TDDB68 + TDDE47 + TDDD82

Lecture:
Processes, threads and scheduling

Mikael Asplund
Real-time Systems Laboratory
Department of Computer and Information Science

Today's lecture

- Processes
  - Concepts
  - Creation, switching and termination
- Threads
- Process interaction
  - Introduction

Reading guidelines

- Silberschatz et al. (9th and 10th eds.)
  - Chapter 3.1-3.4
  - Chapter 4.1-4.3, 4.5

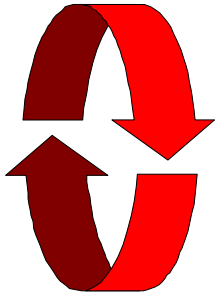
The abstract notion of Process

- An abstraction in computer science used for describing program execution and potential parallelism

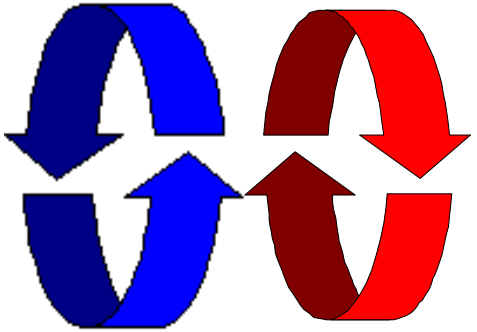
- What other abstractions do you know?
  - Functions
  - Classes, Objects, Methods

- Processes emphasise *the run-time behaviour*

Concurrent Programs



A **sequential** program has a single thread of control.



A **concurrent** program has multiple threads of control allowing it perform multiple computations “in parallel” and to control multiple external activities which occur at the same time.

[Magee and Kramer 2006]

Concurrency

- A mathematical term for modelling computational processes that could **in principle** be run in parallel

- Three possibilities:
  - Single processor, shared memory
  - Multi-processor, shared memory
  - Multiple processors not sharing memory

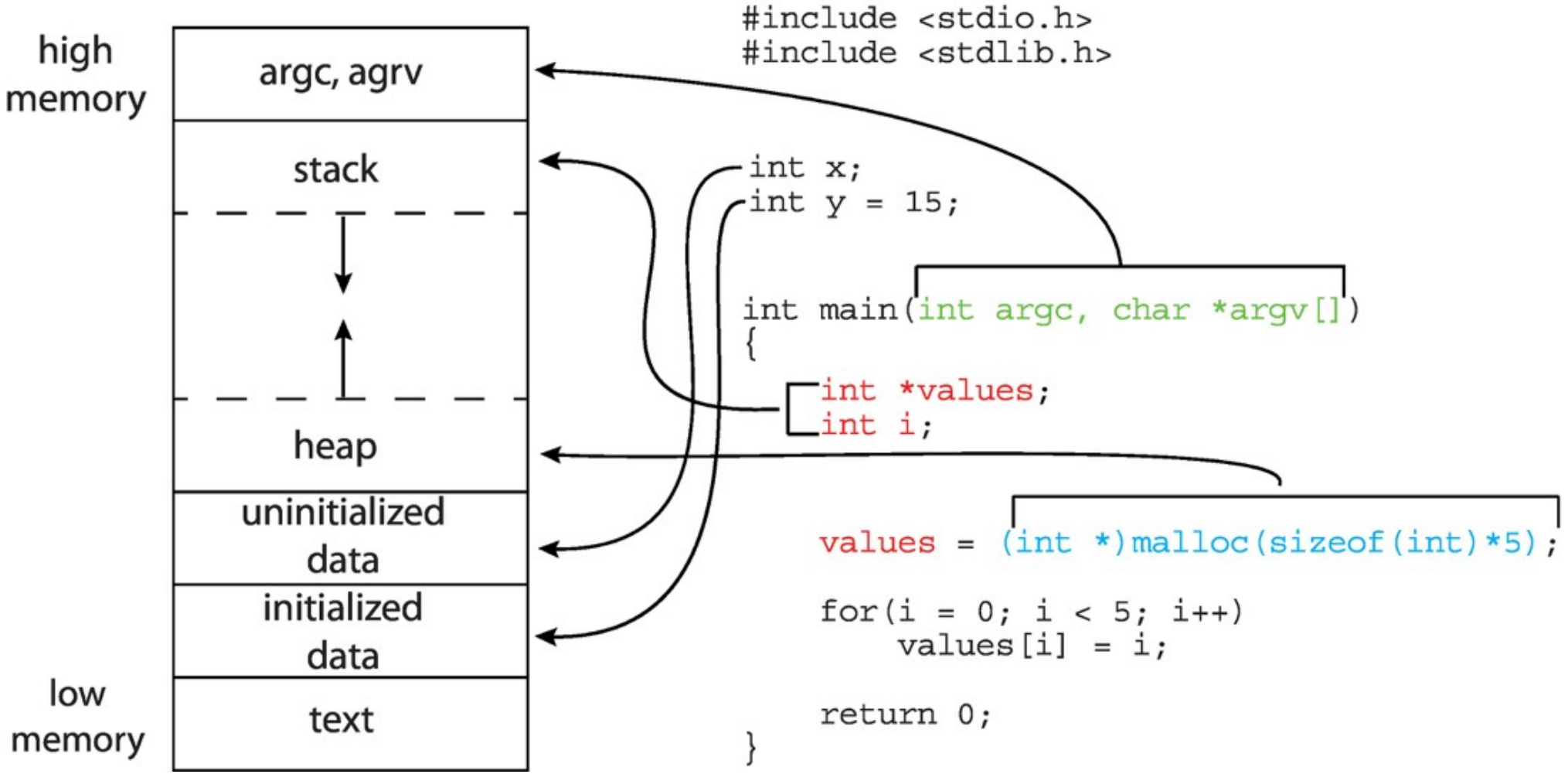
Related terms

• Concurrent programs	• Define actions that *may* be performed simultaneously
• Parallel programs	• A concurrent program that is designed for execution on parallel hardware
• Distributed programs	• Parallel programs designed to run on network of autonomous processors that do not share memory

Typical OS terminology:

A process is a program in execution with its own memory

Process in Memory



Process Control Block (PCB)

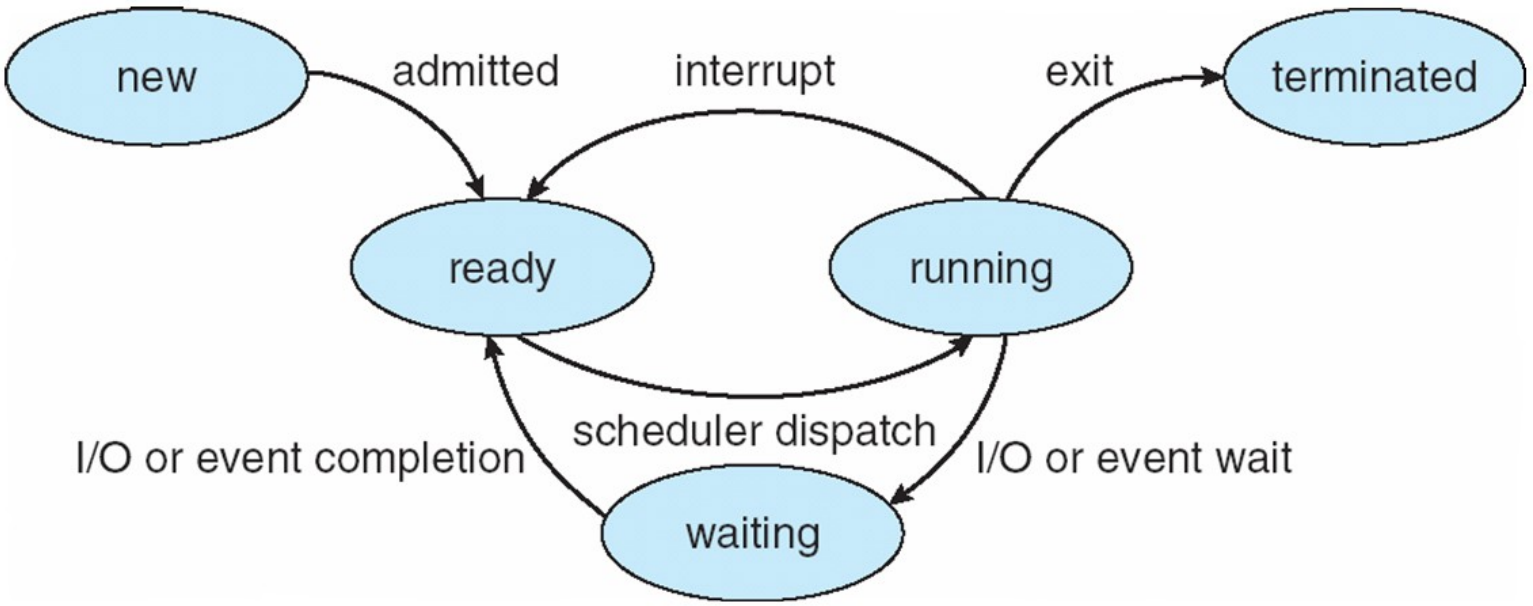
Information associated with each process

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

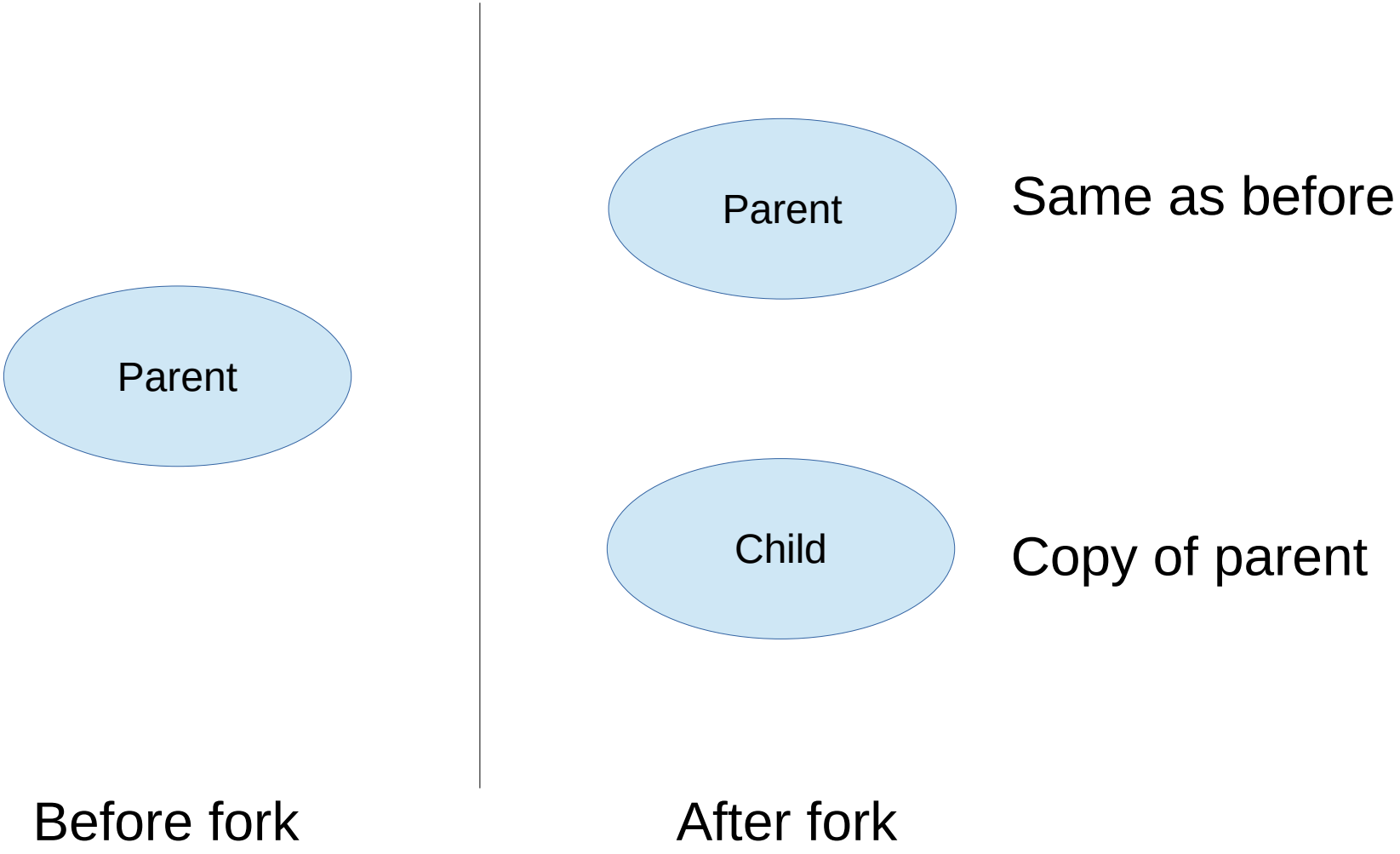
Process representation in Linux

https://github.com/torvalds/linux/blob/master/include/linux/sched.h

Diagram of Process State

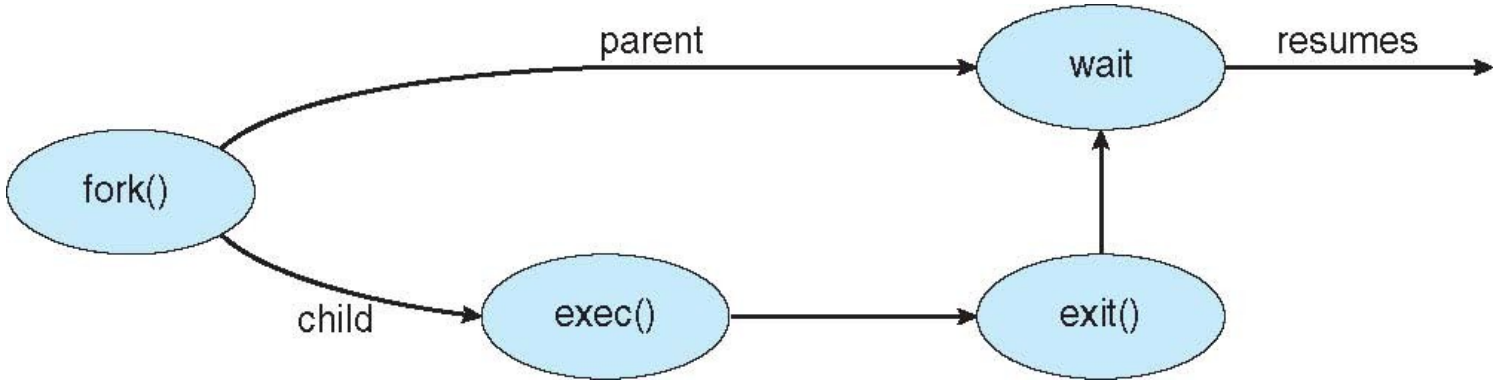


Process Creation



Unix example

- fork() system call creates new process
- exec() system call used after a fork() to replace the process' memory space with a new program



More on fork

- fork returns the process id of the child process
  - This is the only way for a process to know if it is the parent or child after a fork

```
int pid = 0;
pid = fork();
//for child: pid == 0, for parent: pid == <child process id>
```

- In Pintos fork is integrated in exec (not in normal unix)

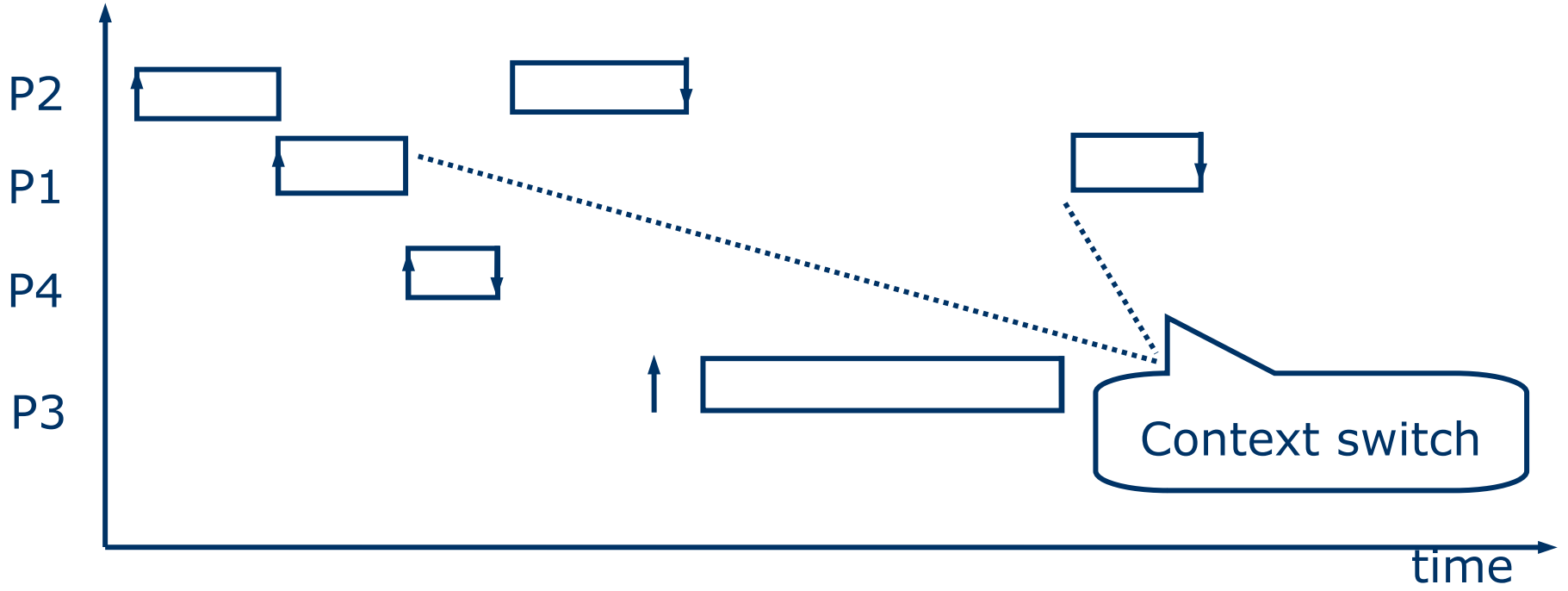
Question for Menti.com (14 16 65)

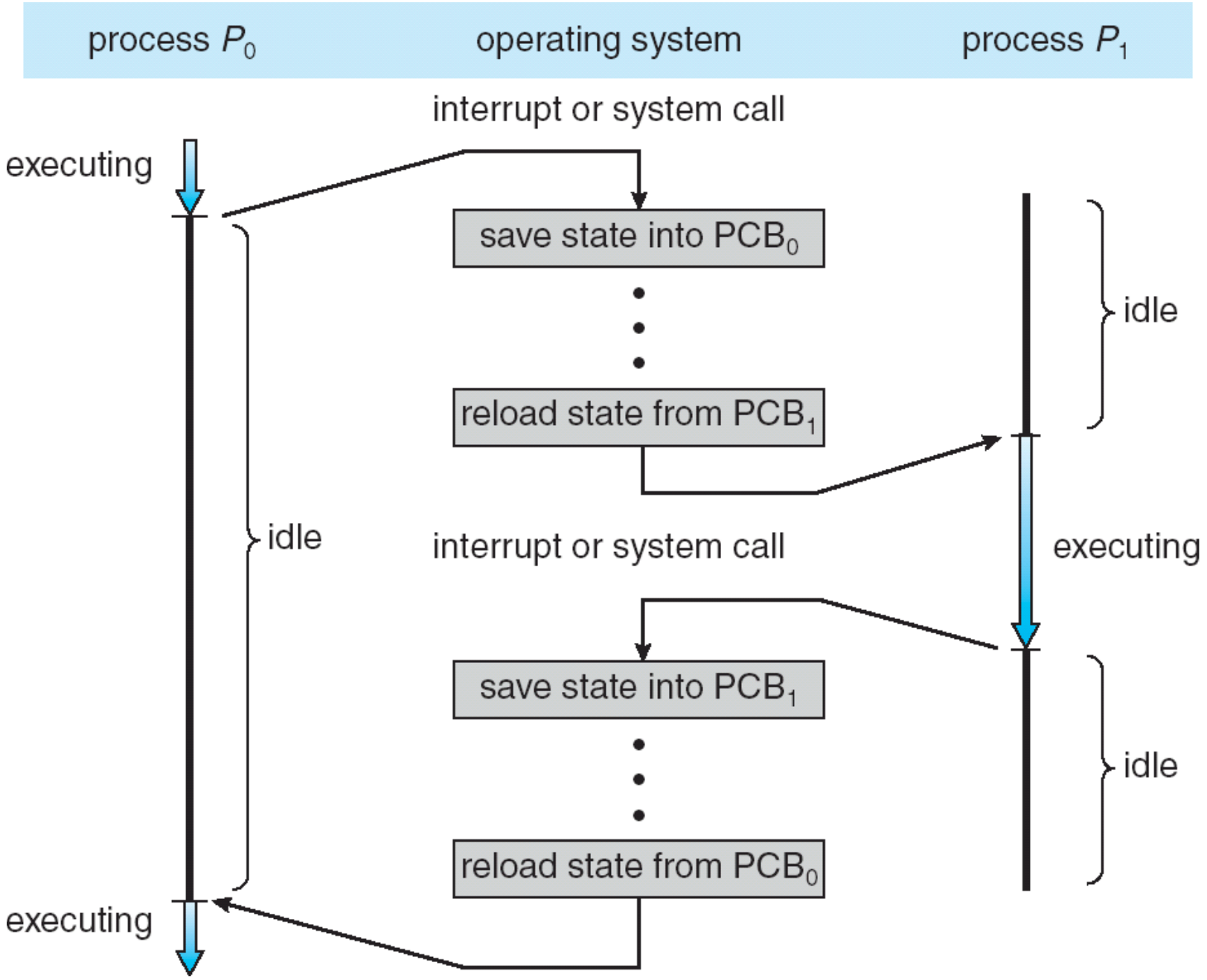
- How many times will the following program output the line ”Hello”?

```
int main(void) {
  int pid1 = 0;
  pid1 = fork();
  if (pid1 == 0) {
    printf("Hello\n");
  }
  int pid2 = fork();
  printf("Hello\n");
  sleep(1);
  return 0;
}
```

Context Switch

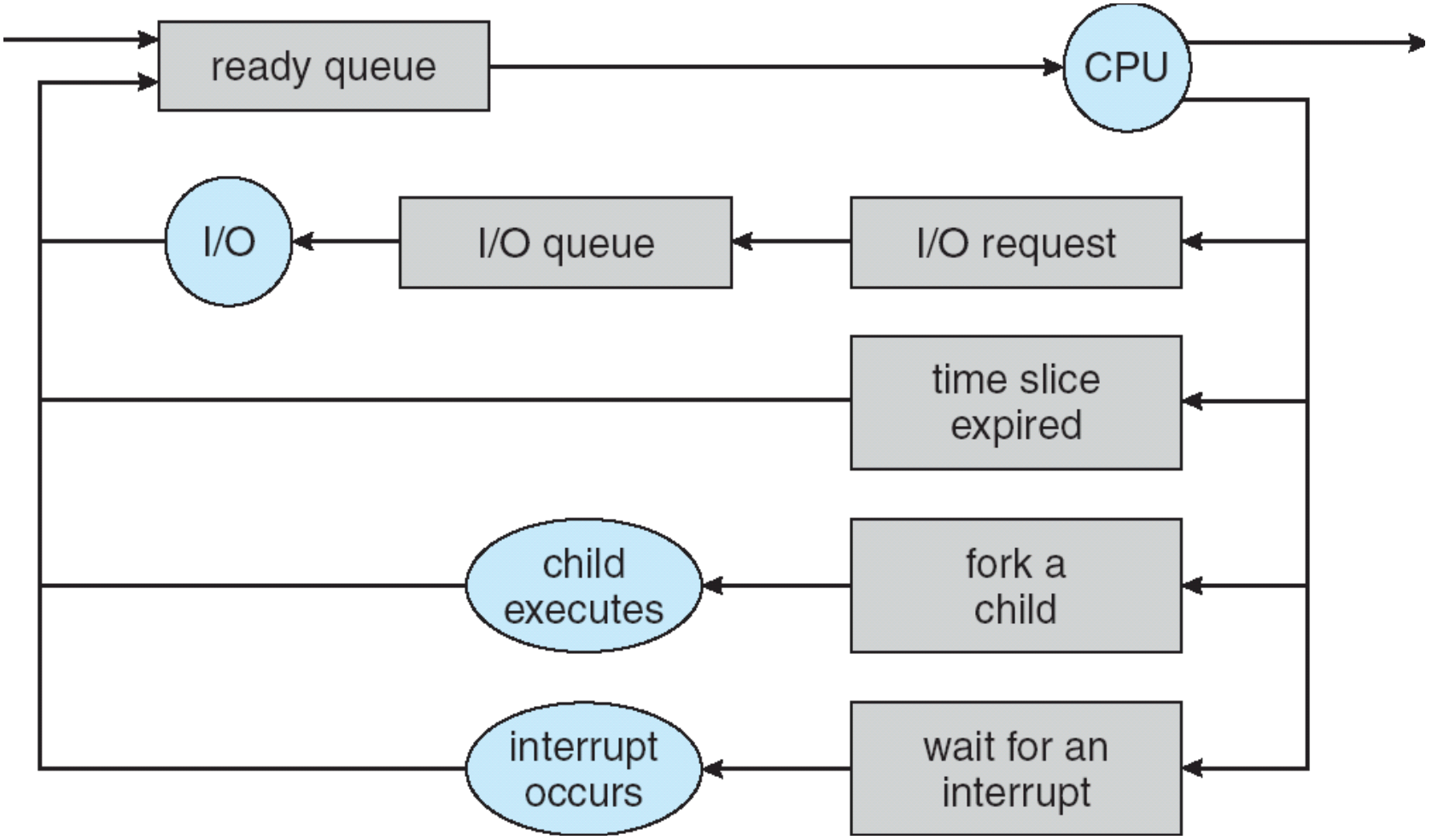
- Consider a program that consists of processes P1,…, P4
- An execution of the concurrent program may look like:




process P_0
operating system
process P_1
interrupt or system call
executing
save state into PCB_0
reload state from PCB_1
idle
idle
interrupt or system call
executing
save state into PCB_1
reload state from PCB_0
idle
executing

What are processes typically doing?

Process queues



For every reason to wait there is a queue. And then there is the ready queue.

Process termination

- Exit a process by calling exit(EXIT_SUCCESS)
  - (same as return 0)
- OS able to free resources
  - Take back memory
  - All of it? No!

Parents must care for their children

- Using processes to run a task in the background:

```
pid_t pid = fork();
if (pid == 0) { //This is run by the child
    //Do something useful
} else { //This is run by the parent
    //Continue with own stuff
    int status;
    wait(&status) //wait until child is done
}
```

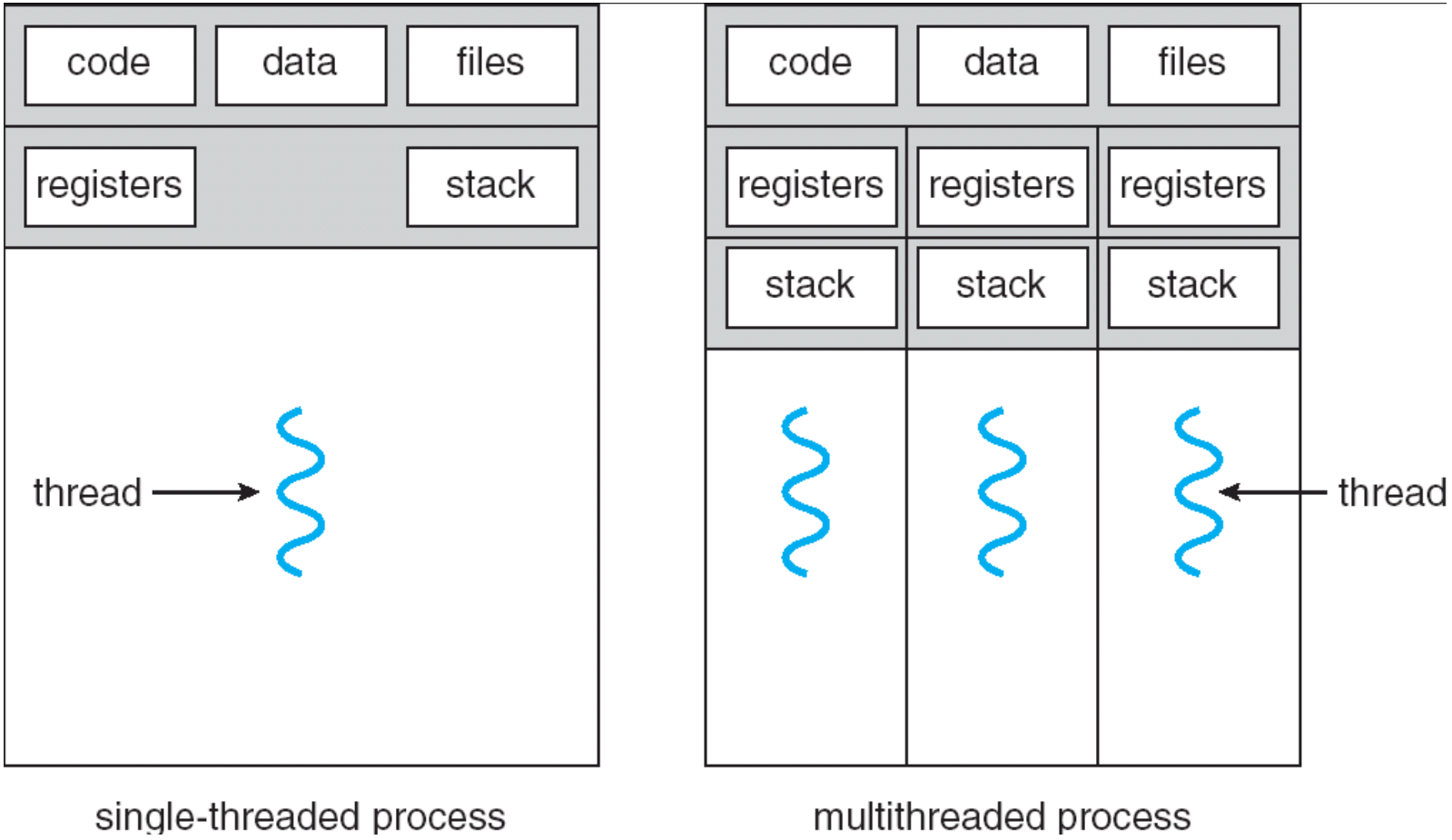
- The OS must manage the interaction between parent and child!

What happens if

- The parent does not call wait?
  - The child becomes a **zombie** process
- The parent terminates before the child?
  - The child becomes an **orphan** process (can be adopted by the init process)

Threads

Single- and Multithreaded Processes



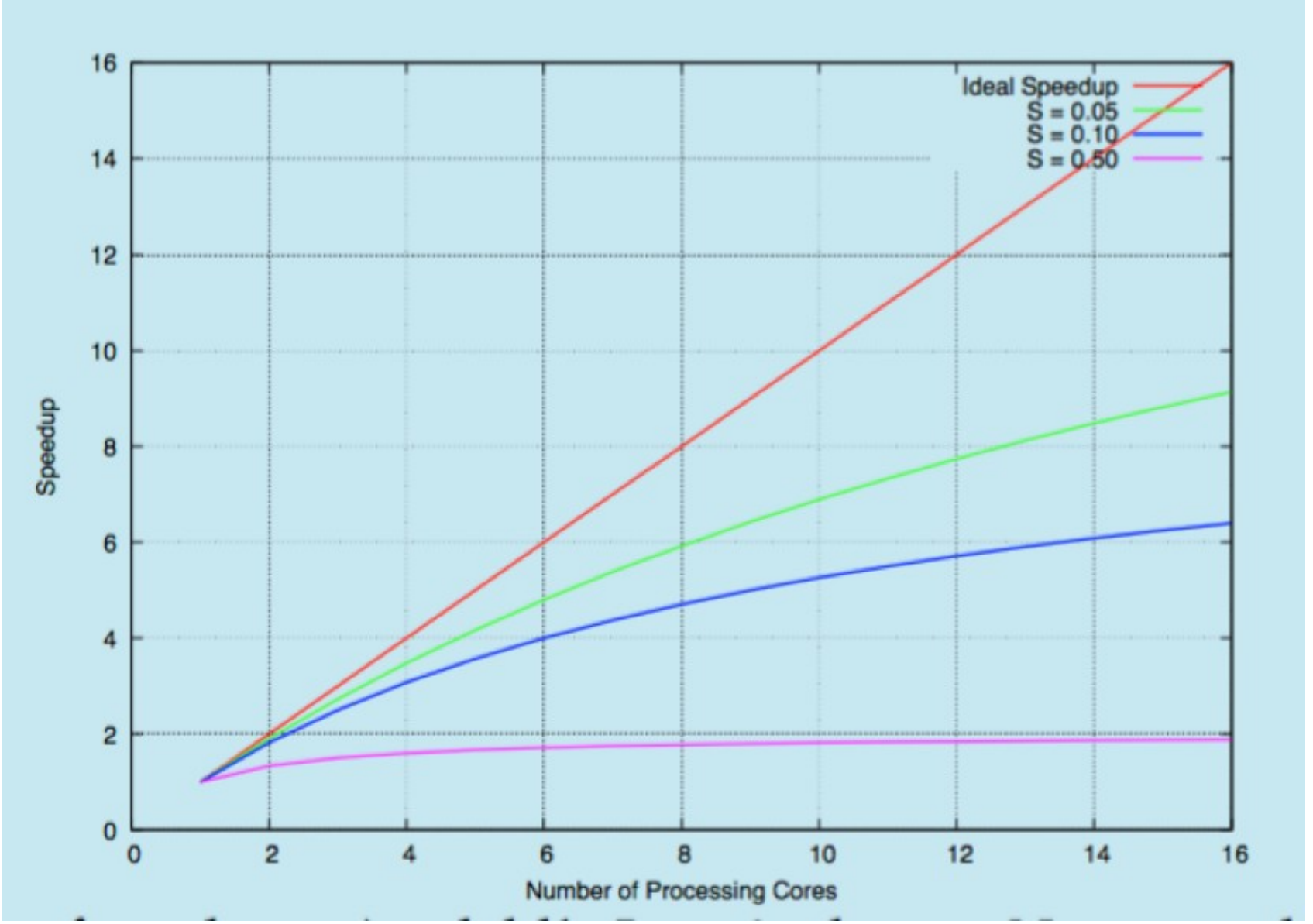
Benefits of Multithreading

- Responsiveness
  - Interactive application can continue even when part of it is blocked
- Resource Sharing
  - Threads of a process share its memory by default.
- Economy
  - Light-weight
  - Creation, management, context switching for threads is much faster than for processes
    - E.g. Solaris: creation 30x, switching 5x faster
- Utilization of multicore architectures

Multi-core architectures

- Necessitated by the end of Moore’s Law
  - We can no longer keep making chips smaller (and thereby faster)
- Problem: Amdahl’s Law...

Multi-core architectures



User-Level Threads

- Threads viewed from the user-level perspective

- Can be provided by
  - Runtime environment of programming language
  - Special libraries
  - Operating system

Kernel-Level Threads

- Managed by the OS kernel (Kernel-specific thread API)
- Each kernel thread services (executes) one or several user threads

Multithreading Models

- Relationship user threads – kernel threads:
  - Many-to-One (M:1)
  - One-to-One (1:1)
  - Many-to-Many (M:N)

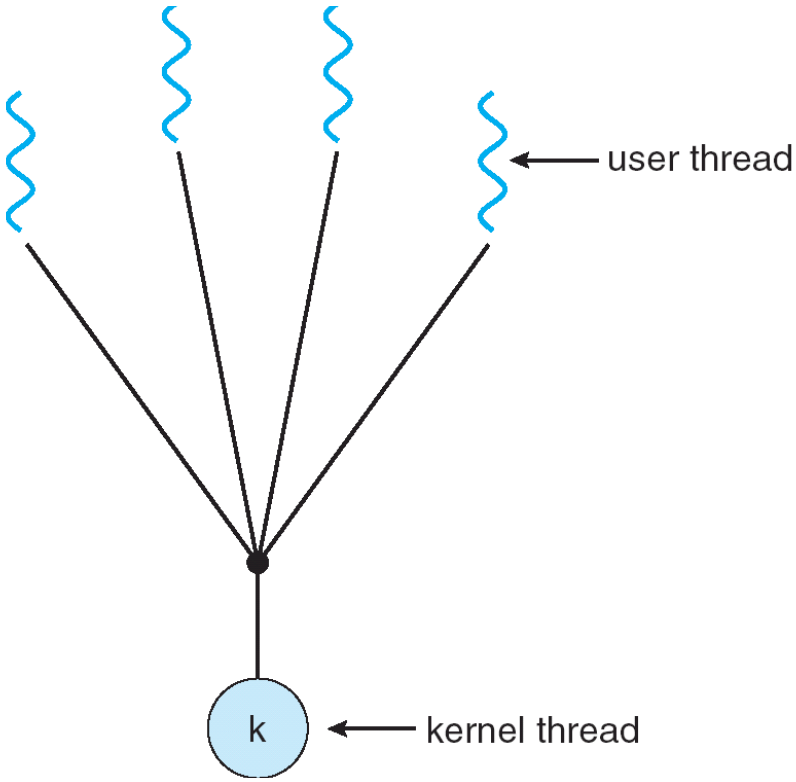
Many-to-One

- Many user-level threads mapped to single kernel thread

☺ Low overhead, very portable

☹ Not scalable to multiprocessors, does not handle blocking calls well

- Example:
  - GNU Portable Threads



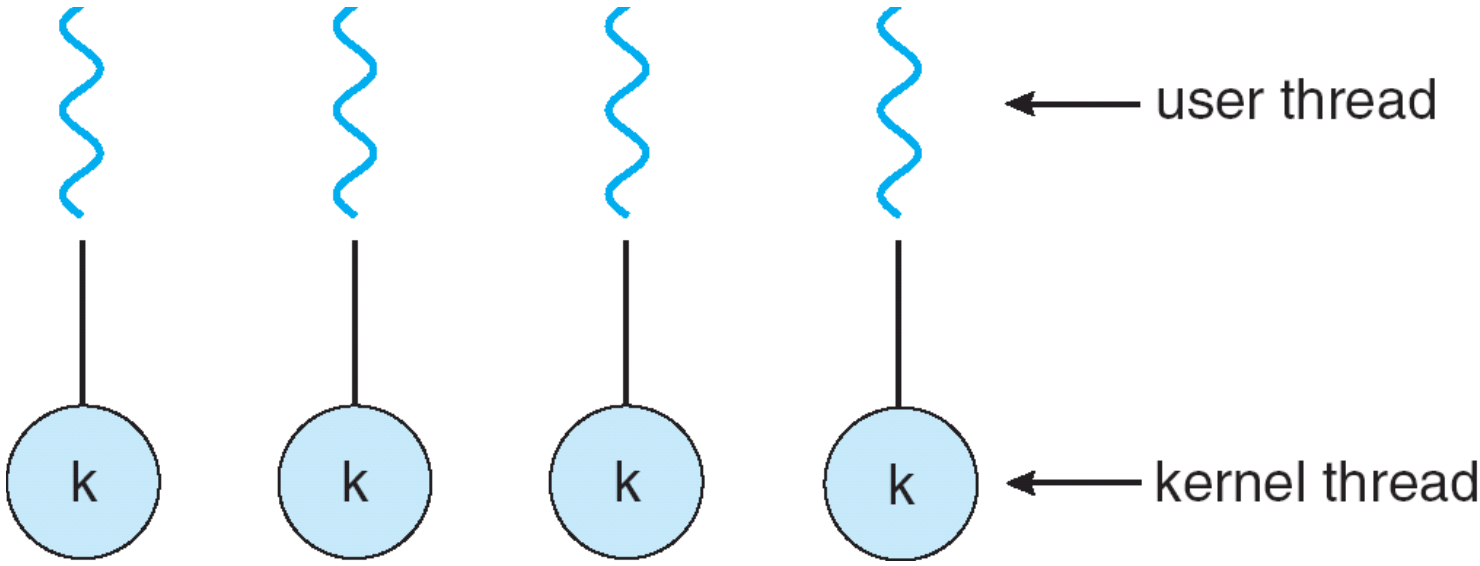
One-to-One

- Each user-level thread maps to one kernel thread

☺ more concurrency; scalable to multiprocessors

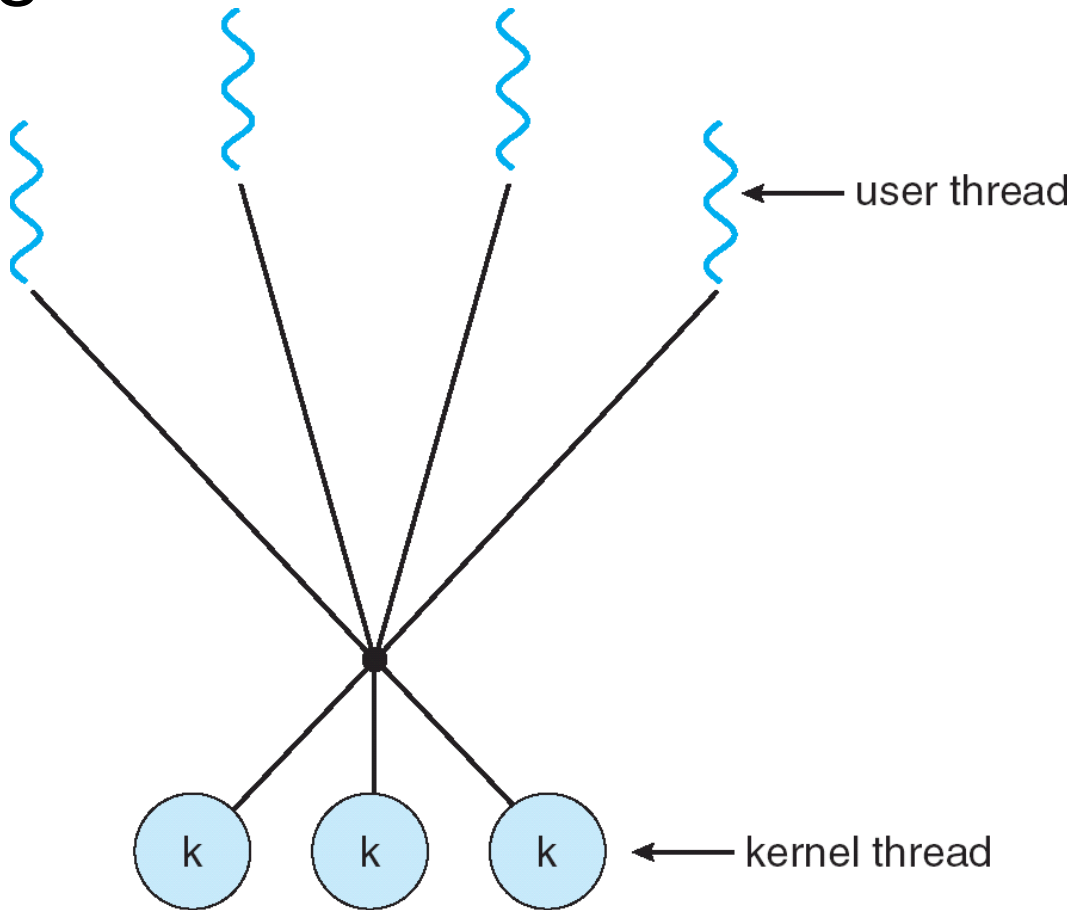
☹ overhead of creating a kernel thread for each user thread

- Examples
  - Windows
  - Unix



Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the OS to create a sufficient number of kernel threads
- Abandoned by most OS:s



Java example

Implementing Runnable interface:

```
class Task implements Runnable
{
   public void run() {
      System.out.println("I am a thread.");
   }
}
```

Creating a thread:

```
Thread worker = new Thread(new Task());
worker.start();
```

Waiting on a thread:

```
try {
   worker.join();
}
catch (InterruptedException ie) { }
```

Implicit threading

- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- Examples
  - Thread Pools (Android, Grand Central Dispatch)
  - Fork-Join (cf. MapReduce)
  - OpenMP

Process interaction

Concurrency creates complexity

Assume each statement of a process is *atomic.*

How many possible traces exist if each of these processes runs exactly once? Menti.com (14 16 65)

```
Process P1 {
  X = 3;
  Y = 0;
  If Y > 1 then
     X = X - 1;
  else
     X = X + 2
}
```

```
Process P2 {
   X = 3;
   Y = 2;
   If Y < 1 then
      X = X + 1
   else
      X = X - 2
}
```

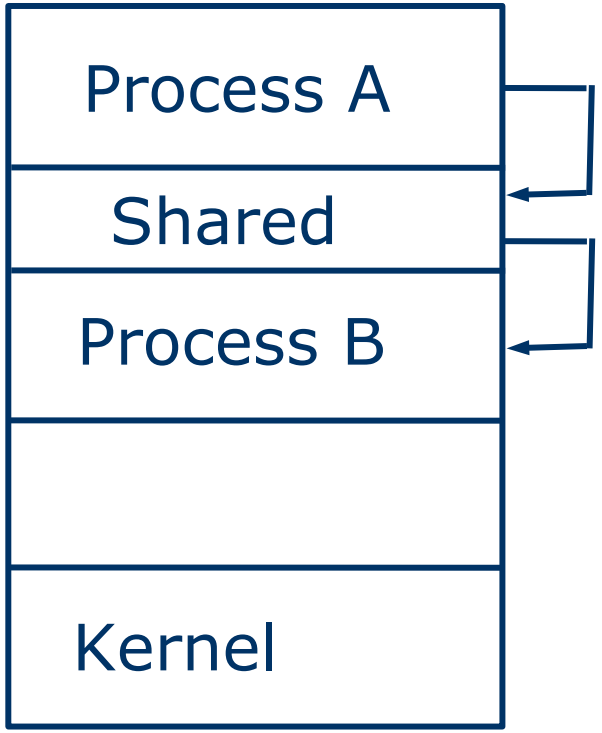
(each process has 4 statements)

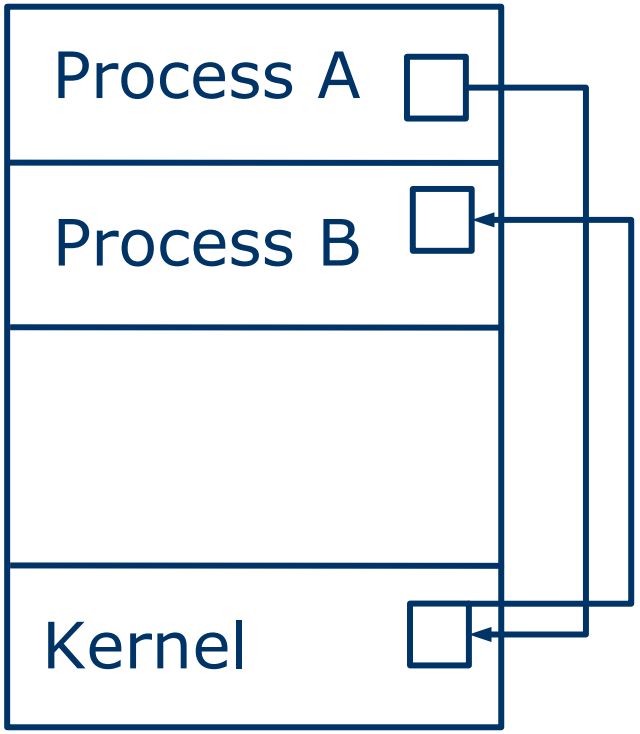
What about larger programs?

- Let’s say each process has 100 atomic statements
- 200 statements and nchoosek(200,100) interleavings
- Approximately 9 * 10^58 interleavings

Communication

- Two modes:
  - Shared variables
  - Message passing





Message passing model

- Resembles a distributed system
- Benefits
  - Clean separation of data
  - Easy to distribute across multiple computers
  - Low risk of data corruption
- Drawbacks
  - Distributed algorithms are difficult to construct
  - Might result in bad performance

Sharing variables

Sharing variables

- Often requires atomicity
- Consider the two processes using a shared variable x initialised at 0:
- What is the outcome of running them both to completion?

```
P0 {                    P1 {
    x = x + 1;              x = x + 1;
}                       }
```

Machine instructions

```
LD R, x  // load register R from x
INC R    // increment register R
ST R, x  // store register R to x
```

- The program may then be compiled into many different interleavings

Non-atomic operations

P0: LD R, x

P0: INC R

P1: LD R, x

P1: INC R

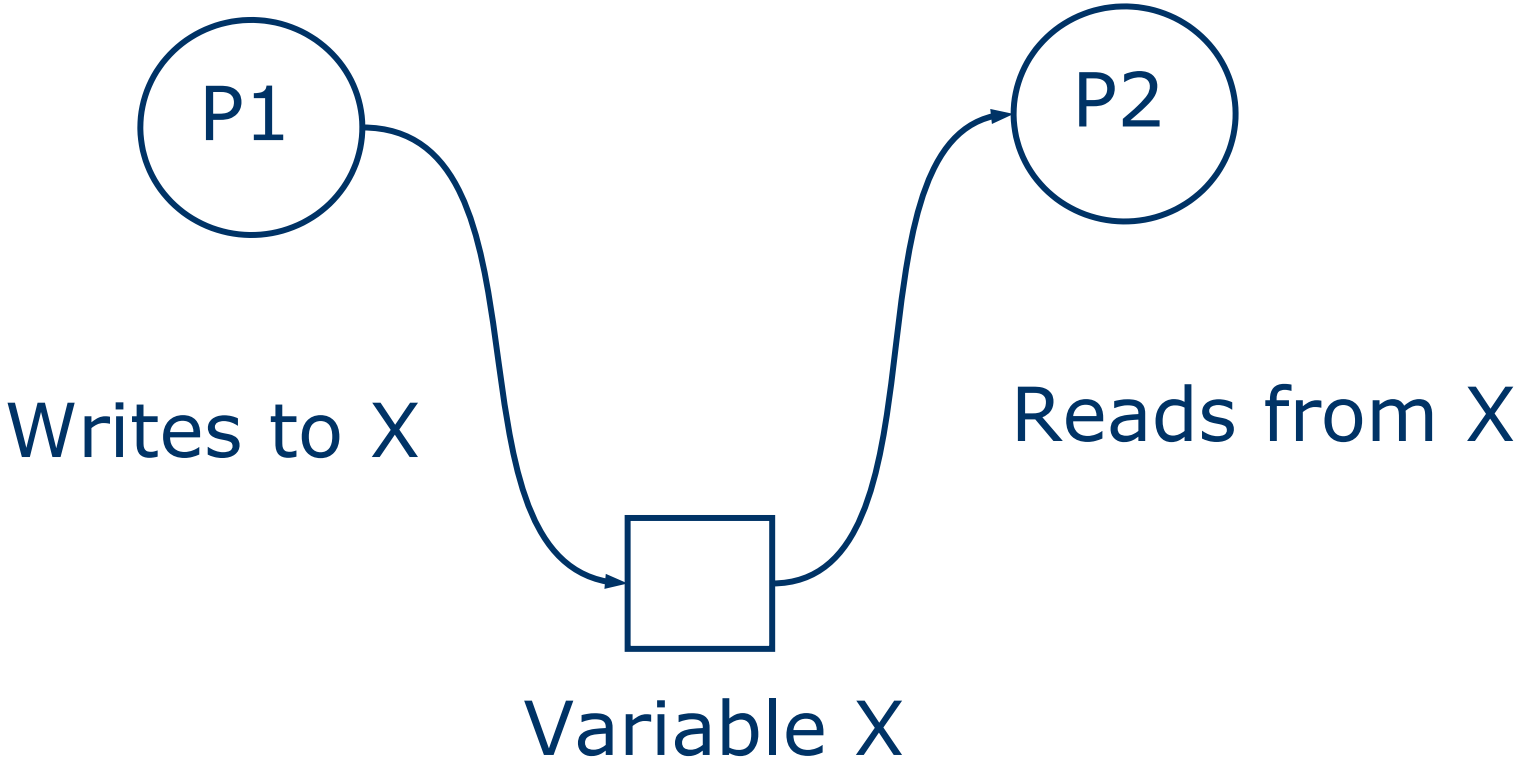
P0: ST R, x

P1: ST R, x

What is the value of x after this trace?

Basic operation

- Communication using shared variables

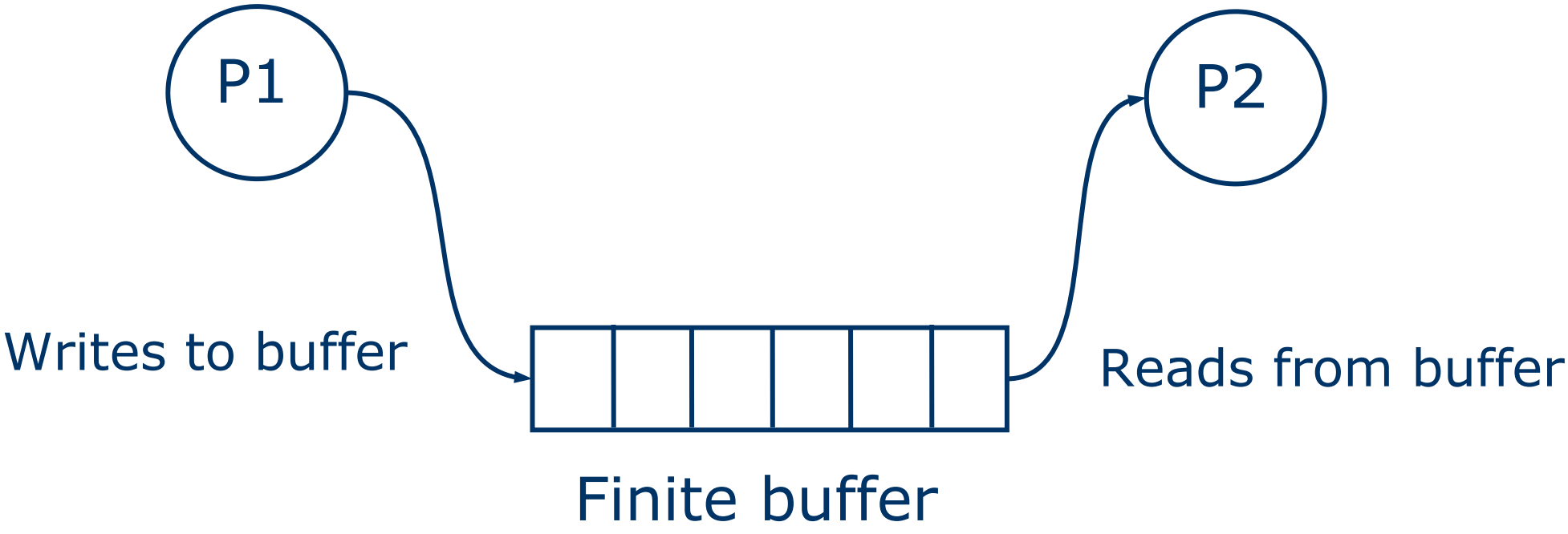


Shared data

- Primitive data types
  - Atomic access often supported by hardware
  - Does not require special protection
- Composite data types
  - E.g., udate date, time and stock value
  - Atomic access needs to be implemented in software

Decoupling from process rates

- Finite buffers



Issues

- Writing to full buffer
- Reading from empty buffer
- Two write operations to the same element

Race condition

If the order of operations performed multiple processes can affect the outcome of the computation, and if this is **unintended**, then the system suffers from a **race condition**

General problems

- Conditional action
  - Examples:
    - Compute the interest when all transactions have been processed
    - Book a flight seat only if seats are available
- Mutual exclusion
  - Example:
    - Two customers shall not be booked on the same seat

Dining-Philosophers Problem