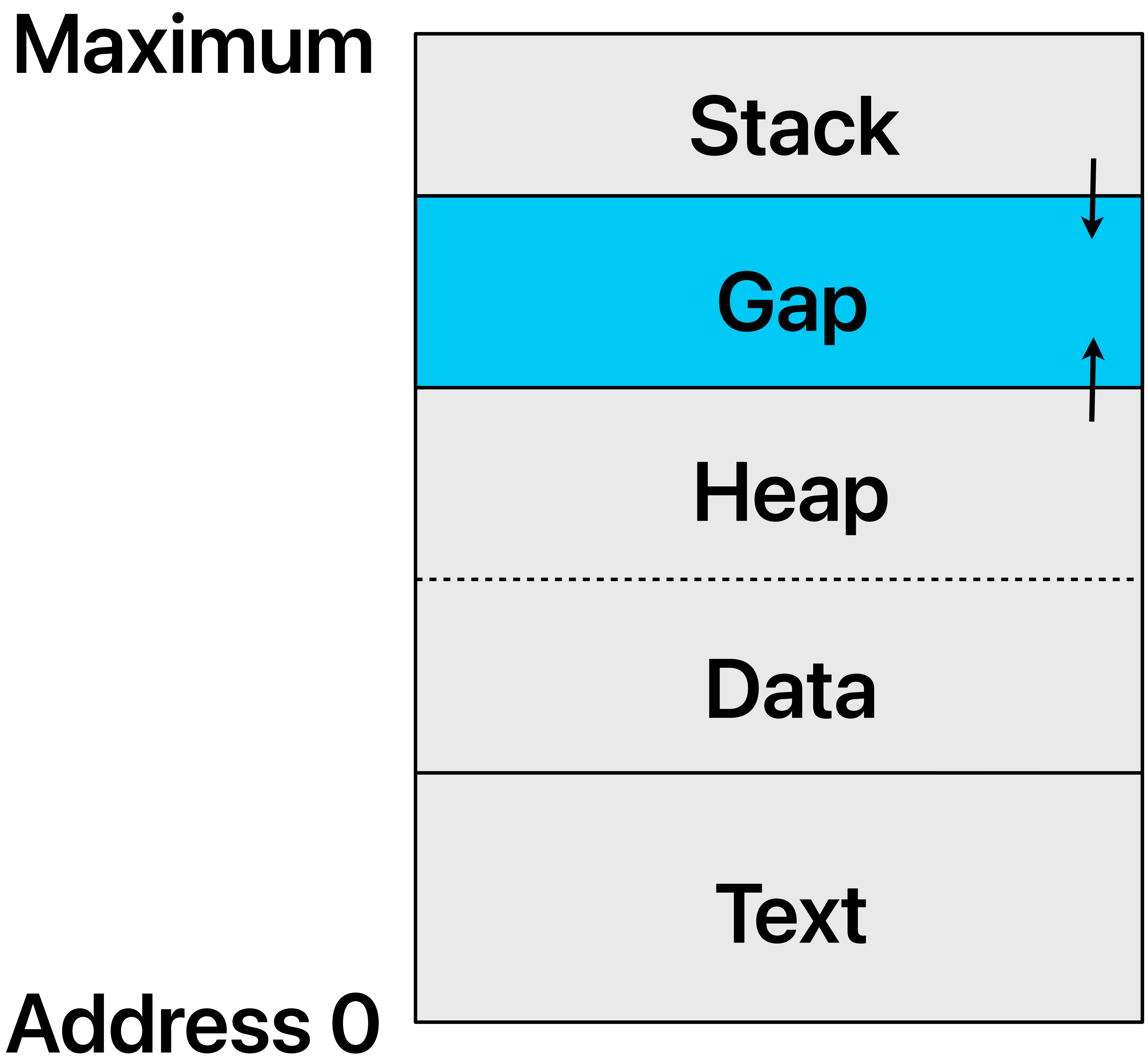
- Registers, Program Counter (PC), Stack Pointer (SP)
- I/O information: a list of files that are currently open

Address Space in a Process

A set of memory sections accessible to a process is called the process' address space

- Text — the program code (usually read only)
- Stack — each frame contains parameters, local variables, and the return address of a function
- Data — global variables and constants
- Heap — dynamically allocated memory (malloc() in C)

The (virtual) address space of a process



Why Use the Process Abstraction?

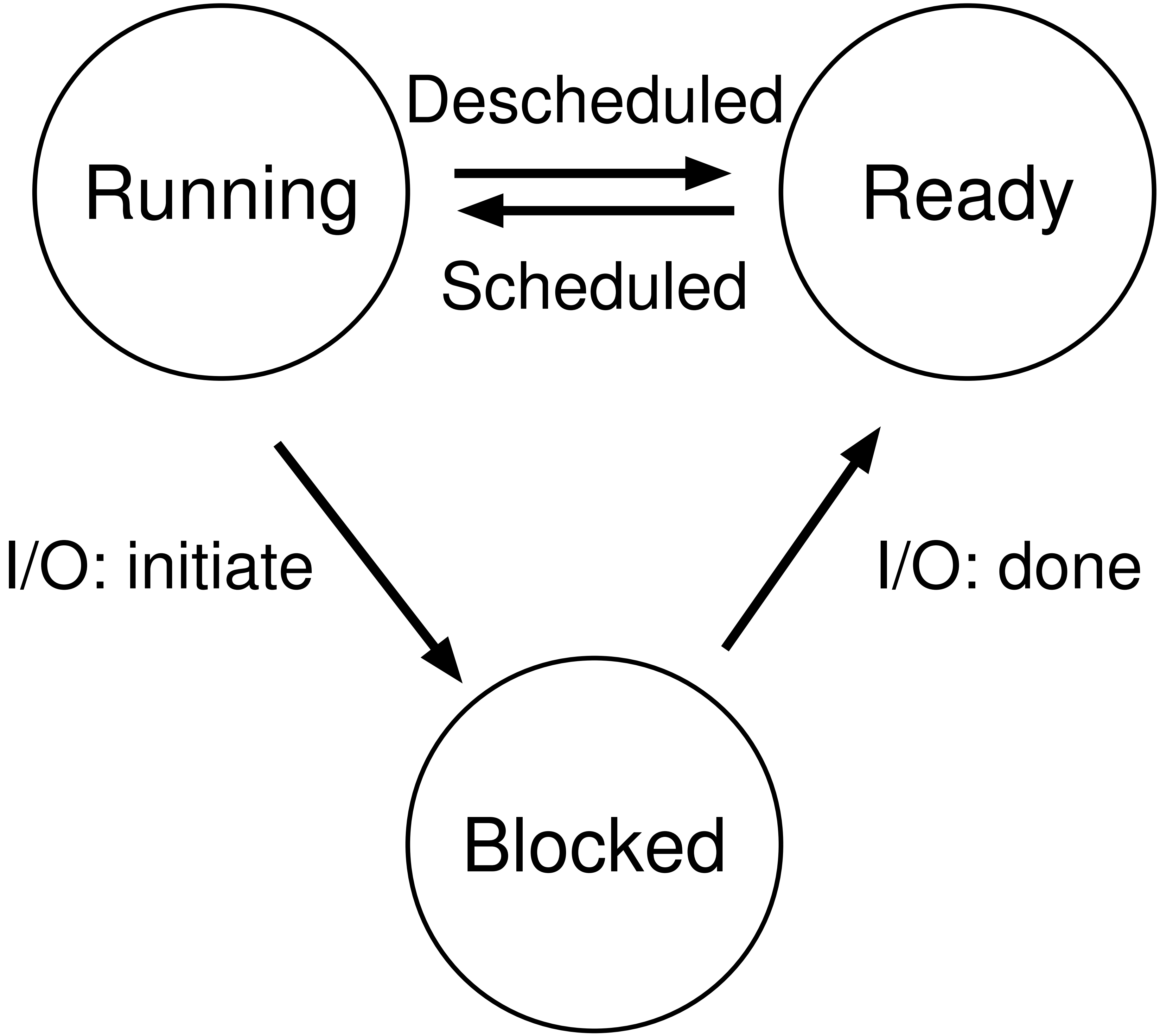
Allows the execution of multiple programs in the same physical address space

Virtualizing the CPU: multiple independent processes running on a physical machine at the same time

- But in reality, at most one process can be active at any instant on each CPU

Process States

Process states



Tracing Process State: CPU Only

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	$Process_0$ now done
5	–	Running	
6	–	Running	
7	–	Running	
8	–	Running	$Process_1$ now done

Tracing Process State: CPU and I/O

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	$Process_0$ initiates I/O
4	Blocked	Running	$Process_0$ is blocked,
5	Blocked	Running	so $Process_1$ runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	$Process_1$ now done
9	Running	–	
10	Running	–	$Process_0$ now done

Process Control Block (PCB)

Information that an OS needs to track about each process:

The process state: blocked, ready, running, zombie

Program counter

CPU registers (for a process that is not running)

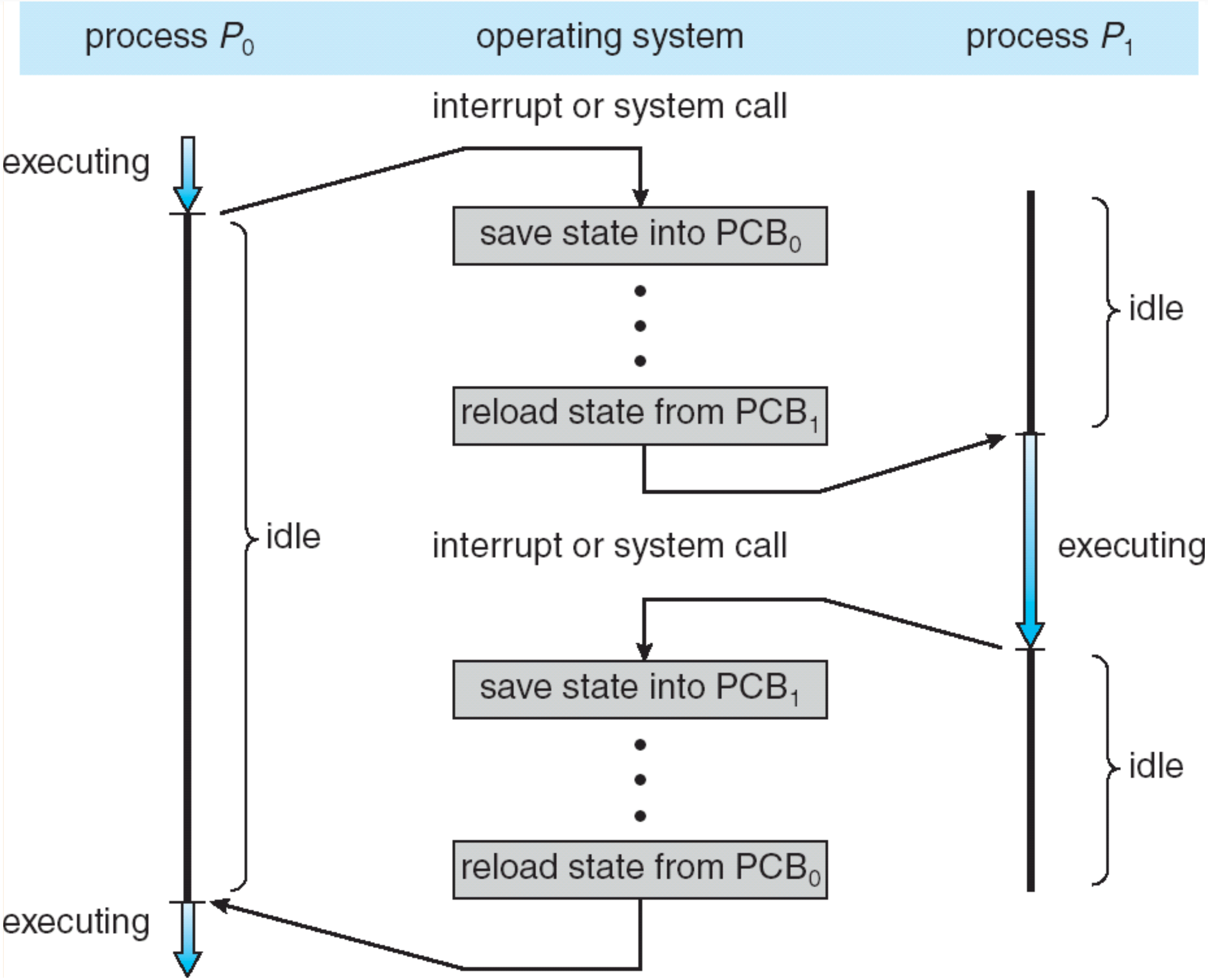
CPU scheduling information: process priority

Memory management information: to be discussed later

Accounting information: amount of CPU and real time used, process ID

I/O status information: a list of open files

PCB is used for saving states in a *context*



An expensive mechanism: context switch

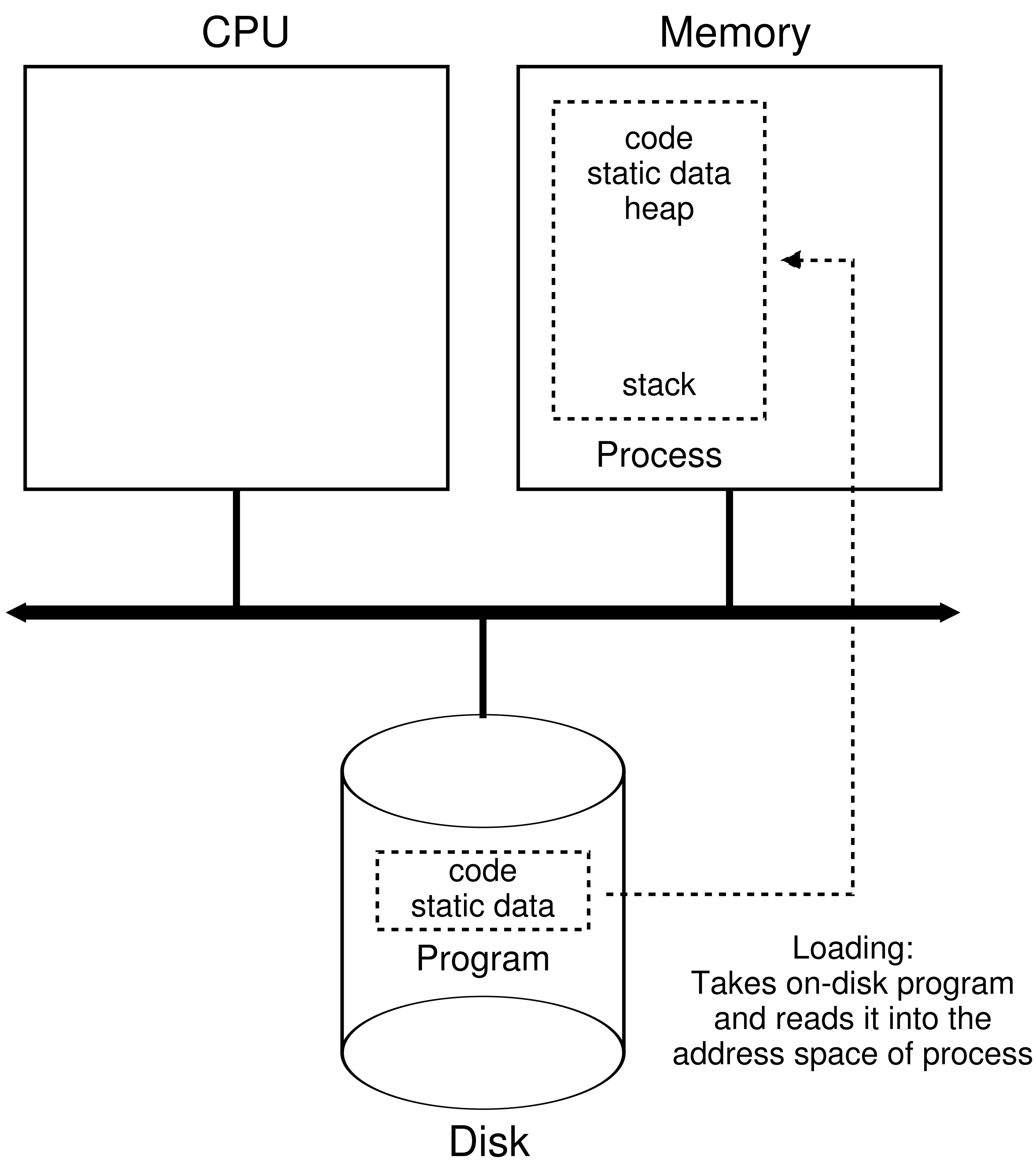
Saving all the states of a process allows a process to be temporarily suspended and later resumed from the same point

Then another process can be resumed by restoring its saved state

The time it takes to perform a context switch is **overhead** that we wish to minimize

System calls related to the process abstraction

Process Creation: loading from the disk



Process creation and termination in UNIX

All processes have a unique process ID

getpid() system call retrieves this ID

Process creation

fork() system call creates a copy of a process and returns in both processes (parent and child), but with a different return value (0 in child)

exec() replaces an address space with a new program

Process termination

exit() or kill() system calls

Example: Process creation in UNIX

csh (pid = 22)

```
pid = fork();

if (pid == 0) {
  // child
  exec(...);
}
else {
  // parent
  wait(NULL);
}
```

Example: Process creation in UNIX

csh (pid = 22)

```
pid = fork();

if (pid == 0) {
  // child
  exec(...);
}
else {
  // parent
  wait(NULL);
}
```

csh (pid = 24)

```
pid = fork();

if (pid == 0) {
  // child
  exec(...);
}
else {
  // parent
  wait(NULL);
}
```

Example: Process creation in UNIX

csh (pid = 22)

```
pid = fork();

if (pid == 0) {
  // child
  exec(...);
}
else {
  // parent
  wait(NULL);
}
```

csh (pid = 24)

```
pid = fork();

if (pid == 0) {
  // child
  exec(...);
}
else {
  // parent
  wait(NULL);
}
```

Example: Process creation in UNIX

csh (pid = 22)

```
pid = fork();

if (pid == 0) {
   // child
   exec(...);
}
else {
   // parent
   wait(NULL);
}
```

csh (pid = 24)

```
pid = fork();

if (pid == 0) {
   // child
   exec(...);
}
else {
   // parent
   wait(NULL);
}
```

Example: Process creation in UNIX

csh (pid = 22)

```
pid = fork();

if (pid == 0) {
  // child
  exec(...);
}
else {
  // parent
  wait(NULL);
}
```

ls (pid = 24)

```
// ls program
int main()
{
   // look up
   // directories
   ...
   return 0;
}
```

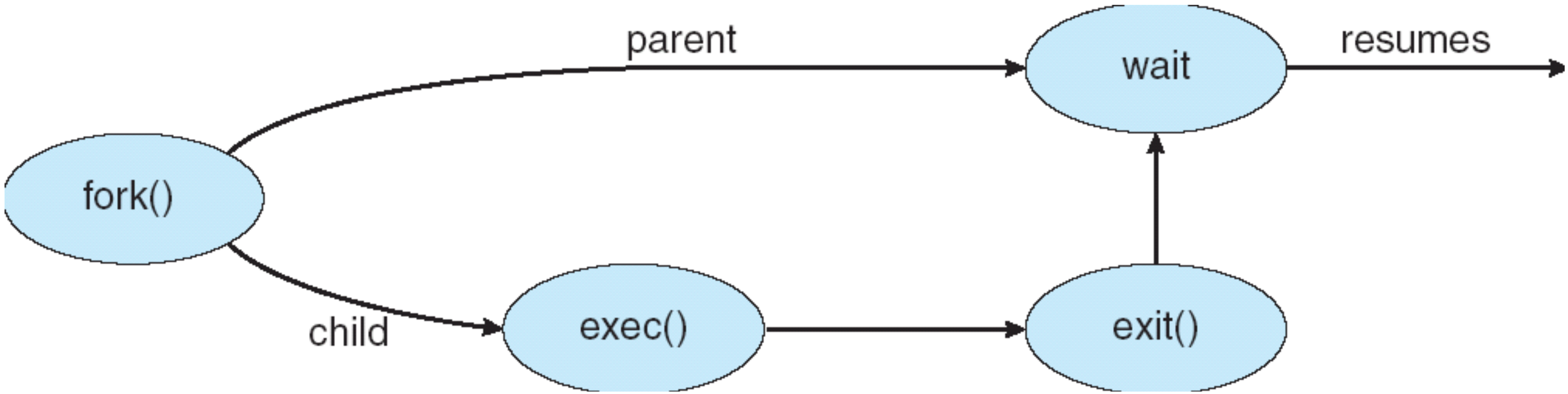
Live Demo

Process creation with fork(): a summary

fork() creates a new process by copying the content of the calling process' address space

The new process has its own

- address space (content is copied from parent)
- Process control block in the OS



What we have covered so far

Three Easy Pieces

Chapter 4: The Abstraction: The Process

Chapter 5: Interlude: Process API