

# Operating System Design

## Exam Review

Neda Nasiriani

Fall 2018



# Exam 1 is next week!

- Office hours:
  - Today 3-5
  - Tomorrow 4:30-6:30

# Review

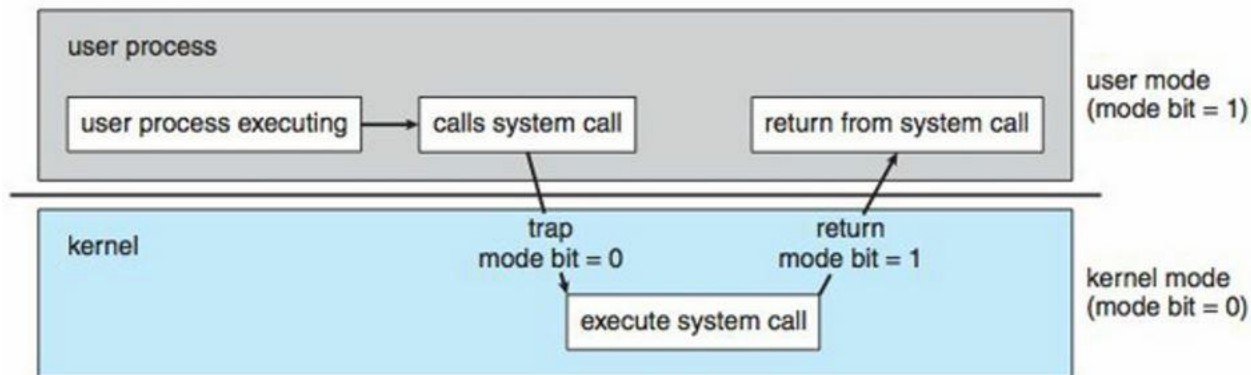
- What is a system call?
  - Provides an interface to the services made available by an operating system
  - Systems execute thousands of system calls per second
    - Every file access
    - Every input/output device
- What is an API?
  - Application programming interface that specifies a set of functions available to the programmers.
- What is the Unix system's API?
  - POSIX which is accessible through C language as **libc** library
- System call interface?
  - Each programming language provides a system call interface that serves as the link to system calls made available by the operating system

# Protecting the Processes and the OS

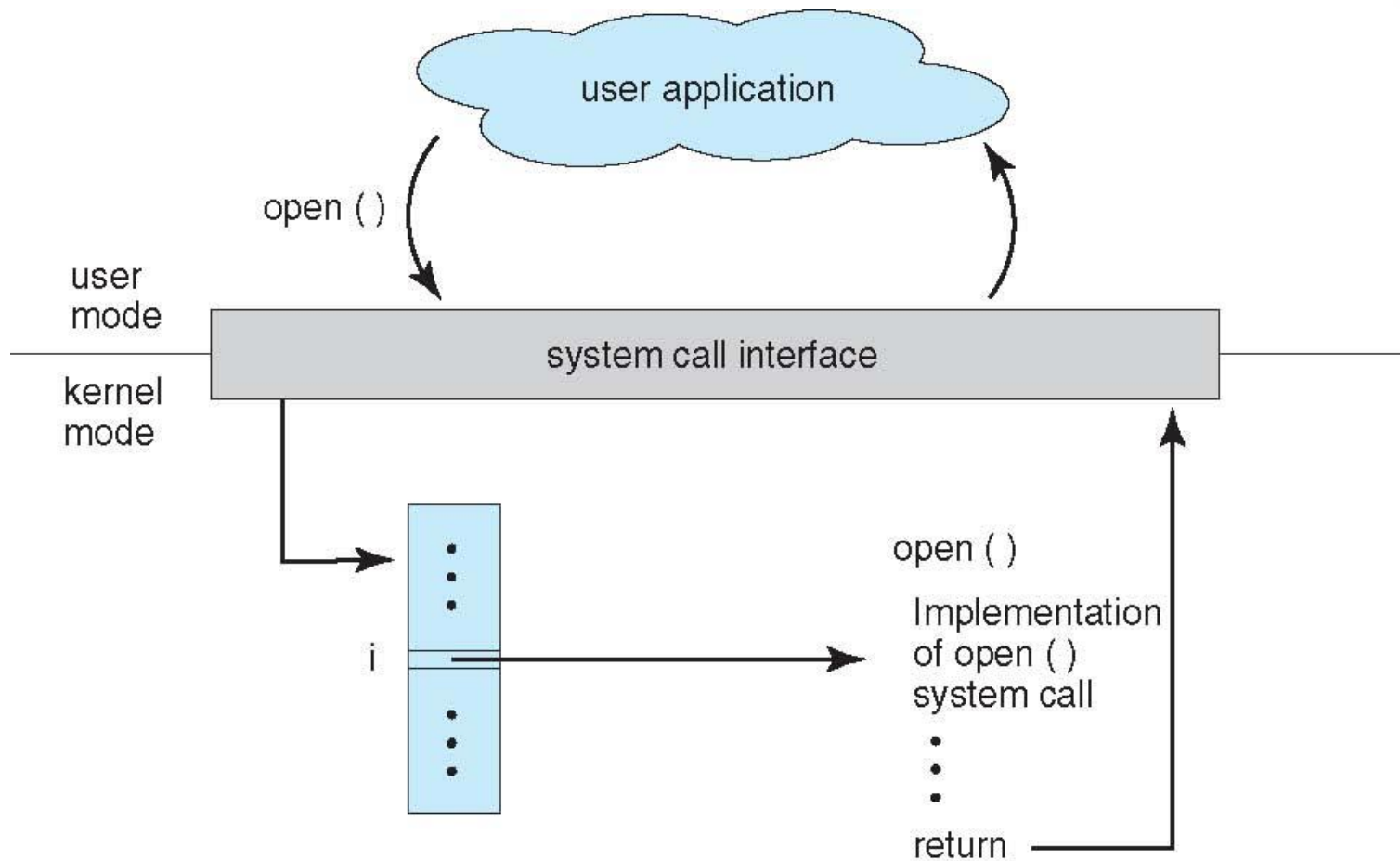
- What if any user is able to execute any system call?
  - Stop another process...
  - Allocate CPU to its own process for as long as it wants
  - Take control of all the devices
  - ....
- What should we do to protect the processes from damaging each other and the OS?
  - We need a layer of protection, but how?
  - Remember that the CPU just pull the instructions and execute them.
  - How can we label instructions which are privileged from the ones that are regular ones?

# Privileged Mode

- There are two modes defined for each instructions which is specified by a **mode bit**: kernel or privileged mode (0) and user mode (1)
- Note that this is embedded in the hardware, hence can not be tampered with by a user process.



**Figure 1.10** Transition from user to kernel mode.



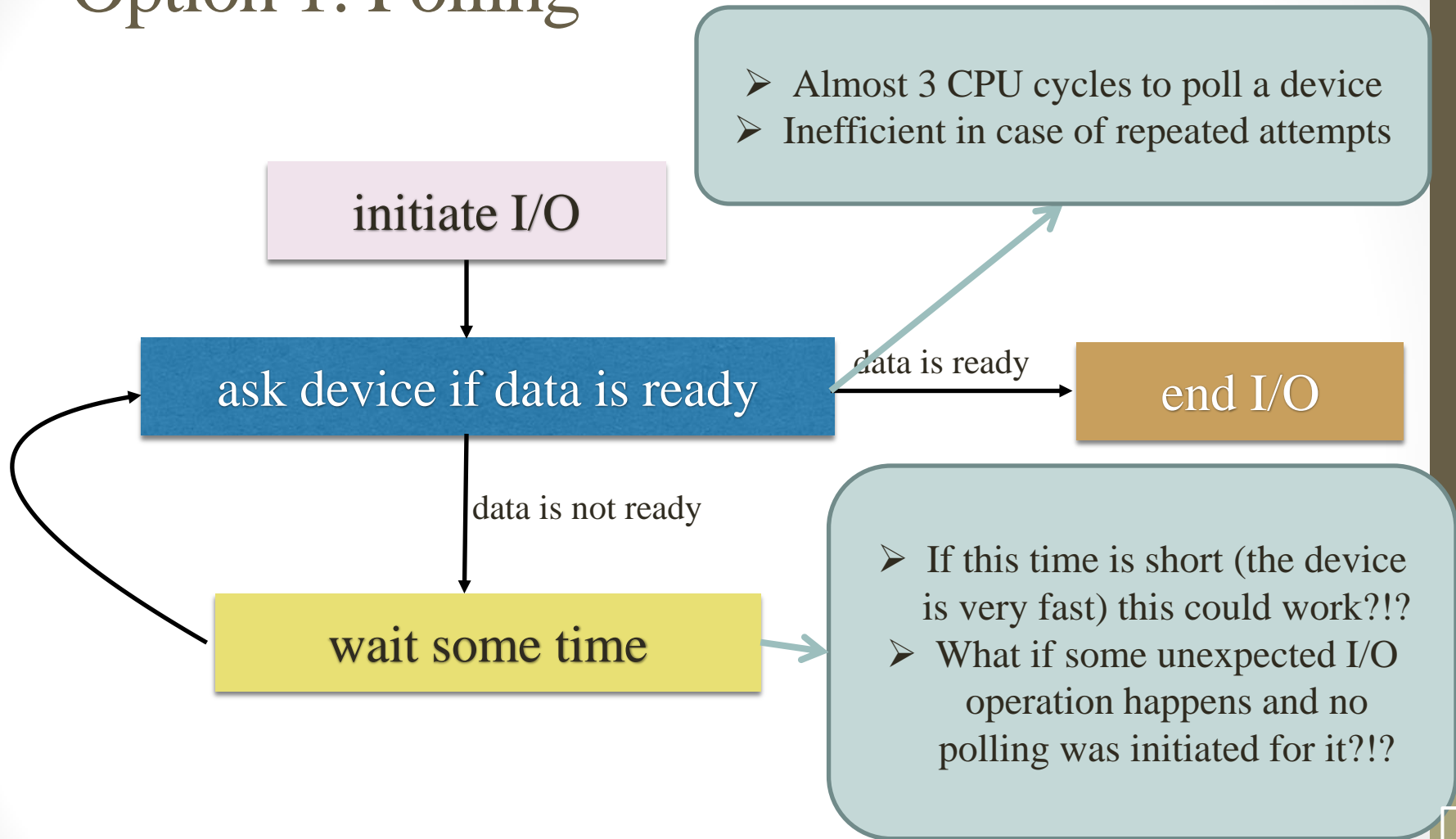
# How does this mode change happen?

- **Trap** is a mechanism that notifies the OS kernel about the intentions of a user process to run a privileged instruction.
- **Trap** is a software signal sent to the OS
  1. One application of traps is **exception handling** when running a user program. E.g., (i) division by zero, (ii) accessing memory that is not allocated to the process
    - some error handling is required by the kernel. Hence trap is the name of the signal that is sent to the OS to ask to handle the situation and send back the appropriate error, or take necessary actions
  2. The other application of traps is **executing system calls**
    - When a user process calls a system call, a trap is initiated that returns the control to the OS to execute the system call in the kernel mode. When the execution is complete, the control is transferred back to the user mode.

# Two Mechanisms for Performing I/O operations?

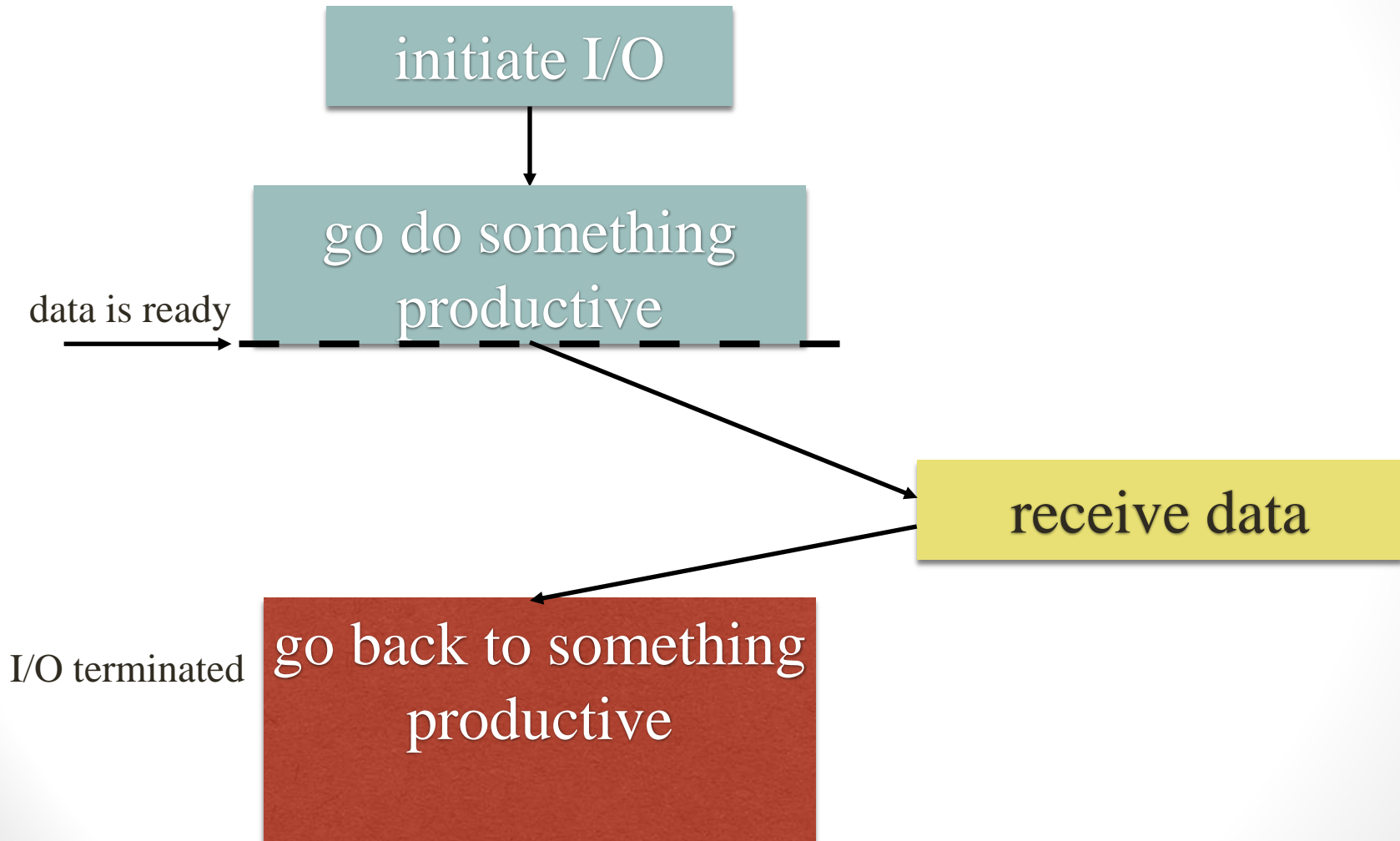


# Option 1: Polling



The Simpsons: <https://www.youtube.com/watch?v=18AzodTPG5U>

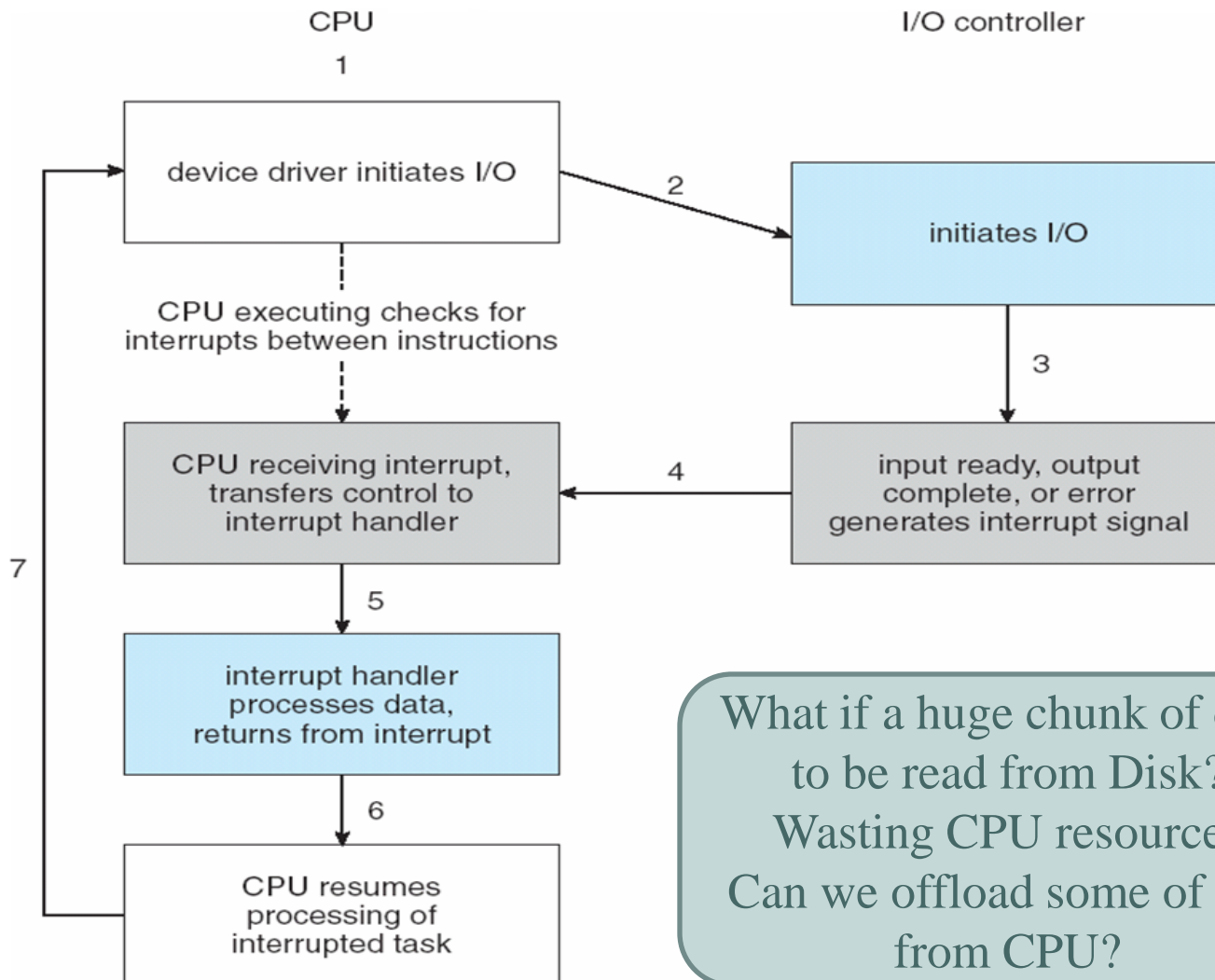
# Option 2: Interrupt



# Interrupts

- CPU **Interrupt-request line** triggered by I/O device
  - Checked by processor after each instruction
- **Interrupt handler** receives interrupts
  - **Maskable** to ignore or delay some interrupts
- **Interrupt vector** to dispatch interrupt to correct handler
  - Context switch at start and end
  - Based on priority
  - Some **nonmaskable**
  - Interrupt chaining if more than one device at same interrupt number
- Remember: **Traps** are software interrupts

# Interrupt Driven I/O Cycle

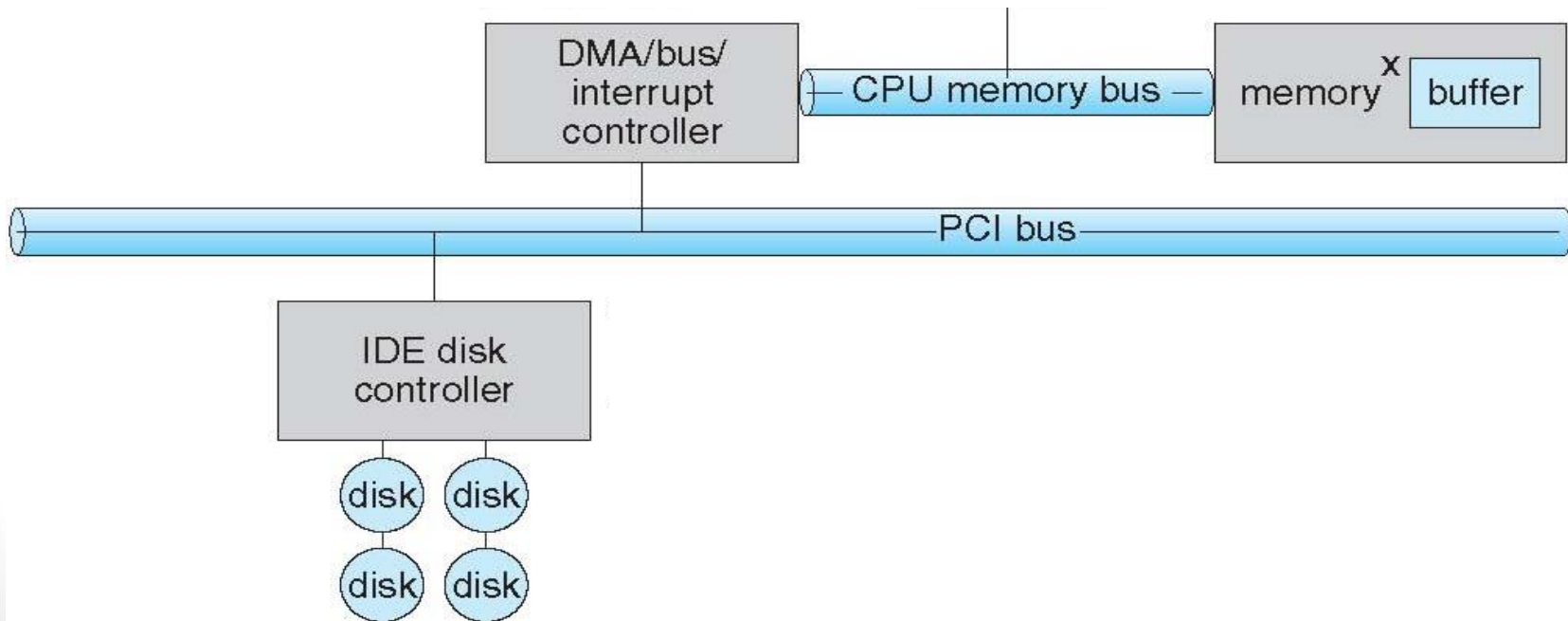


What if a huge chunk of data  
to be read from Disk?  
Wasting CPU resources  
Can we offload some of this  
from CPU?

# Direct Memory Access?

# Direct Memory Access (DMA)

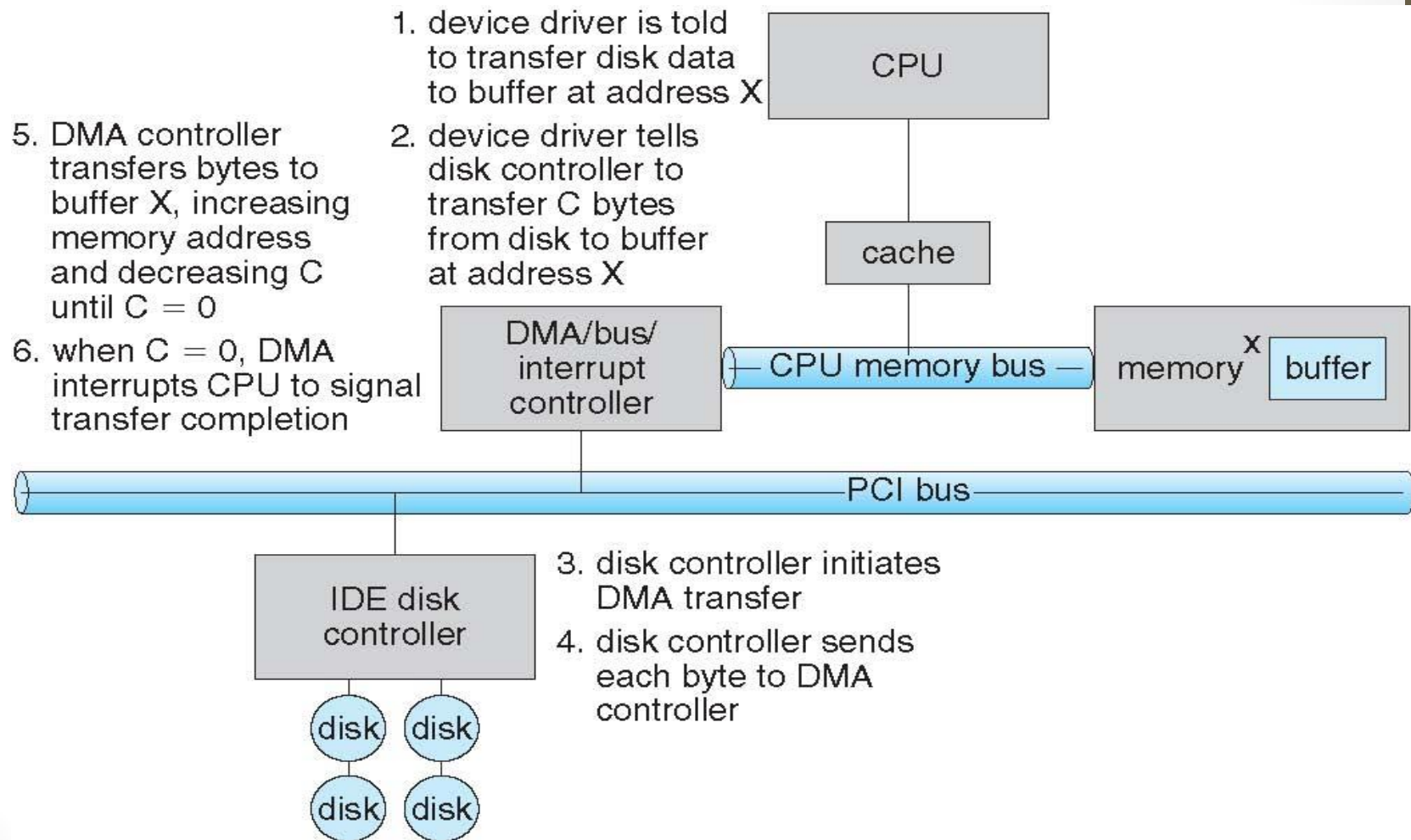
- What if we have a simpler controller that knows
  - the data location to be read from the disk
  - the memory location that it should be copied too
  - can access the memory
  - And can take care of this...



# Direct Memory Access (DMA)

- Bypasses CPU to transfer data directly between I/O device and memory
- OS writes DMA command block into memory
  - Source and destination addresses
  - Read or write mode
  - Count of bytes
- The CPU writes location of command block to DMA controller
- Handshaking between DMA controller and Device controller
  - DMA-request and DMA-acknowledge (for each byte of data transfer)
  - Bus mastering of DMA controller – grabs bus from CPU
    - **Cycle stealing** from CPU but still much more efficient
- When done, DMA controller interrupts to signal completion

# DMA: How it works





# Booting the OS!

# Booting Steps Review

- 1) Where is BIOS stored when your machine powers up?
  - 1) The BIOS is stored on Flash disk which is mapped to BIOS address in memory
- 2) What is in the RAM (main memory) when your machine powers up?
  - 1) Garbage!
- 3) What are example boot devices and who chooses the boot device for the BIOS?
  - 1) CD-ROM, Hard drive, ...
  - 2) The user chooses it
- 4) What is a Master Boot Record (MBR) and what does it contain?
  - 1) First 512 Bytes of the boot device and contains the bootstrapper program and partition table

# OS types

- Batch systems
- Multiprogrammed systems
- Time sharing systems (interactive systems)
- Multi processor systems
- Distributed systems
- Real-time systems

# Processes


# Processes Components

- What are the main components of a process?
  - Text section
    - The code
  - Stack
    - Local variables
    - Function parameters
    - ...
  - Heap
    - Dynamically allocated memory
  - Data Section
    - Global variables
  - What else?

# Processes Components

- Assume processes A is running in a system
  - The CPU decides to switch from process A to another process
  - What information will the CPU need to resume process A later?
    - Program Counter
    - Value of registers
- SO, a process is associated with the following components
  - Text section
  - Data section
  - Heap
  - Stack
  - Program Counter
  - Value of Registers

Process A



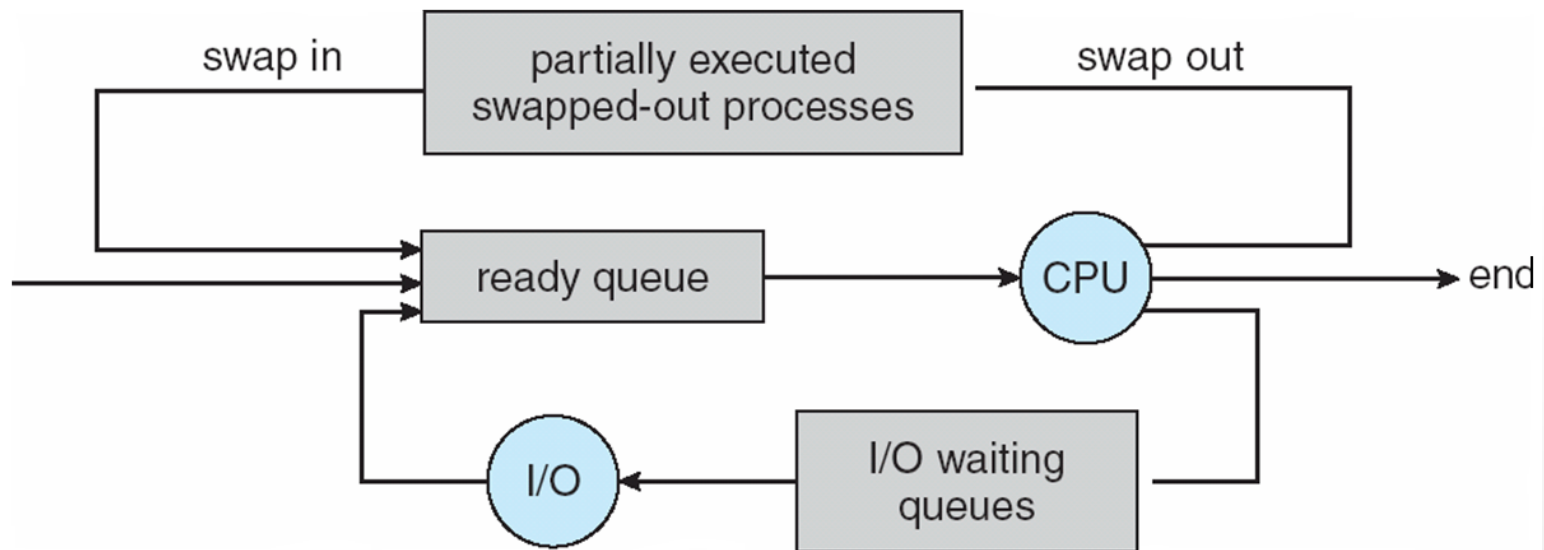
```
lw $t0, offset($s0)
lw $t1, offset($s1)
add $d, $t0, $t1
.
.
.
```

# Scheduler

- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds)  $\Rightarrow$  (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes)  $\Rightarrow$  (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming**
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good *process mix*

# Medium-Term Scheduler

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**





# Process Creation: How?

- An existing process can create a new process by calling the `fork()` system call
- `fork()` runs once in the parent process but returns two times,
  - 1) In the child process, with returning value of 0
  - 2) In the parent process, with the value of the child process id
- Both the child and the parent process start executing with the instruction that follows the `fork()` system call
- The child process gets a copy of the parent's data space, heap and stack (It is a separate copy from the parent's)

# Process Creation using fork(): sharing resources

- The child process gets **its own copy** of the data section, stack and heap of the parent process
- The child process get a duplicate of all open file descriptors in its parent process
  - The parent and the child share a file table entry for every open descriptor
  - The parent and the child share the same file offset
    - If a child process is writing to standard output when the parent process is executing it can append to the end of standard output
  - Note that every UNIX program has three streams opened for it when it starts up, one for input (stdin), one for output (stdout), and one for error messages (stderr) with file descriptors of 0, 1 and 2 respectively.
- Does changes in the child variables change the parent variables?
  - No

# Process Creation: example

before fork

pid = 12387, glob = 7, var = 89

pid = 12386, glob = 6, var = 88

```
#include <stdio.h>
#include <sys/types.h>
```

```
int    globvar = 6;
```

/\* external variable in initialized data \*/

```
int
main(void)
{
```

```
    int    var;          /* automatic variable on the stack */
    pid_t  pid;
```

```
    var = 88;
```

```
    printf("before fork\n");    /* we don't flush stdout */
```

```
    if ((pid = fork()) < 0) {
        printf("fork error");
```

```
    } else if (pid == 0) {      /* child */
        globvar++;             /* modify variables */
        var++;
```

```
    } else {                   /* parent */
```

```
    }
    printf("pid = %ld, glob = %d, var = %d\n", (long)getpid(), globvar, var);
    return;
```

```
}
```

# Process Termination

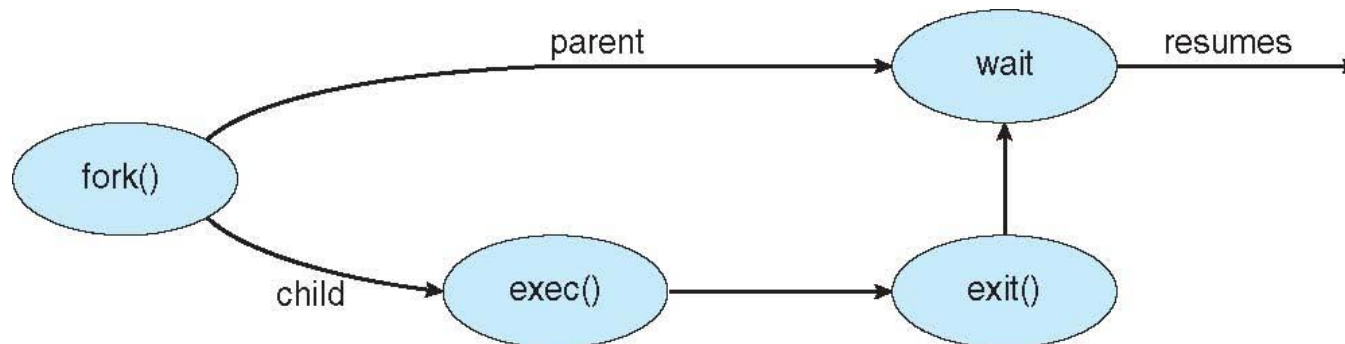
- Process executes last statement and then asks the operating system to delete it using the **exit()** system call.
  - Returns status data from child to parent (via **wait(&status)**)
  - Process' resources are deallocated by operating system
- The parent process may wait for termination of a child process by using the **wait()** system call. The call returns status information and the pid of the terminated process

**pid = wait(&status);**

- If parent has not called **wait()** yet but the child is terminated, the info of child is still kept in the process table. This child process is a **zombie**
- If parent terminated without invoking **wait**, process is an **orphan**, which is adopted by **init** process

# Process Creation Diagram

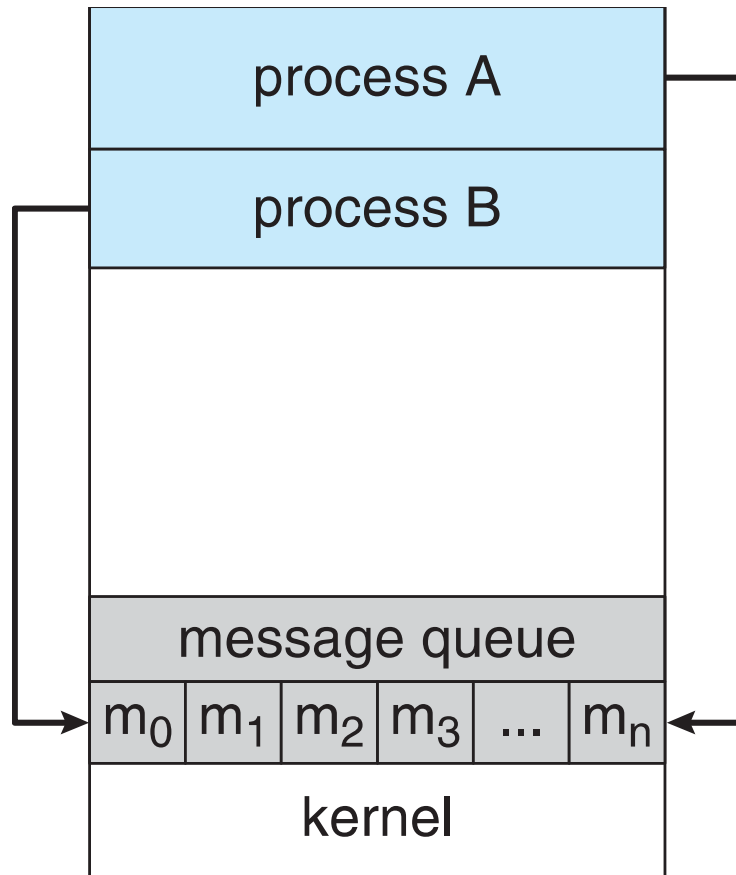
- The parent can wait on the child process by system calls
  - `pid_t wait (int * status);`
  - `pid_t waitpid (pid_t pid, int * status, int options);`
  - Both these return process id on successful return or -1 in case of an error
- Otherwise the parent process might terminate before the child process terminates!!!



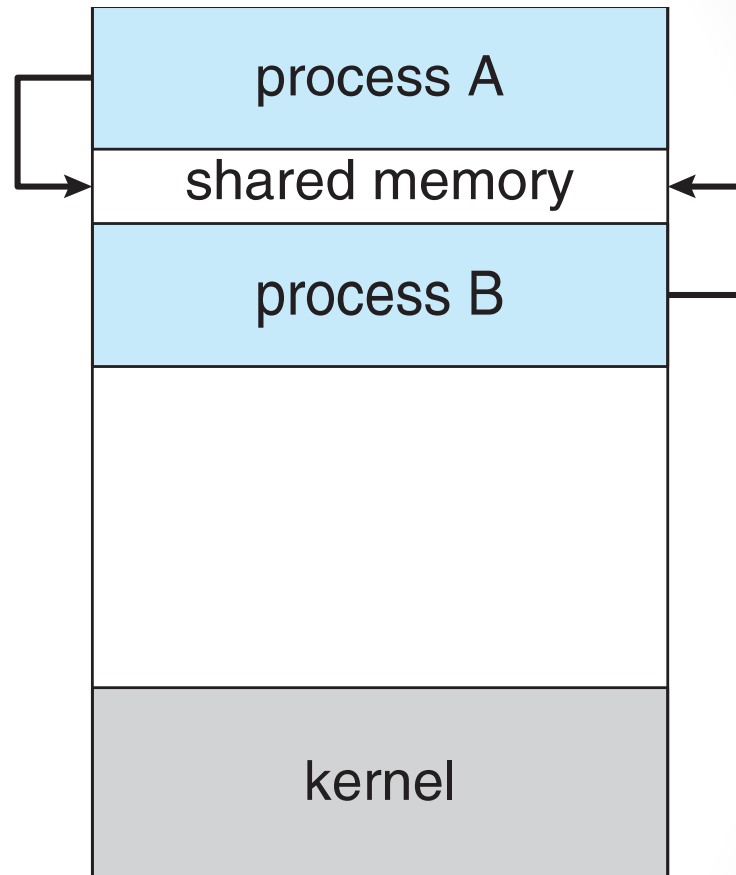
# Interprocess Communication

- Example: Chrome browser
- Cooperating processes need **interprocess communication (IPC)**
- Two models of IPC
  - **Shared memory**
  - **Message passing**

# Communication Models



(a)



(b)

(a) Message passing. (b) shared memory.

# Message Passing

- If processes  $P$  and  $Q$  wish to communicate, they need to:
  - Establish a *communication link* between them
  - Exchange messages via send/receive
- Implementation issues:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

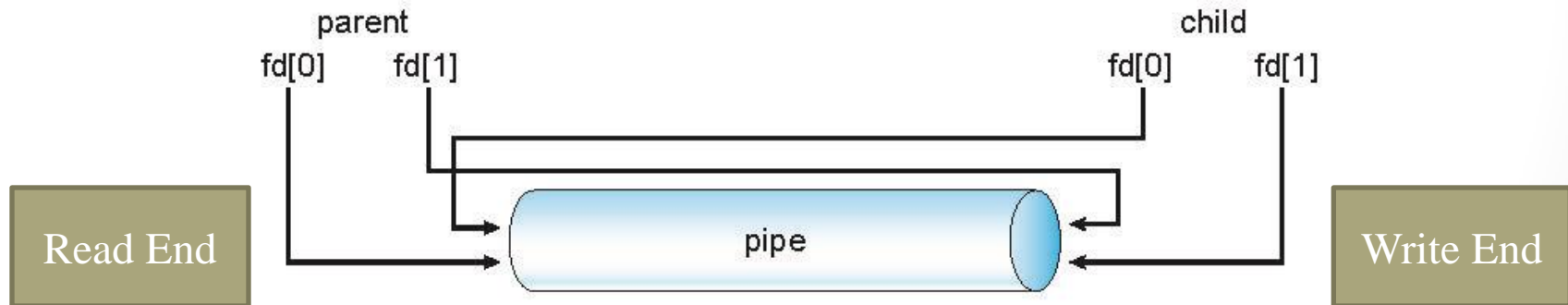


# Message Passing

- Implementation of communication link
  - Physical:
    - Shared memory
    - Hardware bus
    - Network
  - Logical:
    - Direct or indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering

# Ordinary Pipes

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the **write-end** of the pipe)
- Consumer reads from the other end (the **read-end** of the pipe)
- Ordinary pipes are therefore unidirectional
- Require parent-child relationship between communicating processes

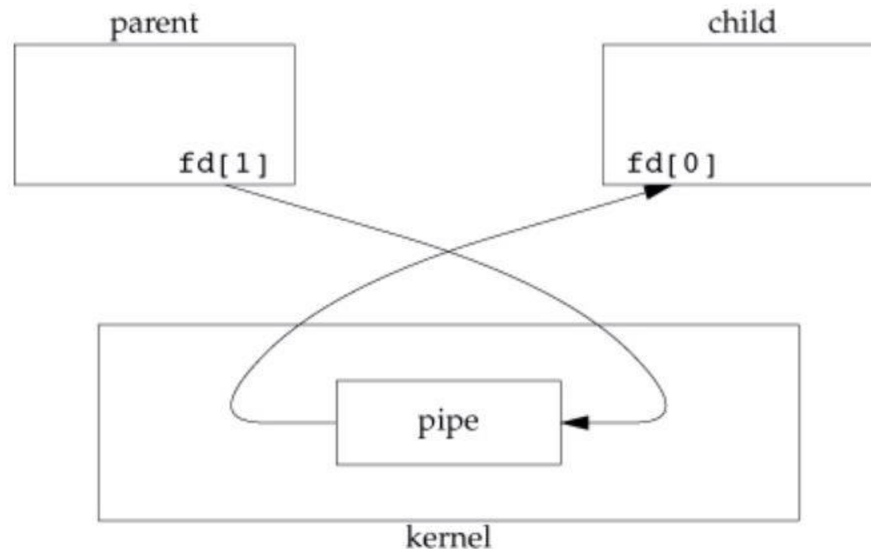


- Windows calls these **anonymous pipes**

# Pipes: creation and setup

```
#include <unistd.h>

int pipe(int fd[2]);
```



- The data in the pipe flows through the kernel.
- Normally, the process that calls pipe then calls fork, creating an IPC channel from the parent to the child, or vice versa.

# Pipes: creation and setup

```
#include "apue.h"

int
main(void)
{
    int      n;
    int      fd[2];
    pid_t    pid;
    char      line[MAXLINE];

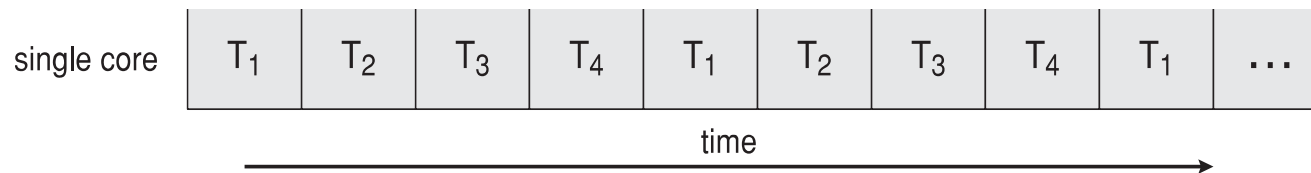
    if (pipe(fd) < 0)
        err_sys("pipe error");
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {          /* parent */
        close(fd[0]);
        write(fd[1], "hello world\n", 12);
    } else {                      /* child */
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
    exit(0);
}
```

# Named Pipes

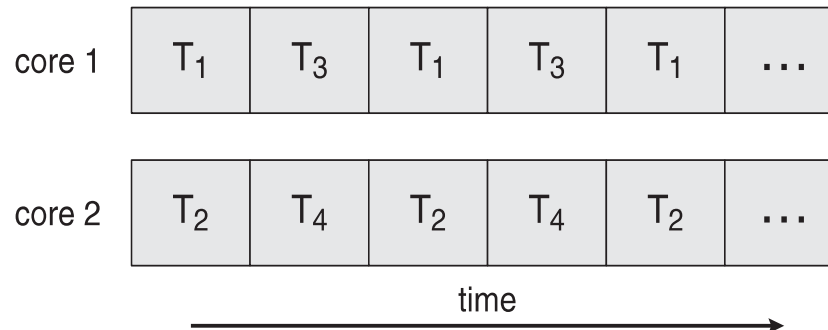
- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems
- Example: FIFO

# Concurrent vs. Parallel computing

- **Concurrent execution on single-core system:**
  - Supports more than one task by allowing all the tasks to make progress



- **Parallelism on a multi-core system:**
  - Perform more than one task simultaneously



# Multicore or Multiprocessor

- Increasing number of processing cores on computer systems
- Parallelism can be achieved
- Decrease the execution time
- What is the potential performance gain from adding another computing core? (AMDAHL'S LAW)

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- $S$  is serial portion
- $N$  processing cores
- If  $S=40\%$  and  $N$  grows very large what is maximum speed up?
  - 2.5 times

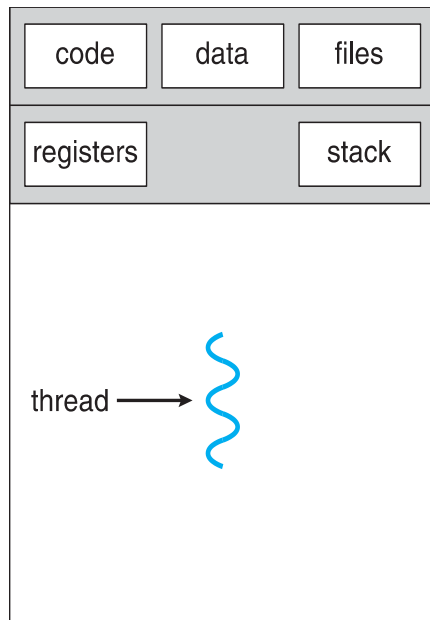
# When multi-process architecture makes sense?

- Web Server Example
  - If you are designing a web server, you need to constantly listen to possible incoming requests
  - Also there could be 1000 of requests every second, how can you address them all in a timely fashion?
- Word Editor Example
  - Allow user to work on a very large file while providing spell checking in the background (without pausing the editing)
  - Apply the keystrokes to the document
  - Automatically saving it without pausing the user work

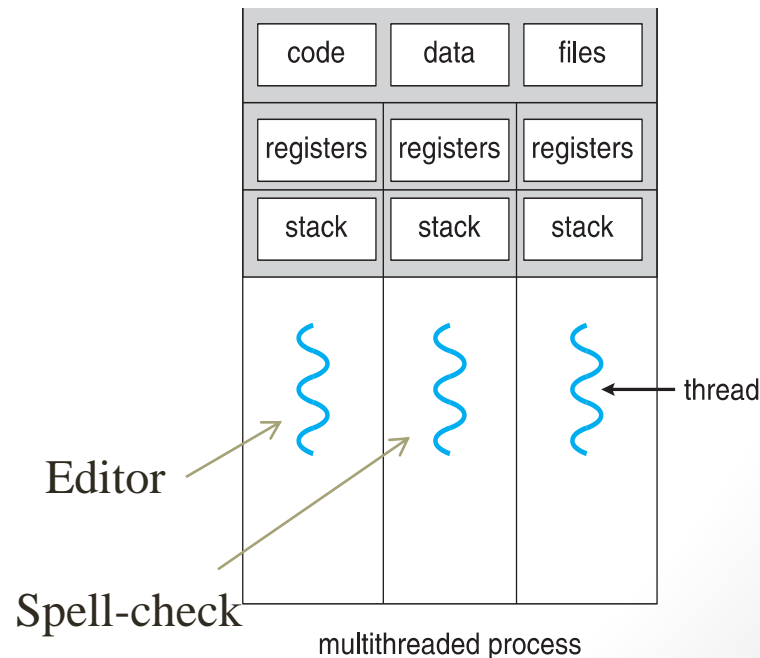


# Multi-threaded Execution

- Let's think about the word editor example again
  - What if the spell checker thread corrects a word spelling and at the same time the user is changing that word in the editor thread?
  - What can go wrong here?
    - Inconsistency in the data section



single-threaded process

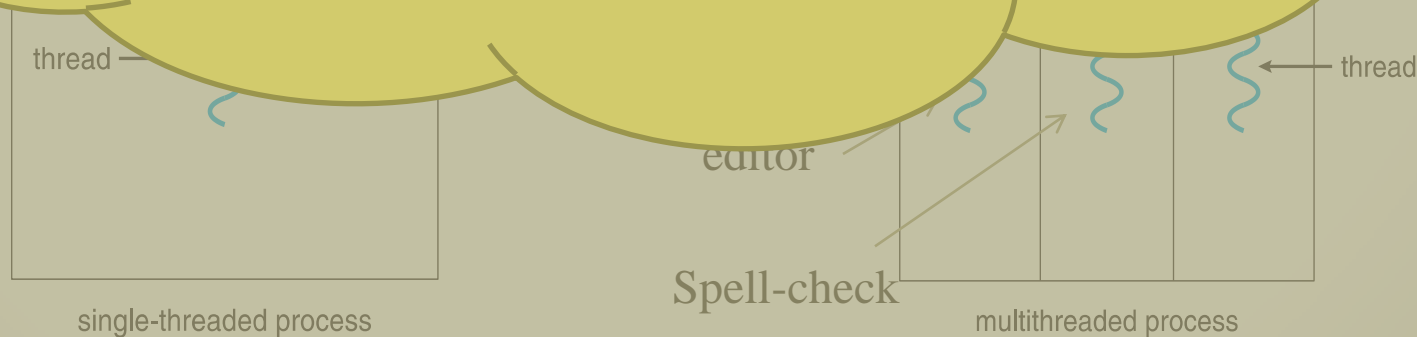


multithreaded process

# Multi-threaded Execution

- Let's think about the word editor example again
  - What if the spell checker thread corrects a word spelling and at the same time the user

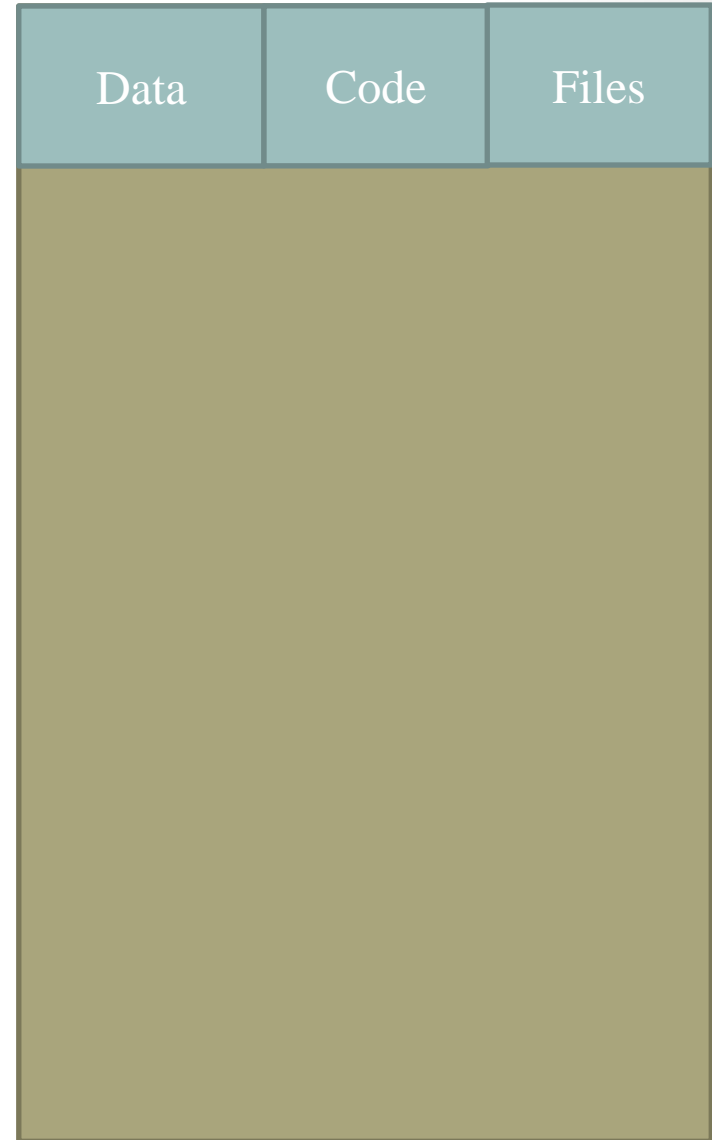
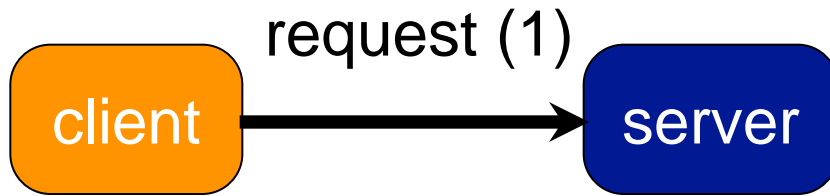
Can we mitigate this inconsistency?  
YES, We can avoid this using  
synchronization techniques that we  
will see in chapter 5.



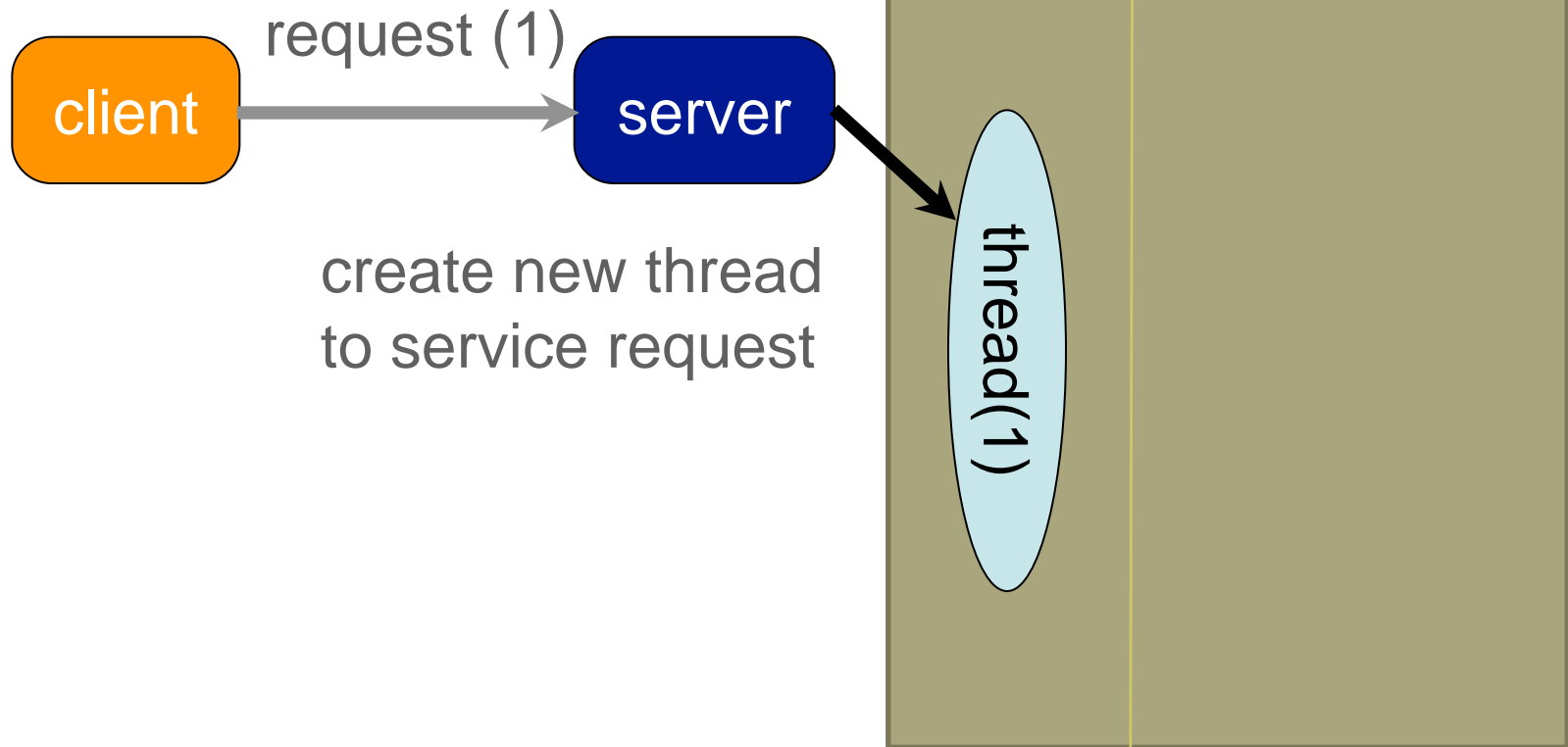
# Web Server Example

- Assume the web server is servicing search request (google search engine)
- Each request to be served is of similar nature (repetitive code) and has to work on the same information (repetitive data)
- What if we could have multiple executions of the same search code within the same process?
- Let's see how it looks!

# Multithreaded Server Architecture



# Multithreaded Server Architecture

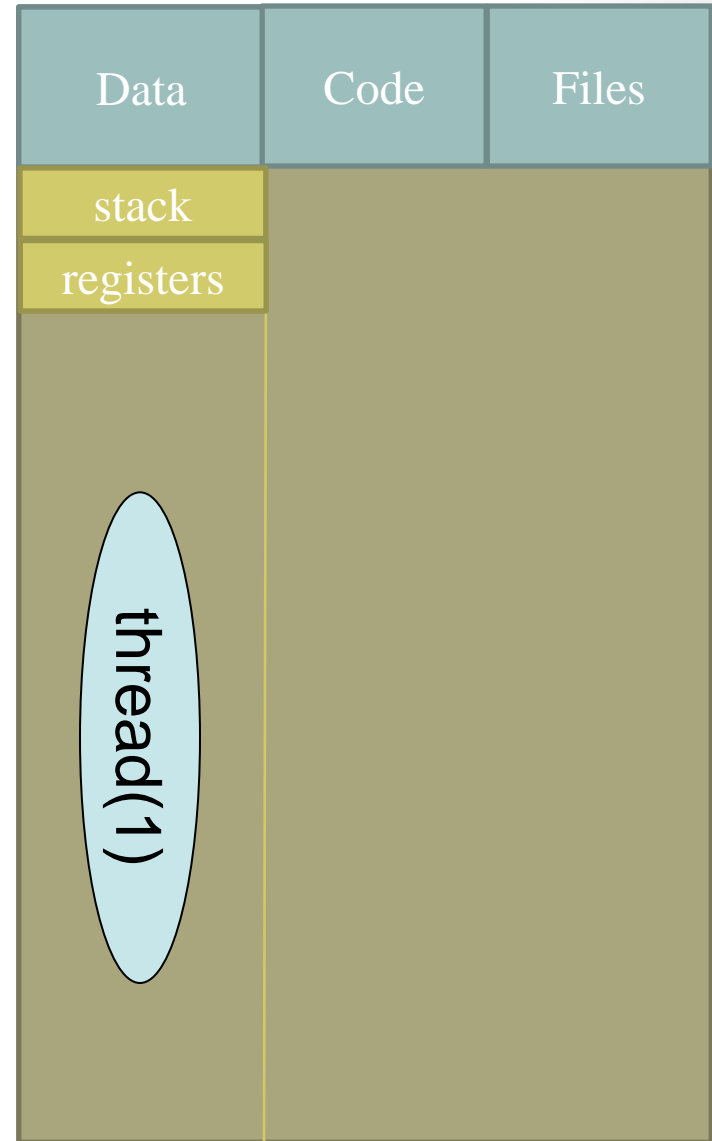


# Multithreaded Server Architecture

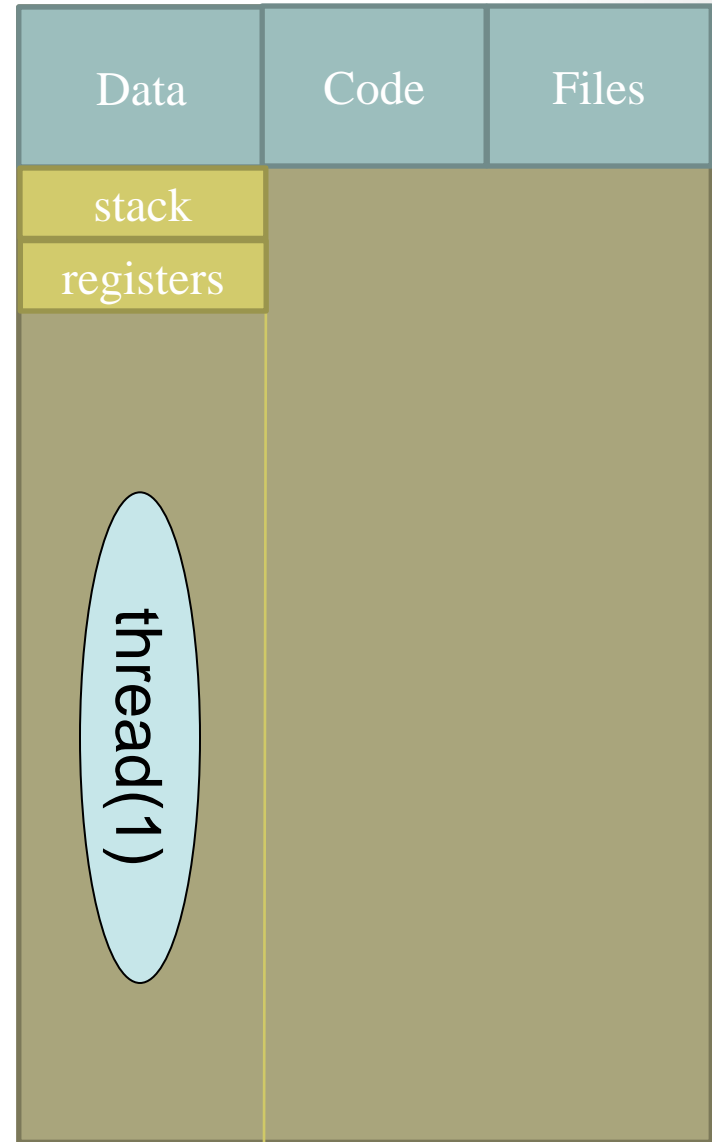
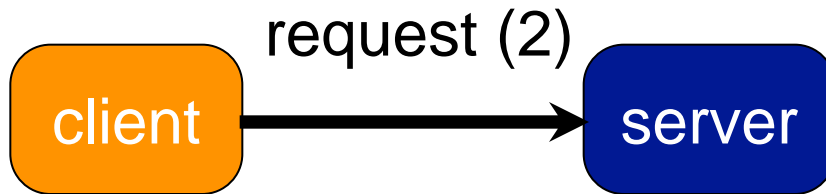
client

server

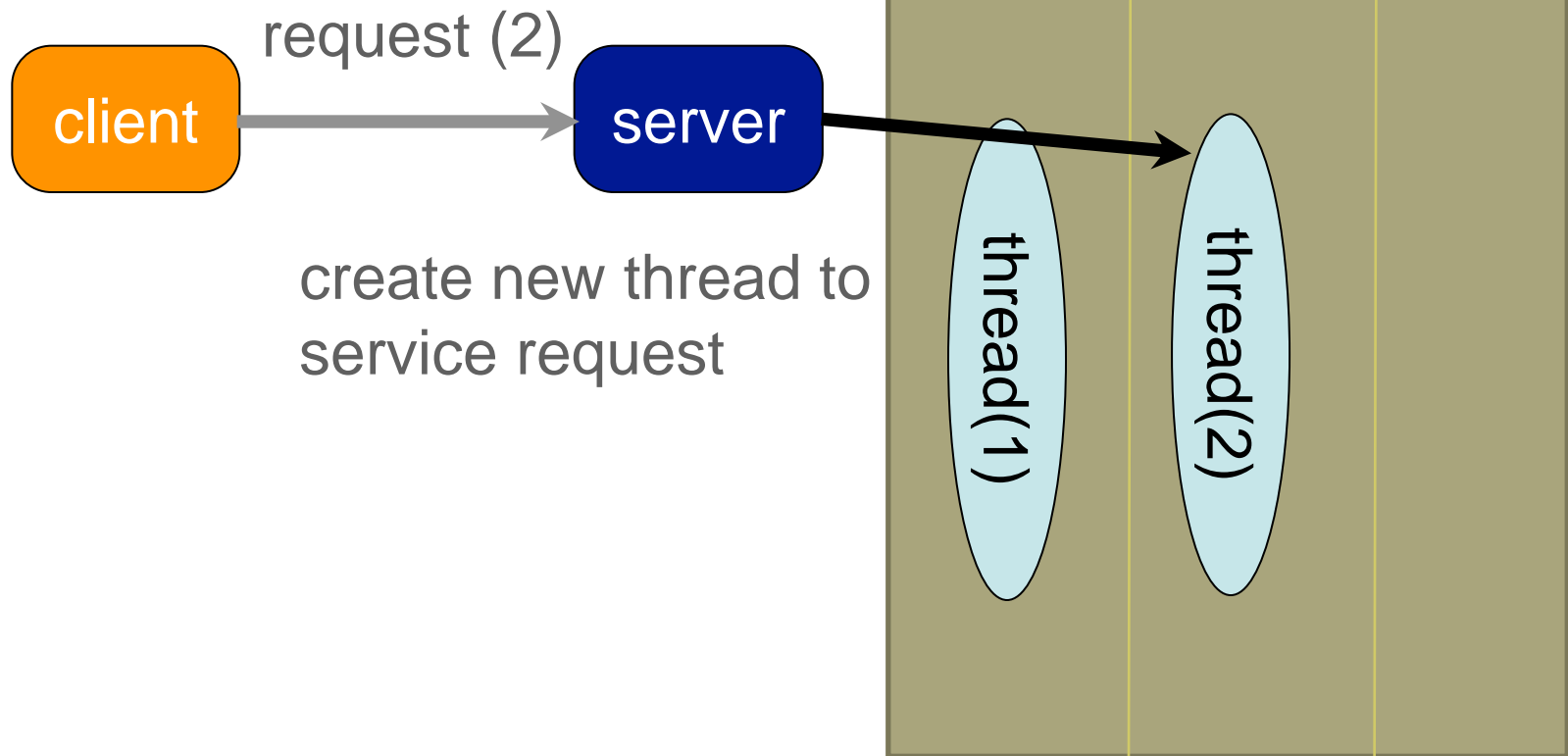
resume  
listening for  
new requests



# Multithreaded Server Architecture



# Multithreaded Server Architecture



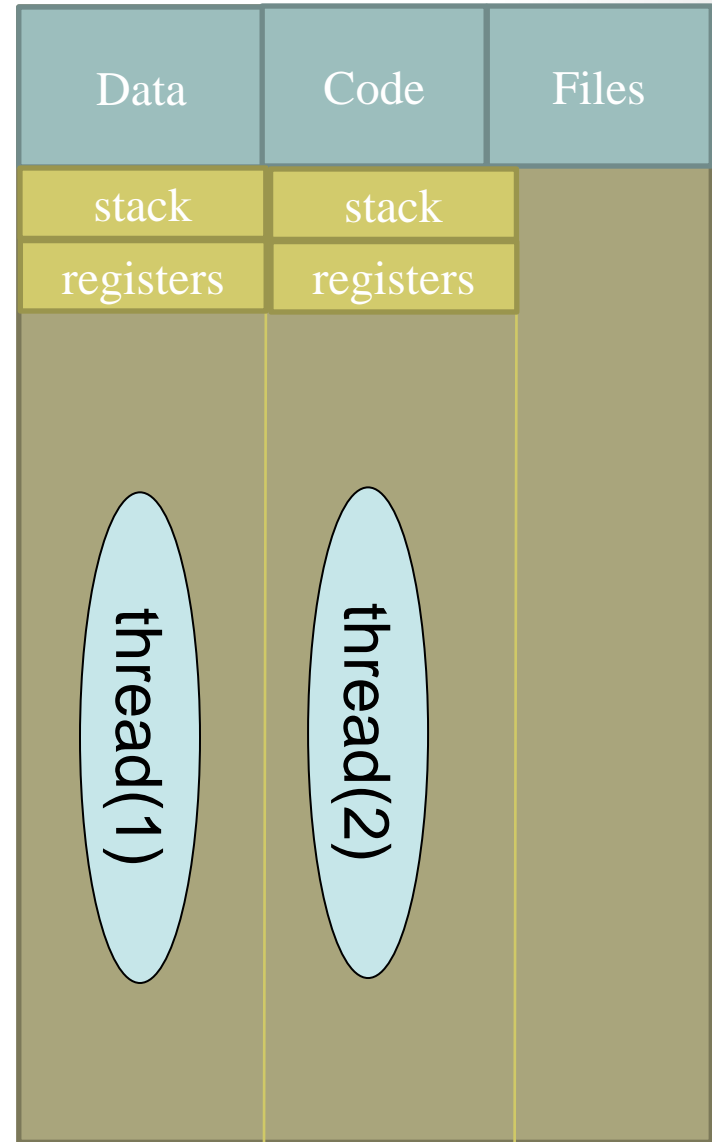


# Multithreaded Server Architecture

client

server

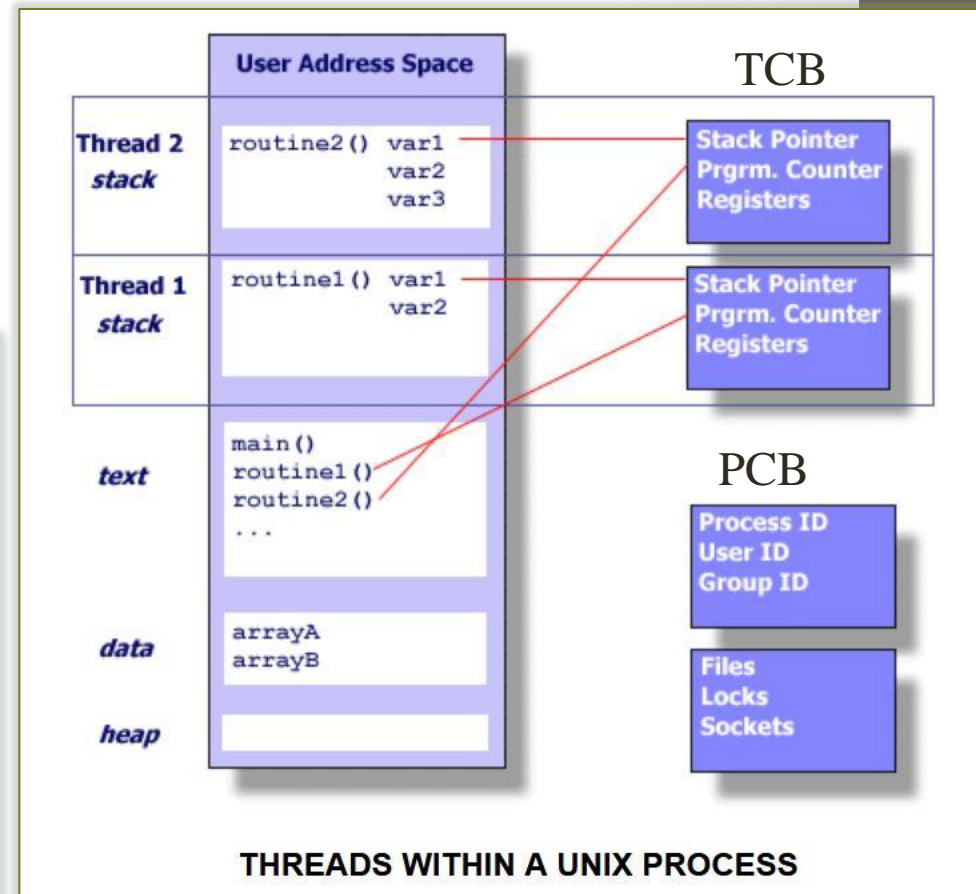
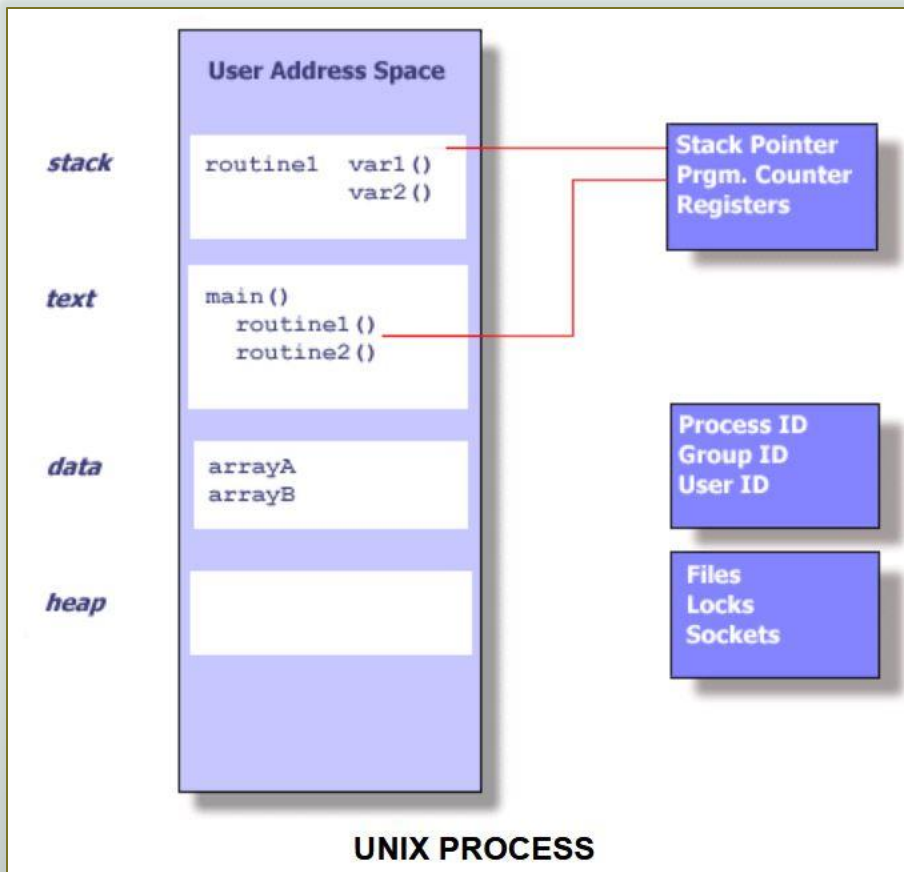
resume  
listening for  
new requests



# Threads vs. Processes

- What are the advantages of Threads over Processes
  - Light weight
  - Processes are costly to create (around 30X time more time)
  - Context switching processes can take up to 5X more time
  - More Efficient
  - Sharing data is easier
  - PCBs are large data types while TCB are way smaller
- What are the disadvantages and challenges of Threads
  - We need consistency when accessing the shared data
  - We should implement synchronization method among threads accessing the shared data
  - Can complicate execution

# PCB vs TCB



# Why Threads?

- No Inter Process Communication (IPC) is necessary
- The only limit is the memory bandwidth which is way more than the shared memory bandwidth as an IPC among processes

Platform	MPI Shared Memory Bandwidth (GB/sec)	Pthreads Worst Case Memory-to-CPU Bandwidth (GB/sec)
Intel 2.6 GHz Xeon E5-2670	4.5	51.2
Intel 2.8 GHz Xeon 5660	5.6	32
AMD 2.3 GHz Opteron	1.8	5.3
AMD 2.4 GHz Opteron	1.2	5.3
IBM 1.9 GHz POWER5 p5-575	4.1	16
IBM 1.5 GHz POWER4	2.1	4
Intel 2.4 GHz Xeon	0.3	4.3
Intel 1.4 GHz Itanium 2	1.8	6.4

# How are Threads Scheduled?

- Assume process P creates 5 threads T\_1, T\_2, T\_3, T\_4, T\_5 in this order T\_2, T\_1, T\_3, T\_5, T\_4
  - Which one of these threads executes first?
  - Which one of these threads finish its execution first?
  - On which core is thread T\_3 scheduled to run? (if there are 4 cores)
  - The answer to all these questions is **WE DON'T KNOW**
- A good multi-threaded program successful execution should be independent of order of execution of its threads
- What can we control?
  - The pthreads API provides several routines that may be used to specify how threads are scheduled for execution
    - FIFO (first-in first-out)
    - RR (round-robin)
    - OTHER (operating system determines)
  - pthreads API also provides the ability to set a thread's scheduling priority value.
  - The Linux operating system may provide a way to set the CPU core to execute the process on using the [sched\\_setaffinity](#) routine.

# Threads Synchronization

- If `main()` finishes before the created threads exit, all of the threads will be terminated because the main thread of execution is terminated
- How can we avoid this?
  - If main thread calls `pthread_exit()` as the last thing it does, `main()` will **block and be kept alive** to support the threads it created until they are done.
  - Using `pthread_join(.)` can block the thread to wait for the spawned threads

# User Threads and Kernel Threads

- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

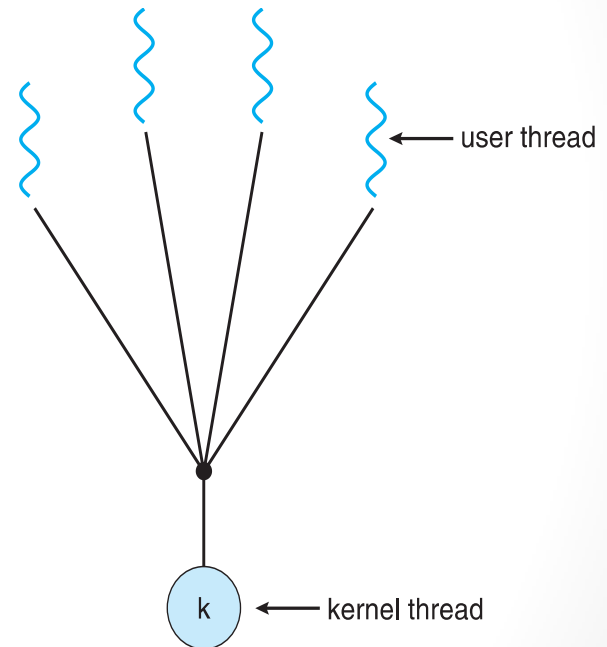
# Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many



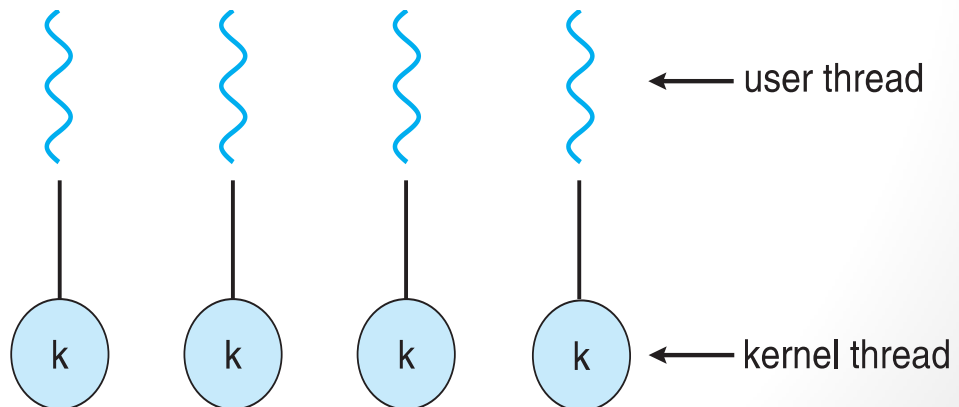
# Many to One

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**



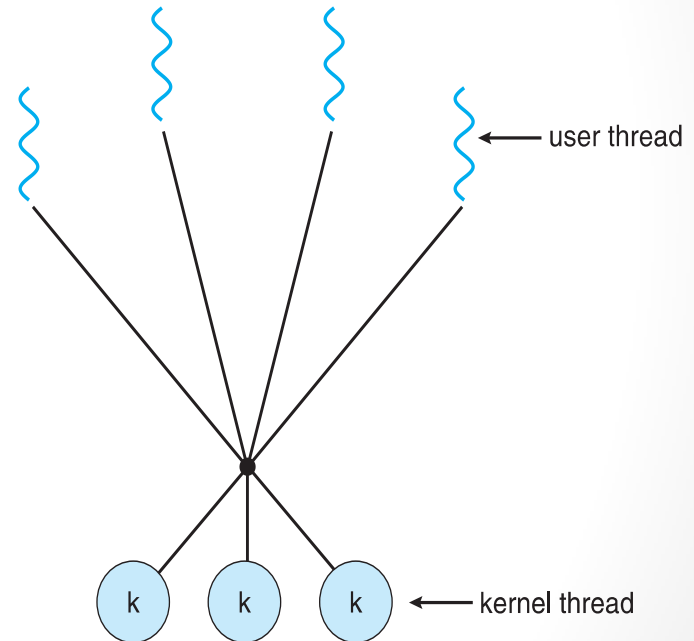
# One to One

- Each user-level thread maps to a kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later



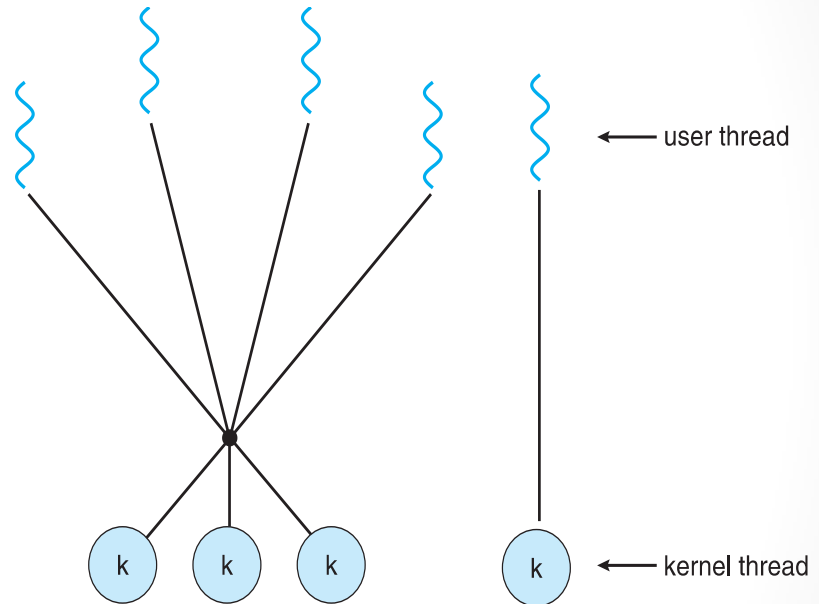
# Many-to-Many

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads (thread pool)
- Solaris prior to version 9
- Windows with the *ThreadFiber* package



# Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
  - Tru64 UNIX
  - Solaris 8 and earlier



# Create Threads

## NAME

**pthread\_create** - create a new thread

## SYNOPSIS

**#include <pthread.h>**

```
int pthread_create(pthread_t *thread,  
                  const pthread_attr_t *attr,  
                  void *(*start_routine) (void *),  
                  void *arg);
```

**Compile and link with `-pthread`**

- `(*start_routine)` is a function pointer to a function that returns a `void *` and has one argument of type `void *`
- `pthread_t` is a unsigned long (%lu)

# Thread Termination

- If any thread within a process calls `exit`, then the entire **process** terminates
- A thread can exit in three ways
  1. Return from the start routine. The return value is the thread's exit code
  2. The thread can be canceled by another thread in the same process
  3. The thread can call `pthread_exit`
- ❖ To allow other threads to continue execution, the main thread should terminate by calling `pthread_exit()` rather than `exit()`.

## NAME

**`pthread_exit` – terminate calling thread**

**`void pthread_exit (void *retval);`**

# Thread Join and Return Value

- If a thread has return values from its start routine, it can send it to other threads in the process by calling `pthread_exit (void* retval)` or simply returning a pointer of type `void *` to the return value
- `retval` is a type-less pointer like the input argument for `pthread_create`
- How can other threads access this value?
  - If a thread needs an input argument from another thread it can use **join function** to block its execution until the other thread exits

## NAME

**`pthread_join`** – calling thread will block until the specific thread calls `pthread_exit`

**`pthread_join (pthread_t tid, void **retval_ptr);`**

**`retval_ptr` has the return value of the thread with ID `tid`**

# Thread Input Argument and Return Value (Output Argument)

- The typeless pointer passed to `pthread_create` and `pthread_exit` can be used to pass the address of a structure containing more complex information.
- Be careful that the memory used for the structure is still valid when the caller has completed.
  - If the **input structure** was allocated on the caller's stack, for example, the memory contents might have changed by the time the structure is used.
  - If a thread allocates an **output structure** on its stack and passes a pointer to this structure to `pthread_exit`, then the stack might be destroyed and its memory reused for something else by the time the caller of `pthread_join` tries to use it.



# Process Synchronization!

# Critical Section Problem

- General Structure of a Process  $P_i$

```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (true);
```

# Producer-Consumer Inconsistency

register<sub>1</sub>

5

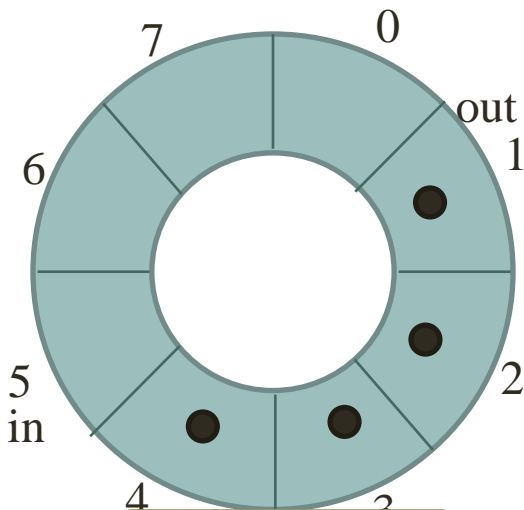
→ register<sub>1</sub> = counter  
register<sub>1</sub> = register<sub>1</sub> + 1  
counter = register<sub>1</sub>

counter

?

Process i  
in = 4  
counter = 4

```
while (true) {  
    /* produce an item in i */  
  
    while (counter == BUFFER.SIZE)  
        ; /* do nothing */  
  
    buffer[in] = next produced;  
    in = (in + 1) % BUFFER.SIZE;  
    counter++;  
}
```



register<sub>2</sub>

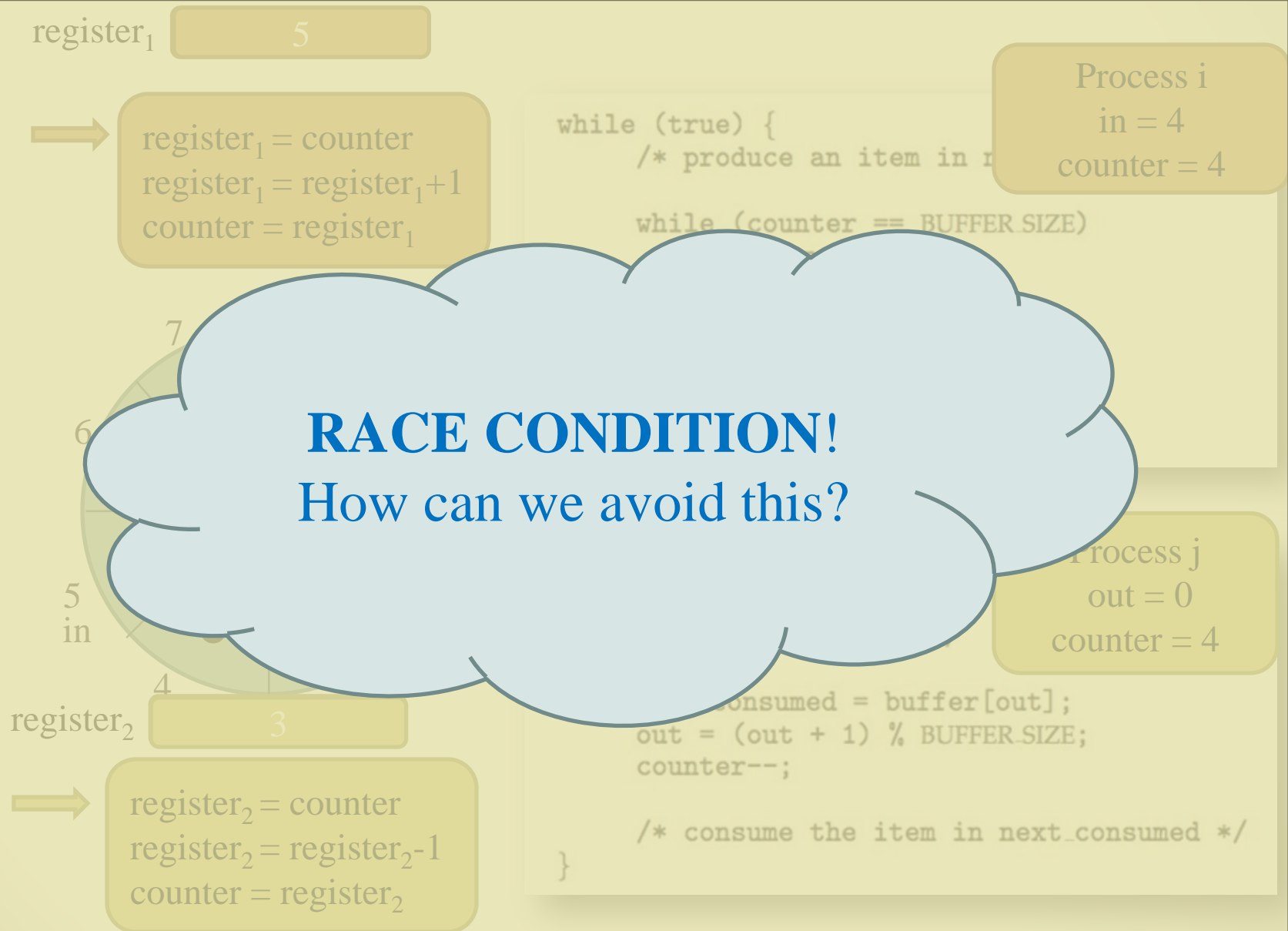
3

→ register<sub>2</sub> = counter  
register<sub>2</sub> = register<sub>2</sub> - 1  
counter = register<sub>2</sub>

```
while (true) {  
    while (counter == 0)  
        ; /* do nothing */  
  
    next_consumed = buffer[out];  
    out = (out + 1) % BUFFER.SIZE;  
    counter--;  
  
    /* consume the item in next_consumed */  
}
```

Process j  
out = 0  
counter = 4

# Producer-Consumer Inconsistency



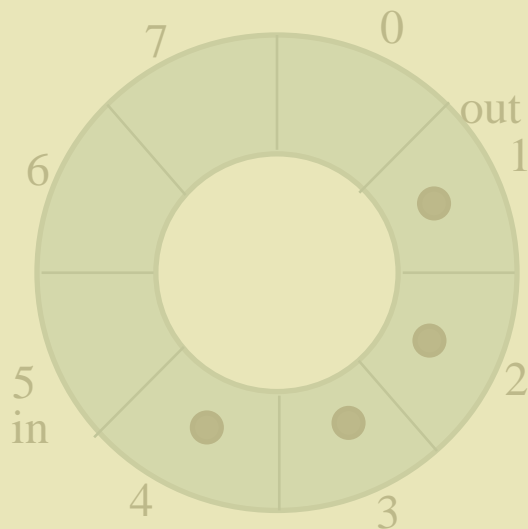
# Solution (1): No interrupts



```
register1 = counter  
register1 = register1 + 1  
counter = register1
```

No interrupts

Process i  
in = 4  
counter = 4



```
register2 = counter  
register2 = register2 - 1  
counter = register2
```

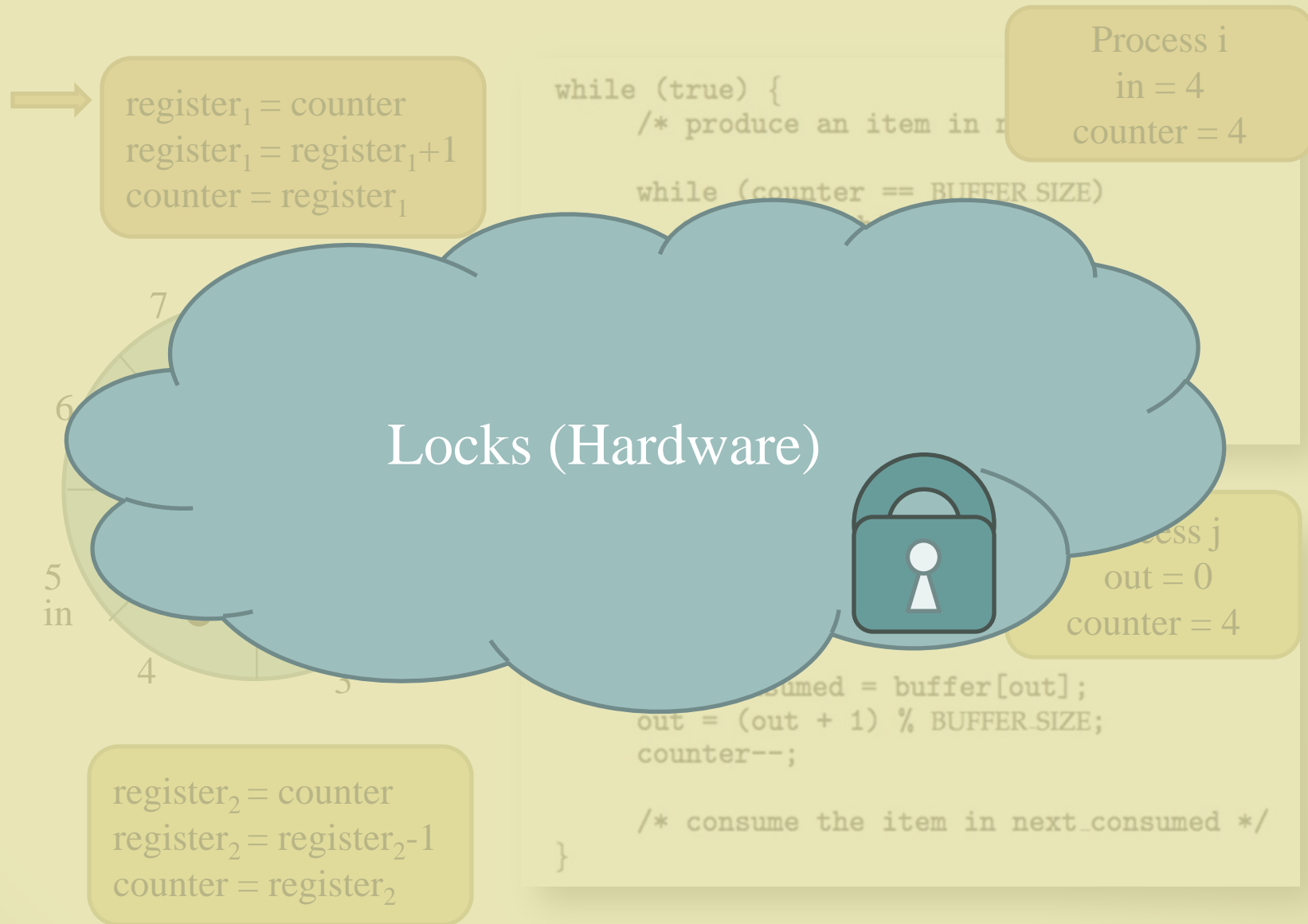
```
while (true) {  
    while (counter == BUFFER_SIZE)  
        ; /* do nothing */  
    buffer[in] = next produced;  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}
```

```
while (true) {  
    while (counter == 0)  
        ; /* do nothing */  
    next_consumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    counter--;
```

Process j  
out = 0  
counter = 4

```
    next_consumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    counter--;  
    /* consume the item in next_consumed */  
}
```

# Solution (2): Atomic Operations



# Valid Solution Properties

1. **Mutual Exclusion** - If process  $P_i$  is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
  - Assume that each process executes at a nonzero speed
  - No assumption concerning **relative speed** of the  $n$  processes

# Peterson's Solution

- **flag** is used to indicate if a process is ready to enter its CS
- **turn** specifies whose turn it is to enter its CS

Process i

```
do{  
    → flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    CS  
    flag[i] = false;  
    RS  
}while(true);
```

Process j

```
do{  
    → flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i);  
    CS  
    flag[j] = false;  
    RS  
}while(true);
```

Mutual Exclusion: Pass

turn

**i or j**



# Peterson's Solution

- **flag** is used to indicate if a process is ready to enter its CS
- **turn** specifies whose turn it is to enter its CS

Process i

```
do{
    flag[i] = true;
    turn = j;
    → while (flag[j] && turn == j);
       CS
    flag[i] = false;
    RS
}while(true);
```

Process j

```
do{
    → flag[j] = true;
      turn = i;
    while (flag[i] && turn == i);
    →   CS
      flag[j] = false;
      RS
}while(true);
```

**Progress: Pass**  
**Bounded Waiting: Pass**

# Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
  - **Atomic** = non-interruptible
    - Either test memory word and set value
    - Or swap contents of two memory words

# Get Help from Hardware for Locks

- What was the problem here?!?

```
do {  
    while (lock);  
    lock = 1;  
        critical section  
    lock = 0;  
        remainder section  
} while (true);
```

# How to have a working lock?

- Can this be fixed if we were able to **test and set** the value of lock in one atomic (uninterruptible) operation?

```
do {  
    while (lock);  
    lock = 1;  
        critical section  
    lock = 0;  
        remainder section  
} while (true);
```

No interrupts

# Test and Set Instruction

- There is hardware support for such instructions
- The whole instruction will be executed as one uninterruptible unit of operation
- One example of such instructions: Test\_and\_Set

```
boolean test_and_set(boolean *target) {  
    boolean rv = *target;  
    *target = true;  
  
    return rv;  
}
```

Atomically

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.

# Lock using Test and Set

- lock initialized to false
- Let's use the test and set operation for implementing our lock!

```
do {  
    while (test_and_set(&lock))  
        ; /* do nothing */  
  
    /* critical section */  
  
    lock = false;  
  
    /* remainder section */  
} while (true);
```

**Mutual Exclusion: Pass**

**Bounded Waiting: ???**

# Compare and Swap Instruction

```
int compare_and_swap(int *value, int expected, int new_value) {  
    int temp = *value;  
  
    if (*value == expected)  
        *value = new_value;  
  
    return temp;  
}
```

1. Executed atomically
2. Returns the original value of passed parameter “value”
3. Set the variable “value” the value of the passed parameter “new\_value” but only if “value” == “expected”. That is, the swap takes place only under this condition.

# Lock using Compare and Swap

- lock initialized to 0

```
do {  
    while (compare and swap(&lock, 0, 1) != 0)  
        ; /* do nothing */  
  
    /* critical section */  
  
    lock = 0;  
  
    /* remainder section */  
} while (true);
```

**Mutual Exclusion: Pass**

**Bounded Waiting: ???**



# Mutex (Mutual Exclusion) Locks



- Solutions seen so far are complicated!
- So Operating Systems designers build software tools to solve CS problem
- A process should **acquire** the lock in the entry section then is allowed to enter its CS
- After the process is done, it should **release** its lock in the exit section

```
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (true);
```

# Mutex Lock

- The function acquire is a blocking operation
- Called also **spinlock**

```
acquire() {  
    while (!available)  
        ; /* busy wait */  
    available = false;;  
}
```

```
release() {  
    available = true;  
}
```



Atomically

# Counting Locks

- What if
  - we have **more than one copy** of the resource?
  - Or want to allow up to  $n$  processes into the critical section?
- We need a counting lock...

# Semaphores

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore  $S$  – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **wait()** and **signal()**
    - Originally called **P()** and **V()**

```
wait(S) {  
    while (S <= 0)  
        ; // busy wait  
    S--;  
}
```

```
signal(S) {  
    S++;  
}
```

Atomically

# Semaphores Continued

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can solve various synchronization problems
- Consider  $P_1$  and  $P_2$  that require  $S_1$  to happen before  $S_2$

Create a semaphore “**synch**” initialized to 0

**P1:**

$S_1;$

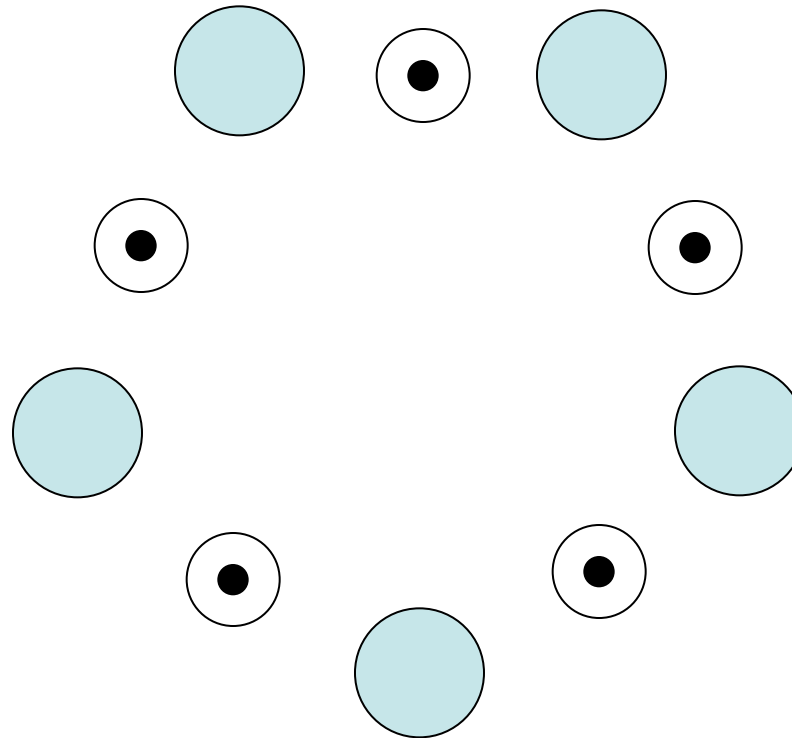
**signal(synch);**

**P2:**

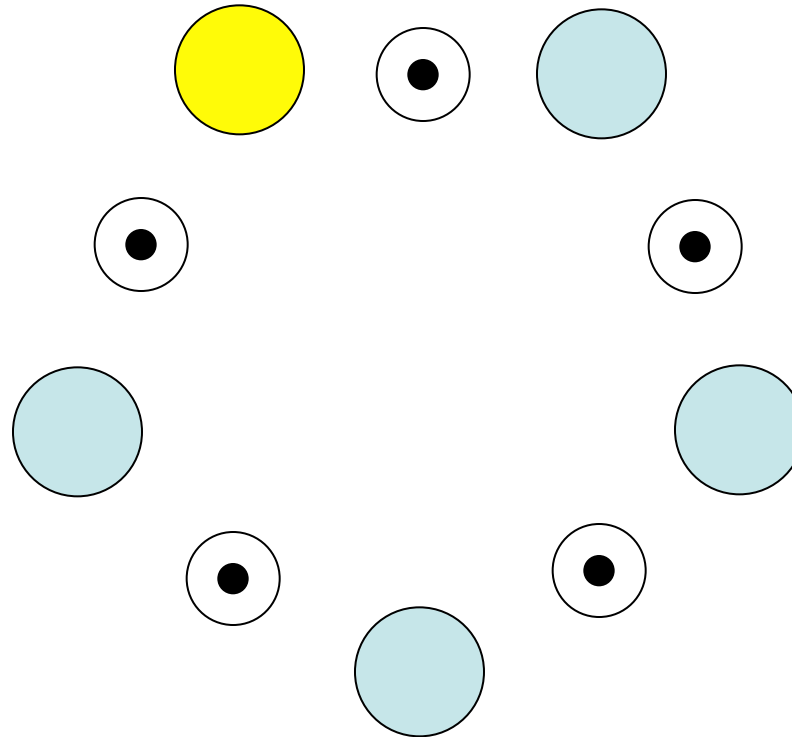
**wait(synch);**

$S_2;$

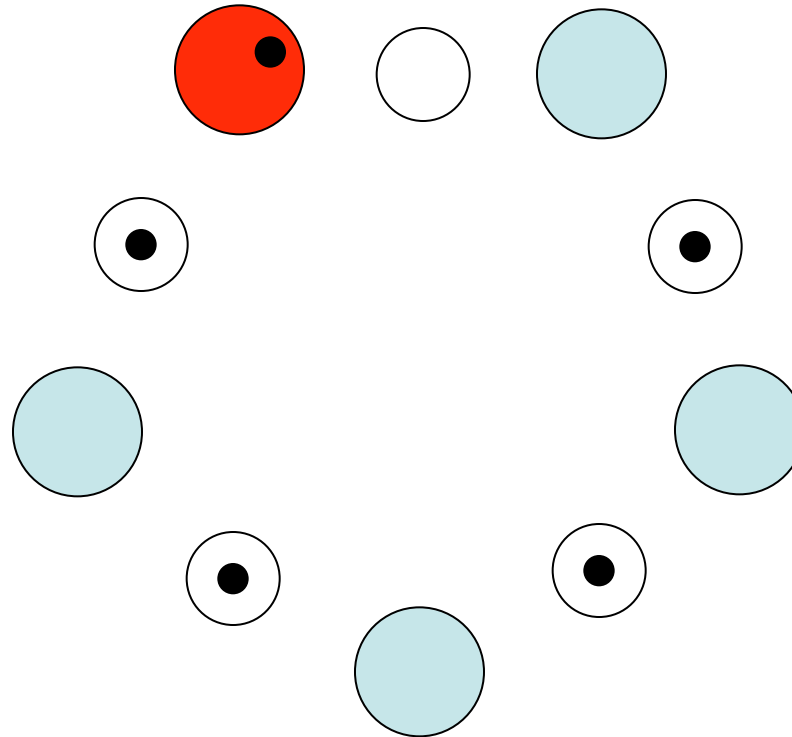
# The *Dining-Philosophers* Problem



# The *Dining-Philosophers* Problem

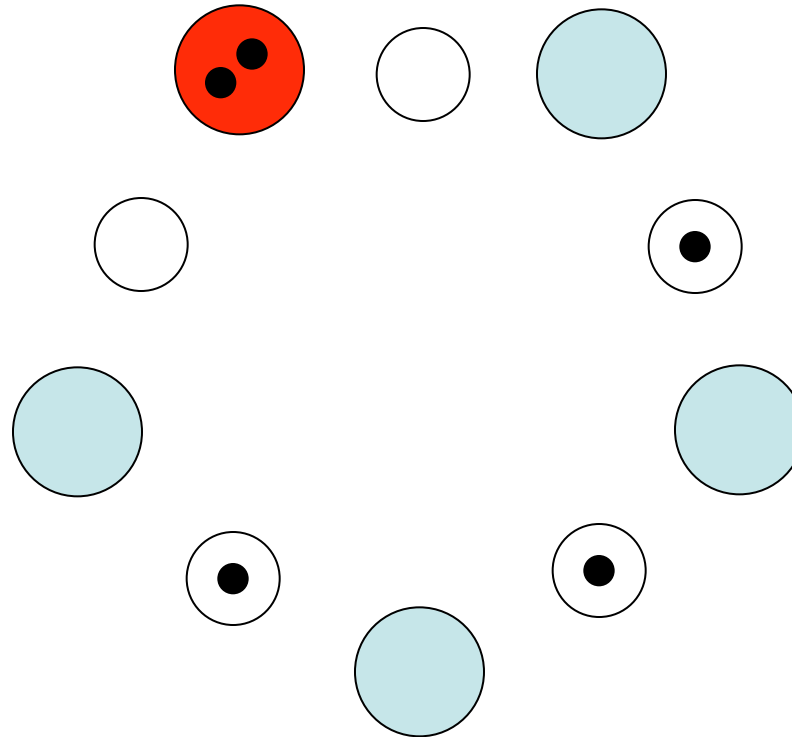


# The *Dining-Philosophers* Problem

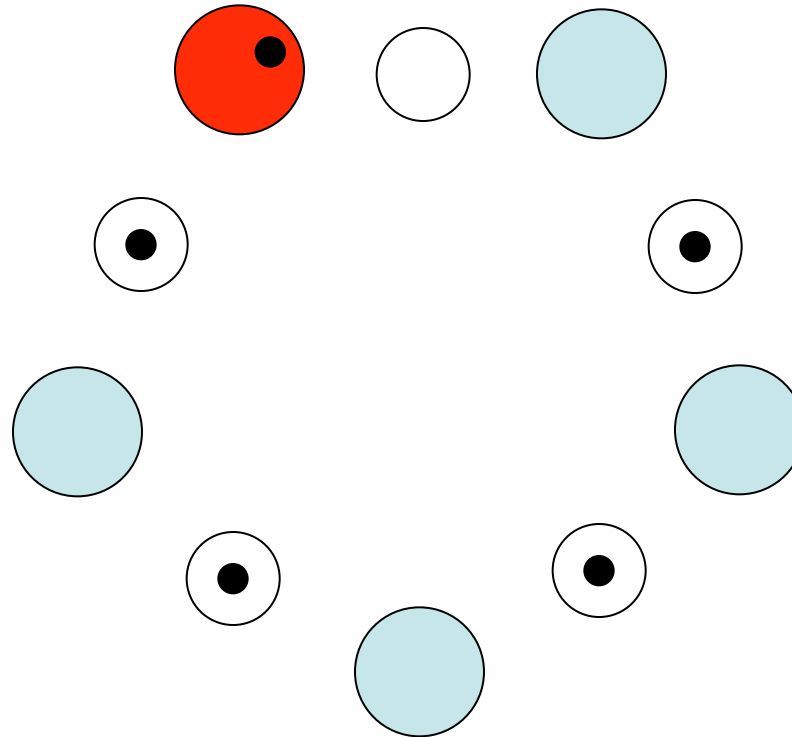




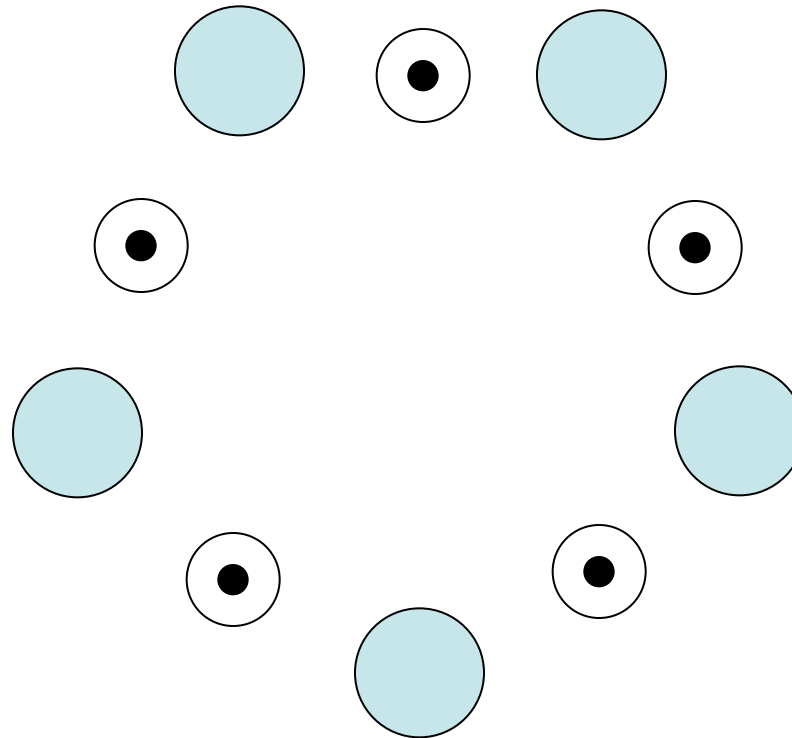
# The *Dining-Philosophers* Problem



# The *Dining-Philosophers* Problem



# The *Dining-Philosophers* Problem



# Limit to Concurrency

What is the maximum number of philosophers that can be eating at any point in time?

# Philosopher's Behavior

- Grab chopstick on left
- Grab chopstick on right
- Eat
- Put down chopstick on right
- Put down chopstick on left

How well does this work?

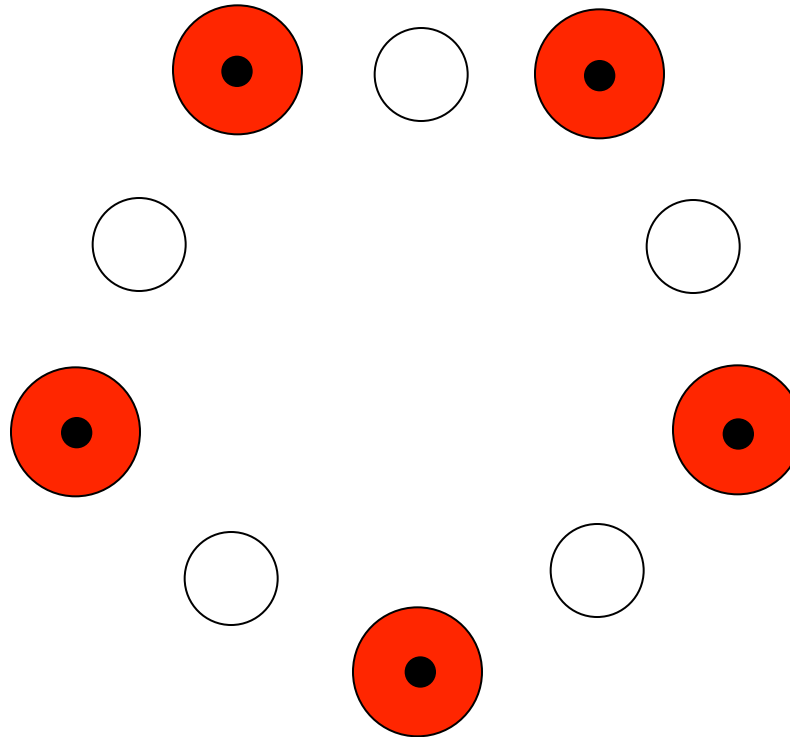
# Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do {  
    wait (chopstick[i] );  
    wait (chopstick[ (i + 1) % 5] );  
  
    // eat  
  
    signal (chopstick[i] );  
    signal (chopstick[ (i + 1) % 5] );  
  
    // think  
  
} while (TRUE);
```

- What is the problem with this algorithm?

# The *Dining-Philosophers* Problem



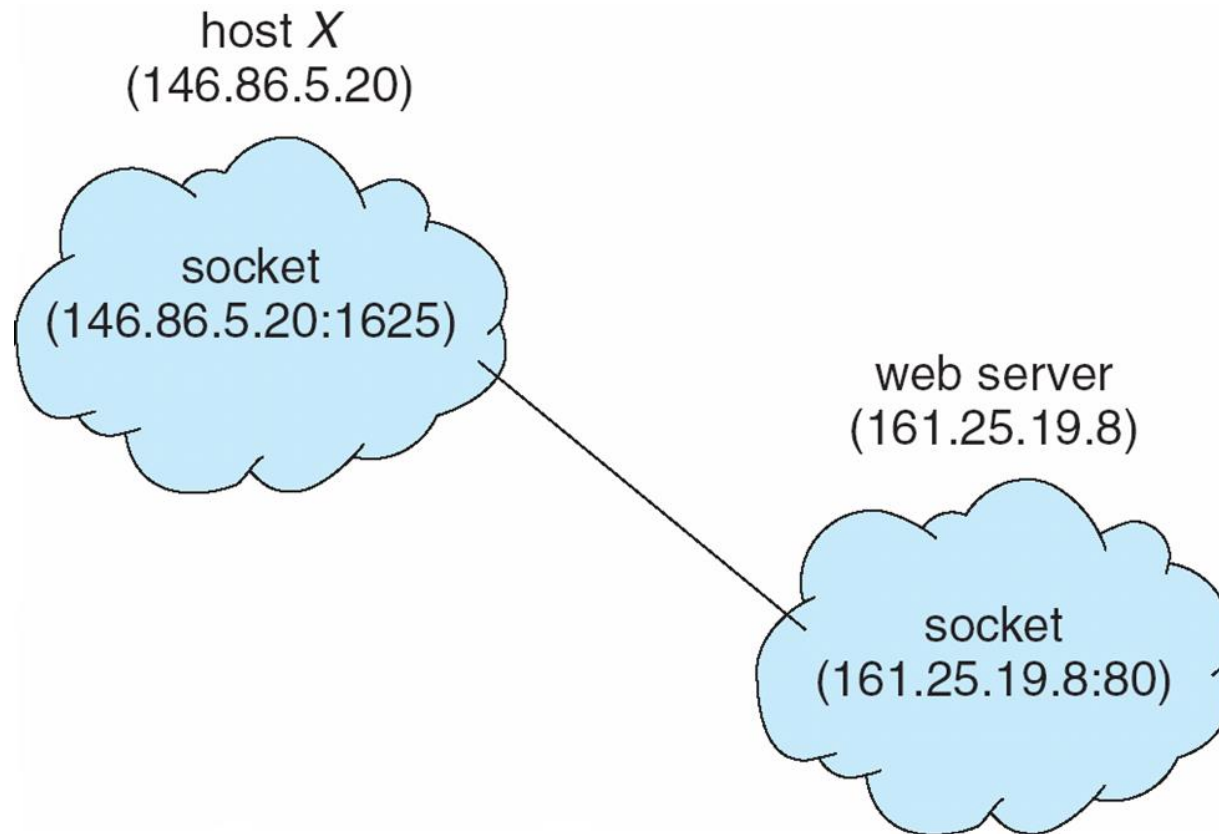
# Network Communications!



# Sockets

- How can two processes on two different machines talk to each other on the web?
- A **socket** is defined as an endpoint for communication
- Concatenation of IP address and **port** – a number included at start of message packet to differentiate network services on a host
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets
- All ports below 1024 are *well known*, used for standard services
- Special IP address 127.0.0.1 (**loopback**) to refer to system on which process is running

# Socket Communication



# Connection Types

- Two types of connection (transport layer)
  - **Connection-oriented (TCP)**
  - **Connectionless (UDP)**

# TCP Connections

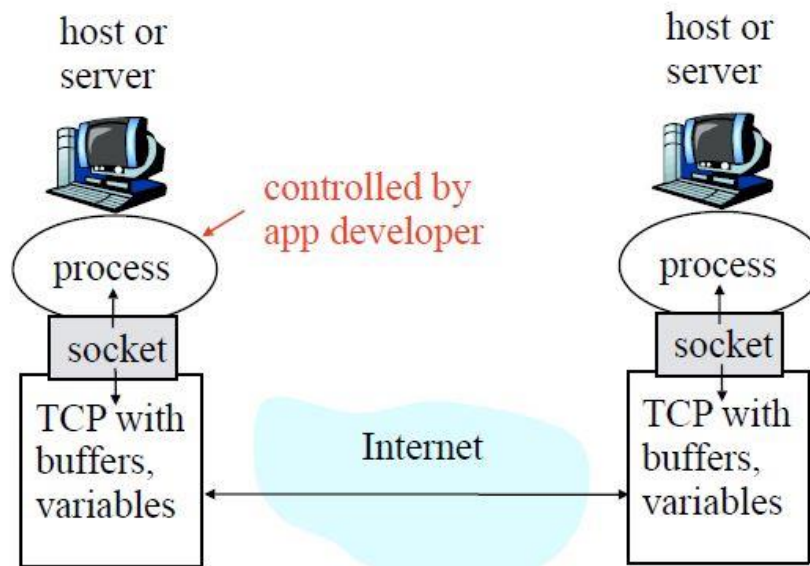
- Service
  - OSI Transport Layer
- Reliable byte stream (interpreted by application)
- 16-bit port space allows multiple connections on a single host
- Connection-oriented
  - Set up connection before communicating
  - Tear down connection when done

# TCP Service

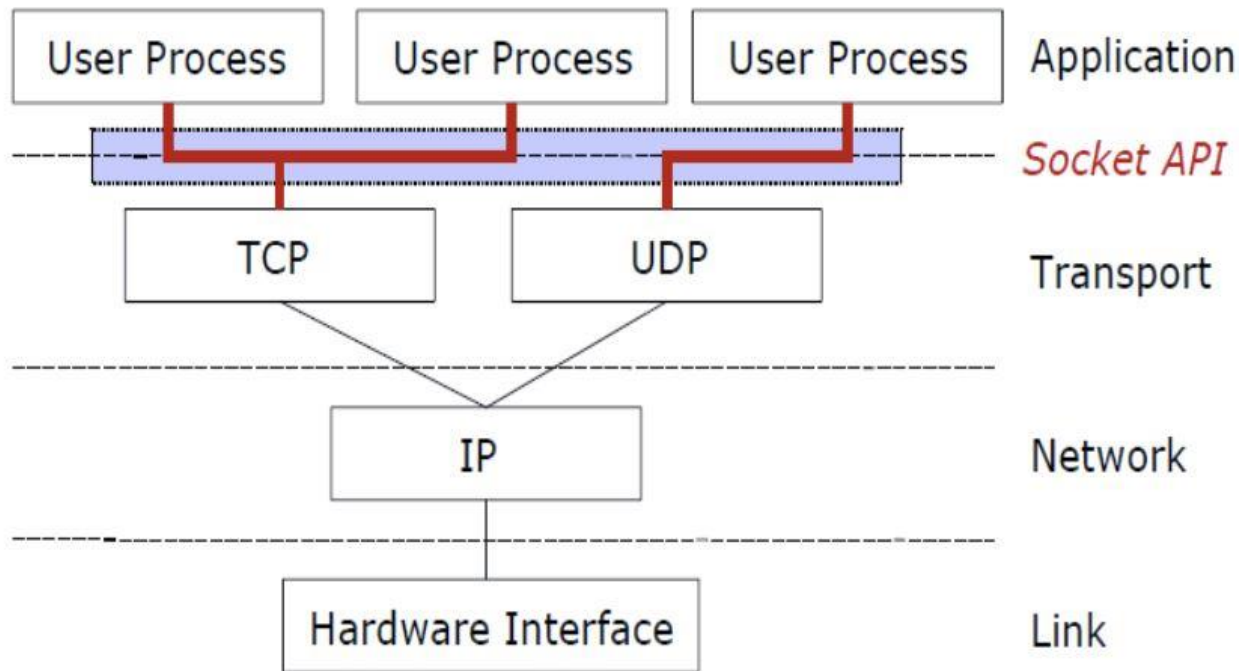
- Reliable Data Transfer
  - Guarantees delivery of all data
  - Exactly once if no catastrophic failures
- Sequenced Data Transfer
  - Guarantees in-order delivery of data
  - If A sends M1 followed by M2 to B, B never receives M2 before M1
- Regulated Data Flow
  - Monitors network and adjusts transmission appropriately
  - Prevents senders from wasting bandwidth
  - Reduces global congestion problems
- Data Transmission
  - Full-Duplex byte stream

# Sample TCP communication

- Transport Control Protocol (TCP)



# TCP connection from OSI P.O.V.



# TCP Connection Establishment

- Connection oriented (streams )
  - `sd = socket(PF_INET, SOCK_STREAM, 0);`

Default Protocol

```
#include <sys/socket.h>
```

```
int socket(int domain, int type, int protocol);
```

Returns: file (socket) descriptor if OK, -1 on error

Domain	Description
AF_INET	IPv4 Internet domain
AF_INET6	IPv6 Internet domain (optional in POSIX.1)
AF_UNIX	UNIX domain
AF_UNSPEC	unspecified

Type	Description
SOCK_DGRAM	fixed-length, connectionless, unreliable messages
SOCK_RAW	datagram interface to IP (optional in POSIX.1)
SOCK_SEQPACKET	fixed-length, sequenced, reliable, connection-oriented messages
SOCK_STREAM	sequenced, reliable, bidirectional, connection-oriented byte streams



# TCP Connection Establishment

- For the internet (PF\_INET) this corresponds to TCP
- `socket()` returns a socket descriptor, an int similar to a file descriptor
- For a server, we need to associate a well-known address with the server's socket on which client requests will arrive
- Clients need a way to discover the address to use to contact a server
  - server reserves an address and register it in `/etc/services`
  - Register with a name service

```
#include <sys/socket.h>
```

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t len);
```

# TCP Connection Establishment

- Use `connect()` on a socket that was previously created using `socket()`:

```
int connect(int sockfd, const struct sockaddr *addr, socklen_t len);
```

- If we're dealing with a connection-oriented network service, we need to create a connection between the socket of the process requesting the service (the client) and the process providing the service (the server)
- The address we specify with `connect` is the address of the server with which we wish to communicate. If `sockfd` is not bound to an address, `connect` will bind a default address for the caller.

# TCP Connection Establishment

```
struct sockaddr_in {  
    sa_family_t    sin_family;    /* address family */  
    in_port_t      sin_port;      /* port number */  
    struct in_addr  sin_addr;      /* IPv4 address */  
    unsigned char   sin_zero[8];   /* filler */  
};
```

# TCP: Client

- `socket()` create the socket descriptor
- `connect()` connect to the remote server.
- `read()`, `write()` communicate with the server
- `close()` end communication by closing socket descriptor

# TCP: Server

- `socket()` create the socket descriptor
- `bind()` associate the local address
- `listen()` wait for incoming connections from clients
- `accept()` accept incoming connection
- `read()`, `write()` communicate with client
- `close()` close the socket descriptor

# Listen

- A server announces that it is willing to accept connect requests by calling the listen function
- The backlog argument provides a hint to the system regarding the number of outstanding connect requests that it should enqueue on behalf of the process

```
int listen(int sockfd, int backlog);
```

# Accept Connections

- Once a server has called listen, the socket used can receive connect requests. We use the accept function to retrieve a connect request and convert it into a connection
- The file descriptor returned by accept is a socket descriptor that is connected to the client that called connect
- The original socket passed to accept is not associated with the connection, but instead remains available to receive additional connect requests

```
int accept(int sockfd, struct sockaddr *restrict addr,  
           socklen_t *restrict len);
```

Returns: file (socket) descriptor if OK, -1 on error

# Scheduling Algorithms



# First-Come, First-Served (FCFS)

<u>Process</u>	<u>Burst Time</u>
$P_1$	24
$P_2$	3
$P_3$	3

- Suppose that the processes arrive in the order:  $P_1, P_2, P_3$   
The **Gantt Chart** for the schedule is:



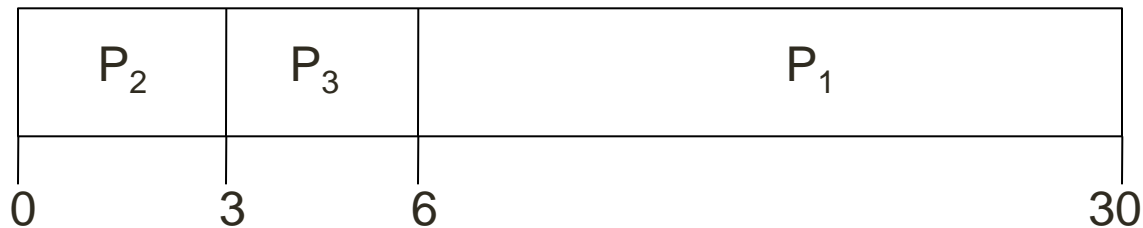
- Waiting time for  $P_1 = 0$ ;  $P_2 = 24$ ;  $P_3 = 27$
- Average waiting time:  $(0 + 24 + 27)/3 = 17$

# FCFS

Suppose that the processes arrive in the order

$P_2, P_3, P_1$

- The Gantt chart for the schedule is:



- Waiting time for  $P_1 = 6$ ;  $P_2 = 0$ ;  $P_3 = 3$
- Average waiting time:  $(6 + 0 + 3)/3 = 3$
- Much better than previous case.
- Convoy effect: all process are stuck waiting until a long process terminates.

# Shortest-Job-First (SJF)

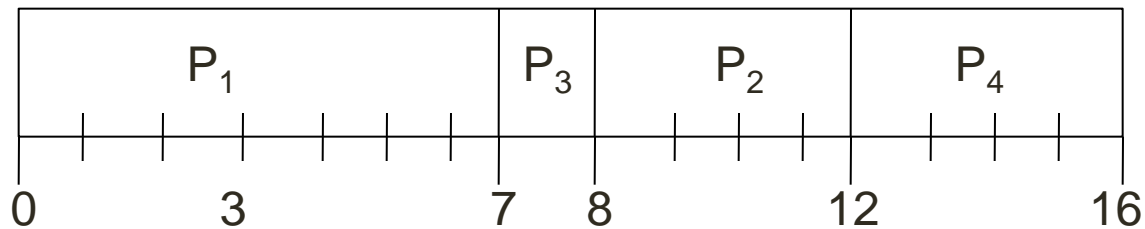
- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.
- Two schemes:
  - Nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst.
  - Preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is known as the Shortest-Remaining-Time-First (SRTF).
- SJF is **optimal** – gives minimum average waiting time for a given set of processes.

**Question:** Is this practical? How can one determine the length of a CPU-burst?

# Non-Preemptive SJF

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
$P_1$	0.0	7
$P_2$	2.0	4
$P_3$	4.0	1
$P_4$	5.0	4

- SJF (non-preemptive)

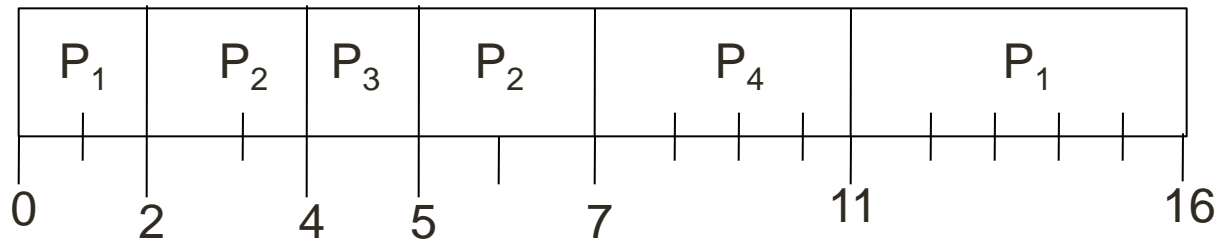


- Average waiting time =  $(0 + 6 + 3 + 7)/4 = 4$

# Preemptive SJF: Shortest Remaining Time First

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
$P_1$	0.0	7
$P_2$	2.0	4
$P_3$	4.0	1
$P_4$	5.0	4

- SJF (preemptive)



- Average waiting time =  $(9 + 1 + 0 + 2)/4 = 3$

# Determining Length of the Next CPU-Burst

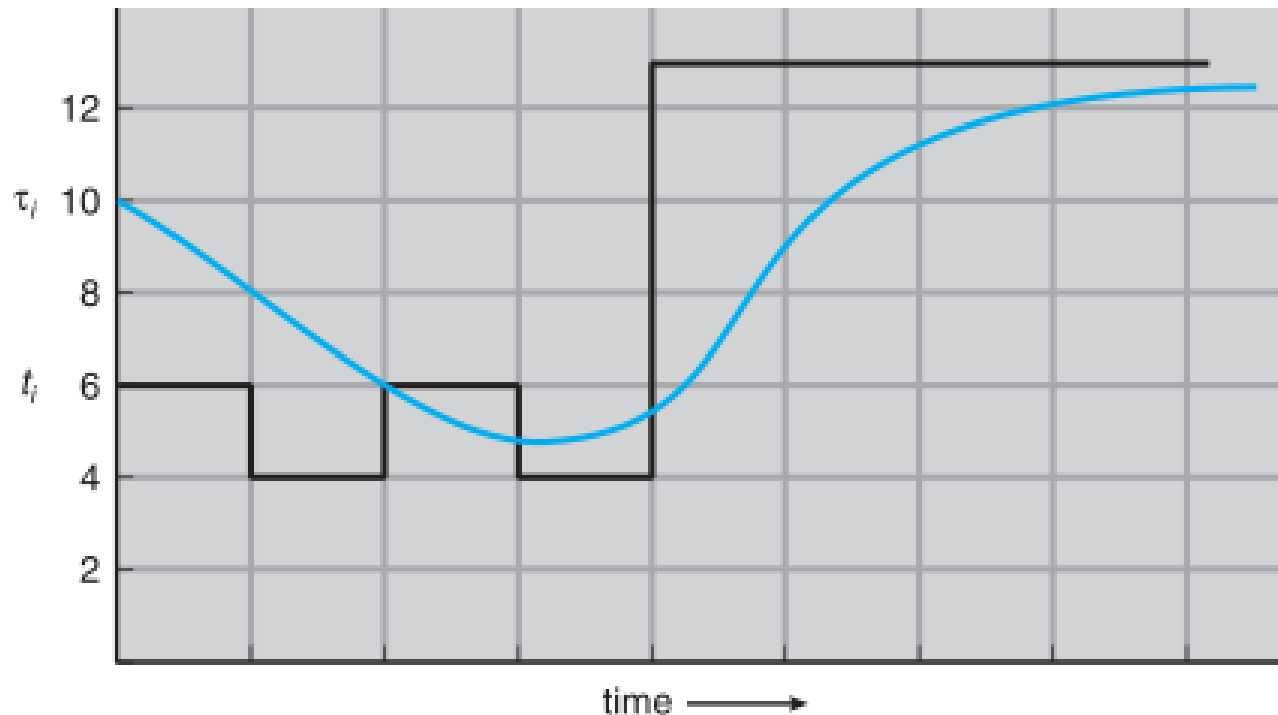
- We can only *estimate* the length.
- This can be done by using the length of previous CPU bursts, using exponential averaging:

$t_n$  = actual length of the  $n^{th}$  CPU burst

$$\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$$

$$0 \leq \alpha \leq 1$$

# Prediction of the Length of the Next CPU-Burst



CPU burst ( $t_i$ )	6	4	6	4	13	13	13	...	
"guess" ( $\tau_i$ )	10	8	6	6	5	9	11	12	...

# Priority Scheduling

- A priority number (integer) is associated with each process.
- The CPU is allocated to the process with the highest priority (smallest integer  $\equiv$  highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU-burst time.
- Problem: **Starvation** – low priority processes may never execute.
- Solution: **Aging** – as time progresses increase the priority of the process.