

# Review 2



**Operating Systems**  
**Wenbo Shen**

# What have we studied so far

---

- Computer architecture
- OS overview
- OS structures
- Processes
- IPC
- Thread
- Scheduling
- Synchronization
- Deadlock

# Revisit - Process Concept

---

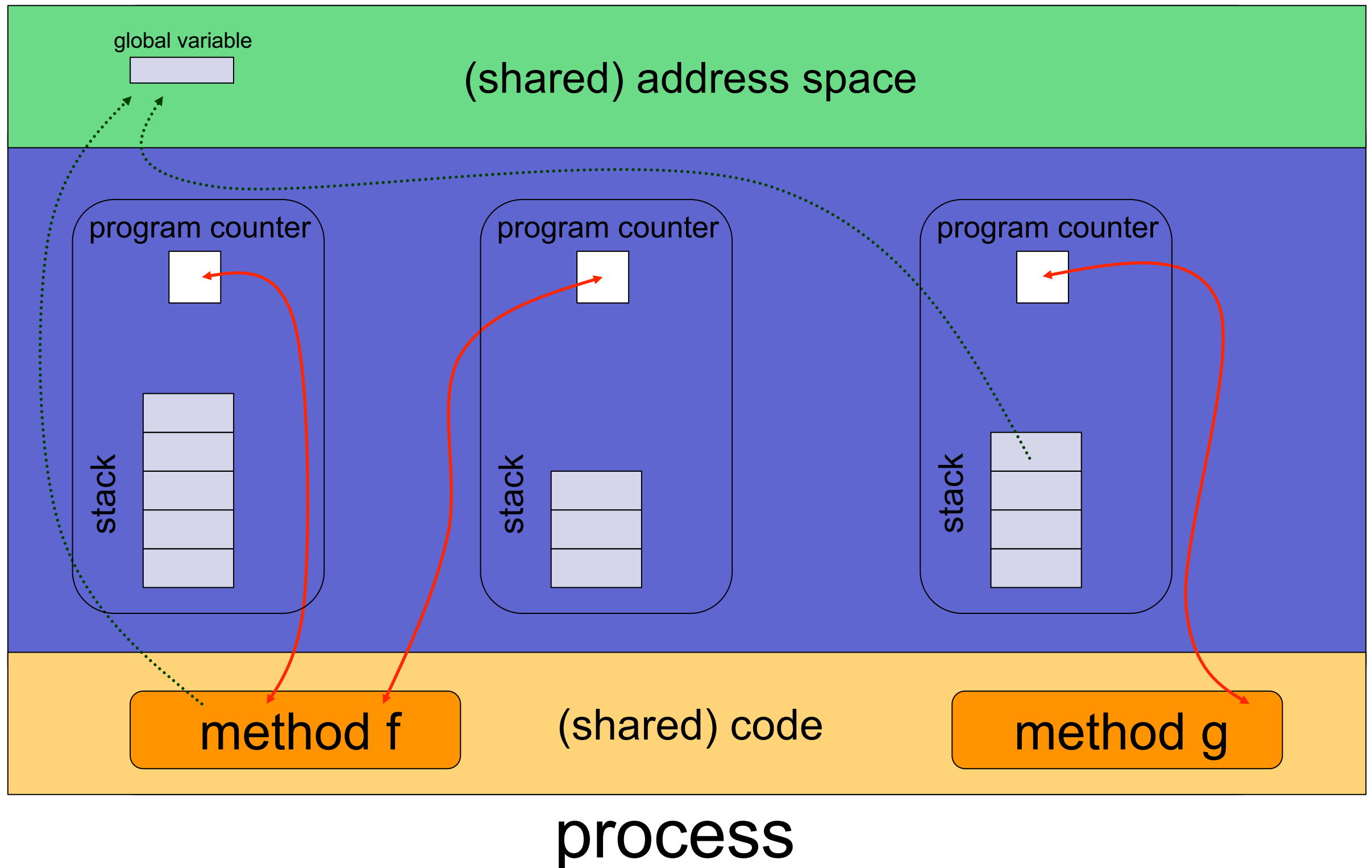
- Process =
  - **code** (also called the **text**)
    - initially stored on disk in an executable file
  - **program counter**
    - points to the next instruction to execute (i.e., an address in the code)
  - content of the processor's **registers**
  - a runtime **stack**
  - a **data section**
    - global variables (.bss and .data in x86 assembly)
  - a **heap**
    - for dynamically allocated memory (malloc, new, etc.)

# Thread Definition

---

- A thread is a basic unit of CPU utilization within a process
- Each thread has its own
  - thread ID
  - program counter
  - register set
  - Stack
- It shares the following with other threads within the same process
  - code section
  - data section
  - the heap (dynamically allocated memory)
  - open files and signals
- **Concurrency:** A multi-threaded process can do multiple things at once

# A More Detailed Figure



# Advantages of Threads?

---

- Economy:
  - Creating a thread is cheap
    - Much cheaper than creating a process
  - Context-switching between threads is cheap
    - Much cheaper than between processes
- Resource Sharing:
  - Threads naturally share memory
    - With processes you have to use possibly complicated IPC (e.g., Shared Memory Segments)
  - Having concurrent activities in the same address space is very powerful
    - But fraught with danger

# Advantages of Threads?

---

- Responsiveness
  - A program that has concurrent activities is more responsive
    - While one thread blocks waiting for some event, another can do something
    - e.g. Spawn a thread to answer a client request in a client-server implementation
  - This is true of processes as well, but with threads we have better sharing and economy
- Scalability
  - Running multiple “threads” at once uses the machine more effectively
    - e.g., on a multi-core machine
  - This is true of processes as well, but with threads we have better sharing and economy

# Drawbacks of Threads

---

- Threads may be more memory-constrained than processes
  - Due to OS limitation of the address space size of a single process
  - Not a problem any more on 64-bit architecture
- Threads do not benefit from memory protection
  - Concurrent programming with Threads is hard
    - But so is it with Processes and Shared Memory Segments



# User Threads vs. Kernel Threads

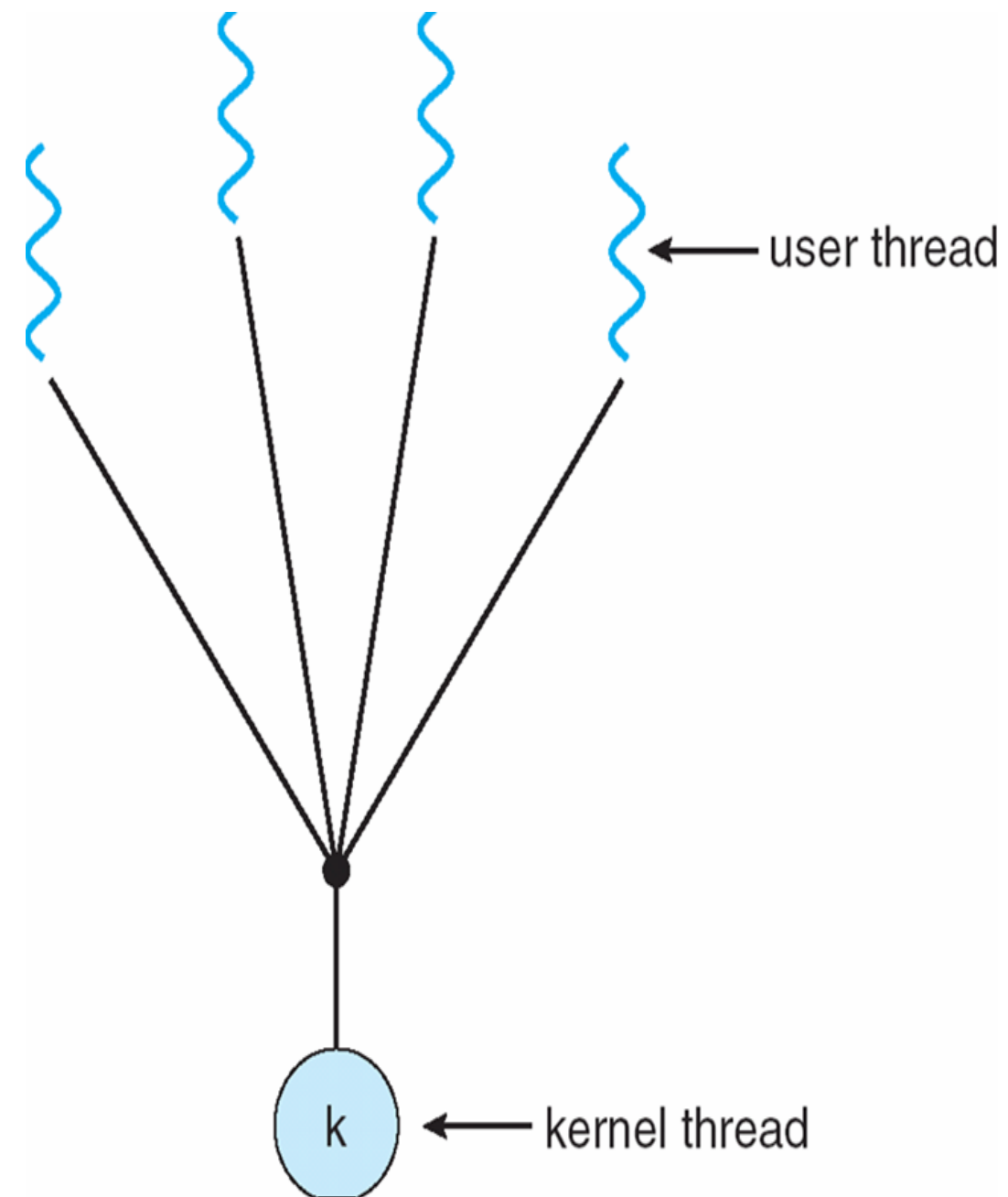
---

- Threads can be supported solely in User Space
  - Threads are managed by some user-level thread library (e.g., Java Green Threads)
- Threads can also be supported in Kernel Space
  - The kernel has data structure and functionality to deal with threads
  - Most modern OSes support kernel threads
    - ▶ In fact, Linux doesn't really make a difference between processes and threads (same data structure)

# Many-to-One Model

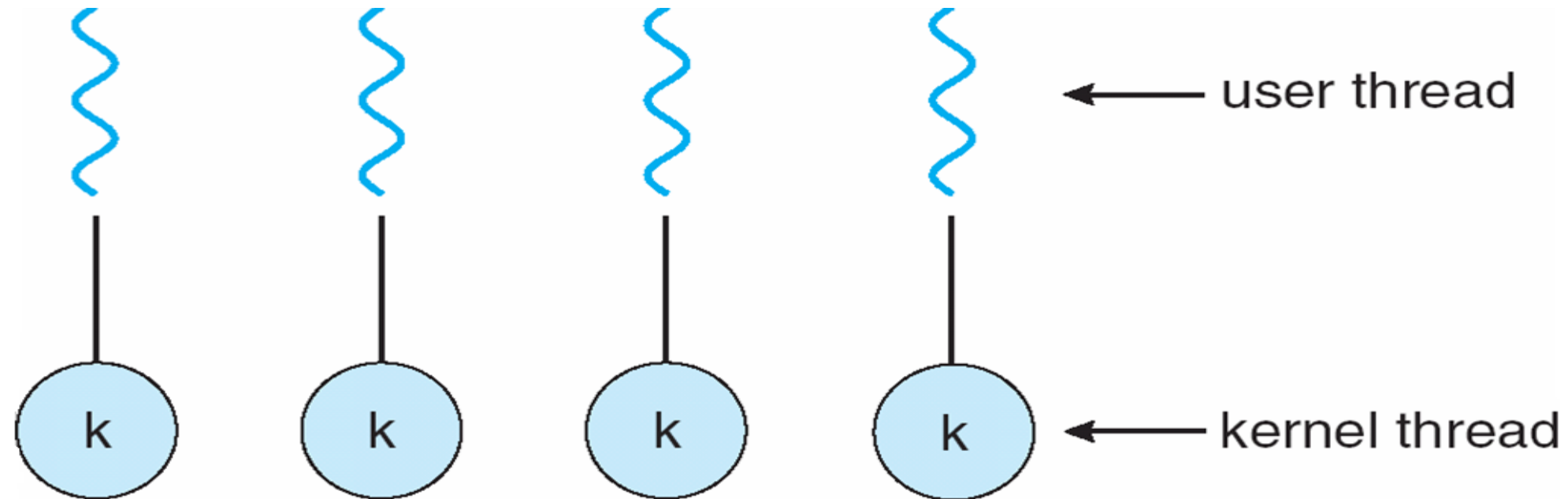
---

- Advantage: multi-threading is efficient and low-overhead
  - No syscalls to the kernel
- **Major Drawback #1:** cannot take advantage of a multi-core architecture!
- **Major Drawback #2:** if one threads blocks, then all the others do!
- Examples (User-level Threads):
  - Java Green Threads
  - GNU Portable Threads



# One-to-One Model

---

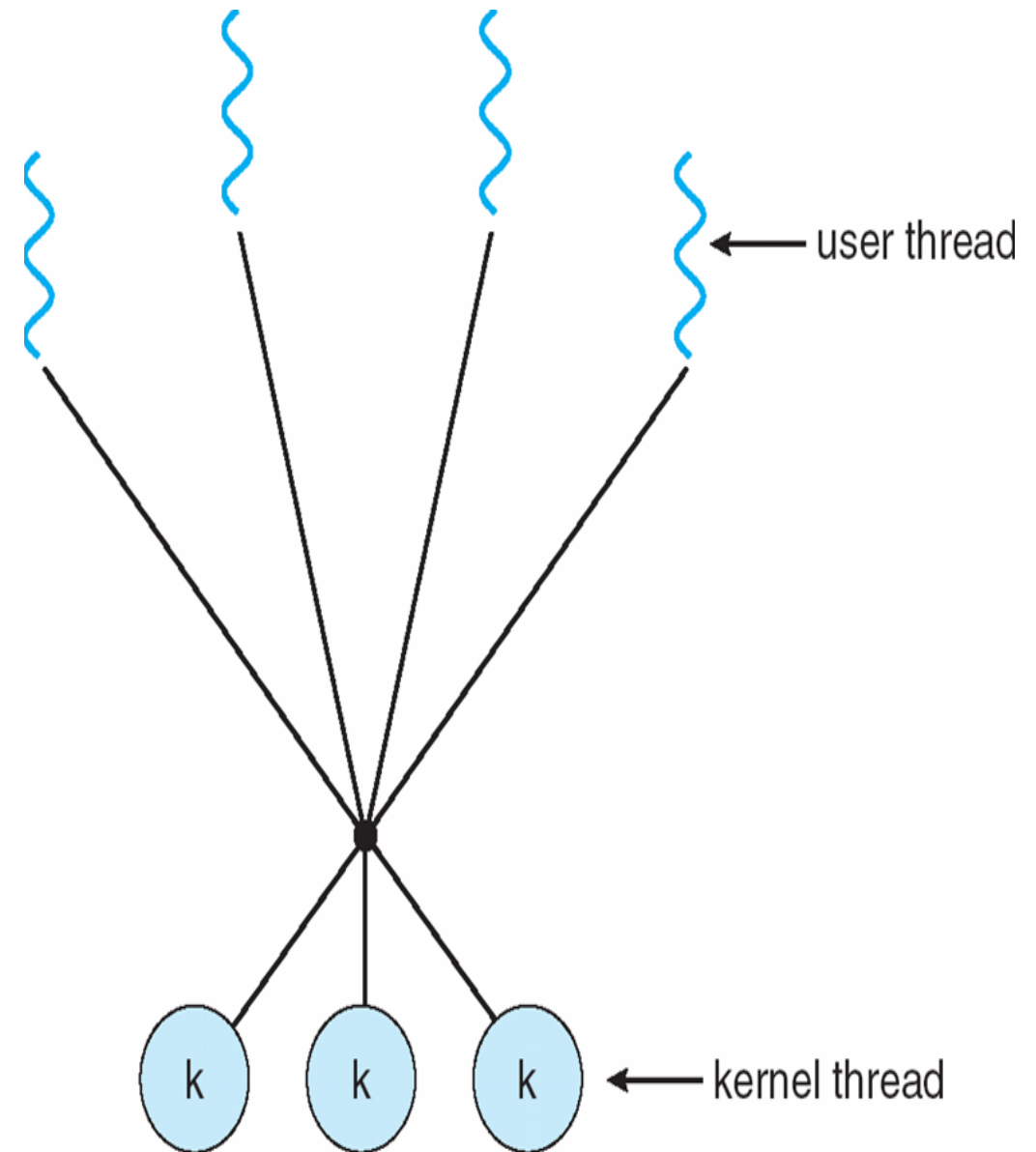


- Removes both drawbacks of the Many-to-One Model
- Creating a new threads requires work by the kernel
  - Not as fast as in the Many-to-One Model
- Example:
  - Linux
  - Windows
  - Solaris 9 and later

# Many-to-Many Model

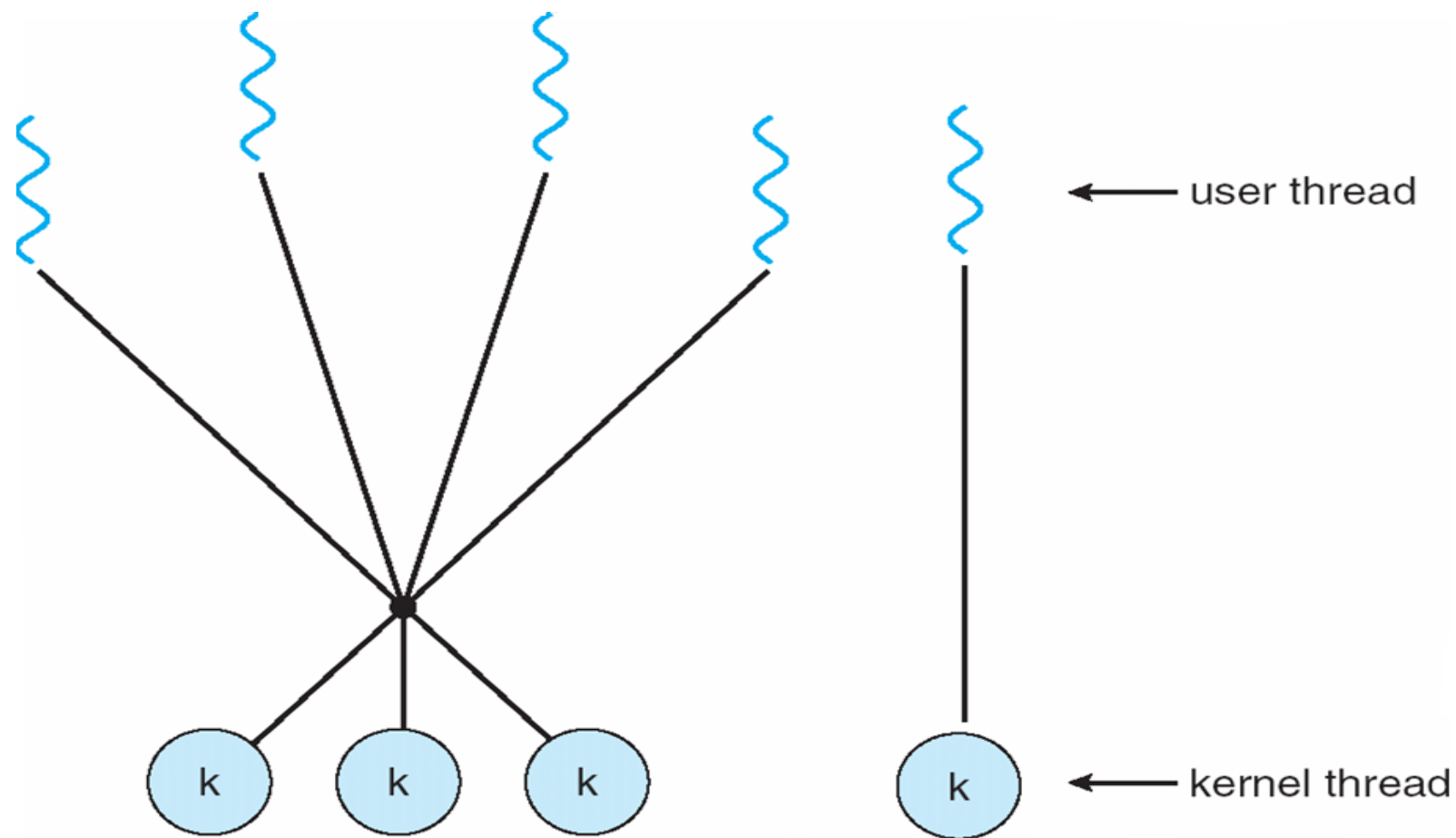
---

- A compromise
- If a user thread blocks, the kernel can create a new kernel threads to avoid blocking all user threads
- A new user thread doesn't necessarily require the creation of a new kernel thread
- True concurrency can be achieved on a multi-core machine
- Examples:
  - Solaris 9 and earlier
  - Win NT/2000 with the ThreadFiber package



# Two-Level Model

---



- The user can say: “Bind this thread to its own kernel thread”
- Example:
  - IRIX, HP-UX, Tru64 UNIX
  - Solaris 8 and earlier

# Linux Threads

---

- In Linux, a thread is also called a light-weight process (LWP)
- The clone() syscall is used to create a task
  - Shares execution context with its parent
  - pthread library uses clone() to implement threads. Refer to `./nptl/sysdeps/pthread/createthread.c`

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

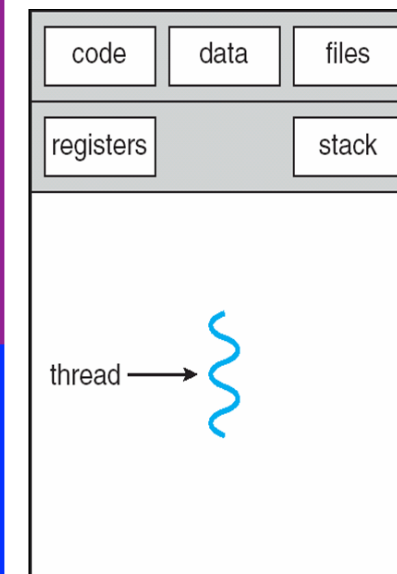


# Linux Threads

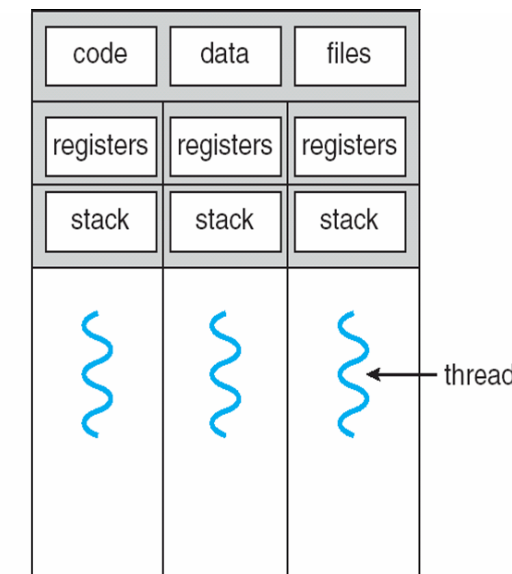
- Single-threaded process vs multi-threaded process

```
wenbo@wenbo-desktop:~/KERNEL/linux.git$ ps -eLf
```

UID	PID	PPID	LWP	C	NLWP	STIME	TTY	TIME	CMD
root	1	0	1	0	1	3月11 ?		00:00:19	/sbin/init splash
root	2	0	2	0	1	3月11 ?		00:00:00	[kthreadd]
root	4	2	4	0	1	3月11 ?		00:00:00	[kworker/0:0H]
root	6	2	6	0	1	3月11 ?		00:00:00	[mm_percpu_wq]
root	7	2	7	0	1	3月11 ?		00:00:00	[ksoftirqd/0]
root	8	2	8	0	1	3月11 ?		00:00:31	[rcu_sched]
root	9	2	9	0	1	3月11 ?		00:00:00	[rcu_bh]
root	10	2	10	0	1	3月11 ?		00:00:00	[migration/0]
root	11	2	11	0	1	3月11 ?		00:00:00	[watchdog/0]
root	704	1	704	0	1	3月11 ?		00:00:00	/usr/sbin/cron -f
root	718	1	718	0	16	3月11 ?		00:00:00	/usr/lib/snapd/snapd
root	718	1	882	0	16	3月11 ?		00:00:00	/usr/lib/snapd/snapd
root	718	1	883	0	16	3月11 ?		00:00:00	/usr/lib/snapd/snapd
root	718	1	884	0	16	3月11 ?		00:00:00	/usr/lib/snapd/snapd
root	718	1	885	0	16	3月11 ?		00:00:00	/usr/lib/snapd/snapd
root	718	1	917	0	16	3月11 ?		00:00:00	/usr/lib/snapd/snapd
root	718	1	921	0	16	3月11 ?		00:00:01	/usr/lib/snapd/snapd
root	718	1	922	0	16	3月11 ?		00:00:00	/usr/lib/snapd/snapd
root	718	1	923	0	16	3月11 ?		00:00:01	/usr/lib/snapd/snapd
root	718	1	924	0	16	3月11 ?		00:00:01	/usr/lib/snapd/snapd



single-threaded process



multithreaded process

# Linux Threads

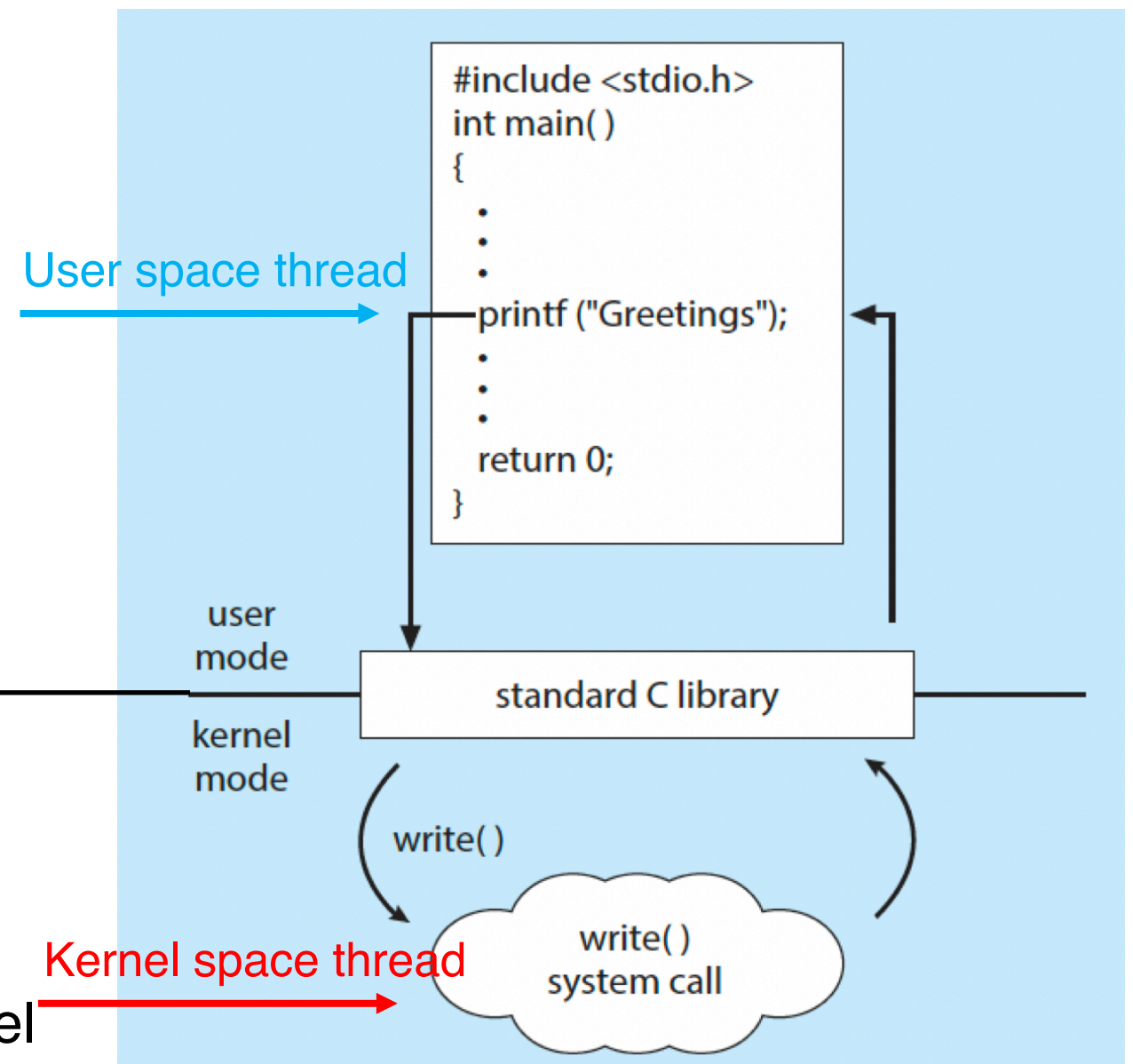
---

- Linux does not distinguish between processes and threads: they're called **tasks**
  - Kernel data structure: task\_struct
- A process is
  - either a **single** thread + an address space
    - PID is thread ID
  - or **multiple** threads + an address space
    - PID is the leading thread ID



# User thread to kernel thread mapping

- One task
  - One task struct – PCB
  - Execute in one thread
  - Can be executed in user space
    - User code, user space stack
- Can be executed in kernel space
  - Such as calls a system call
  - Execution flow traps to kernel
  - Execute kernel code, use kernel space stack



# What have we studied so far

---

- Computer architecture
- OS overview
- OS structures
- Processes
- IPC
- Thread
- **Scheduling**
- Synchronization
- Deadlock

# CPU Scheduling

---

- **Definition:** the decisions made by the OS to figure out which ready processes/threads should run and for how long
  - Necessary in multi-programming environments
- CPU Scheduling is important for system performance and productivity
  - Maximizes CPU utilization so that it's never idle
- The **policy** is the scheduling strategy
- The **mechanism** is the **dispatcher**
  - A component of the OS that's used to switch between processes
    - That in turn uses the context switch mechanism
  - Must be lightning fast for time-sharing (dispatcher latency)

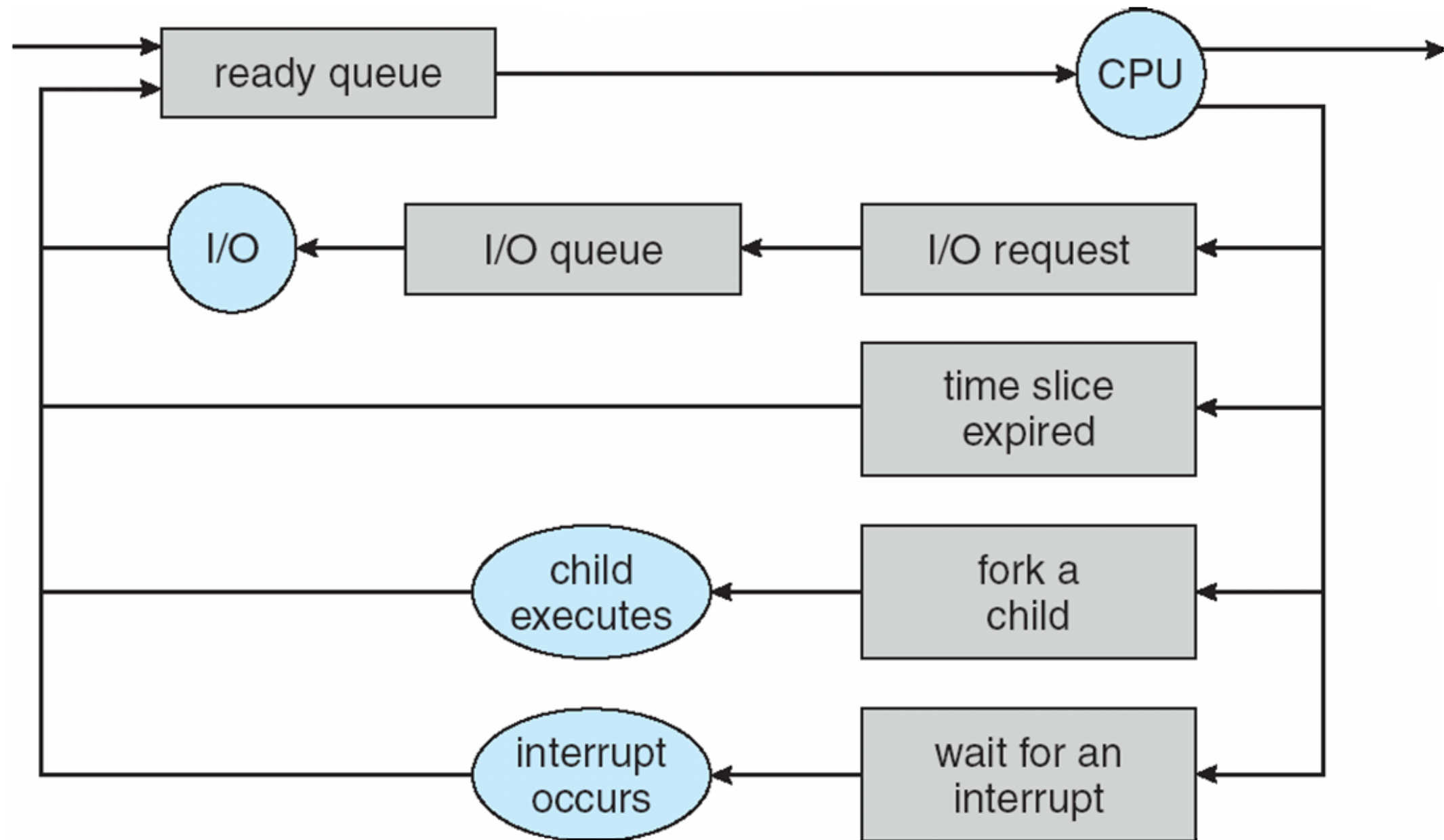
# Scheduling Objectives

---

- Finding the right objective is an open question
- There are many conflicting goals that one could attempt to achieve
  - Maximize **CPU Utilization**
    - Fraction of the time the CPU isn't idle
  - Maximize **Throughput**
    - Amount of “useful work” done per time unit
  - Minimize **Turnaround Time**
    - Time from process creation to process completion
  - Minimize **Waiting Time**
    - Amount of time a process spends in the READY state
  - Minimize **Response Time**
    - Time from process creation until the “first response” is received
- Question: should we optimize averages, maxima, variances?
  - Again, a lot of theory here...

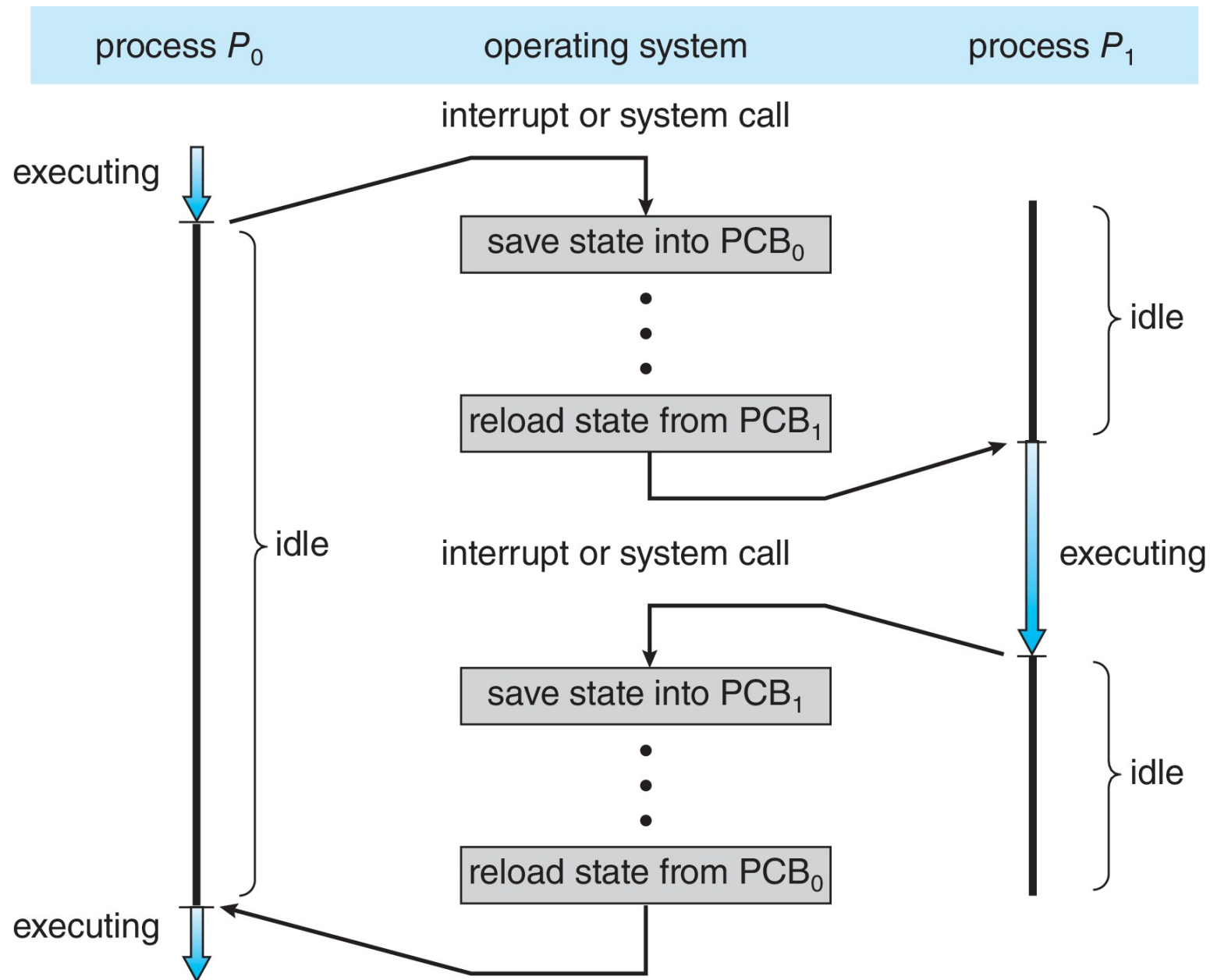
# Scheduling and Queues

---



# CPU Switch From Process to Process

A context switch occurs when the CPU switches from one process to another.



# Context Switch

---

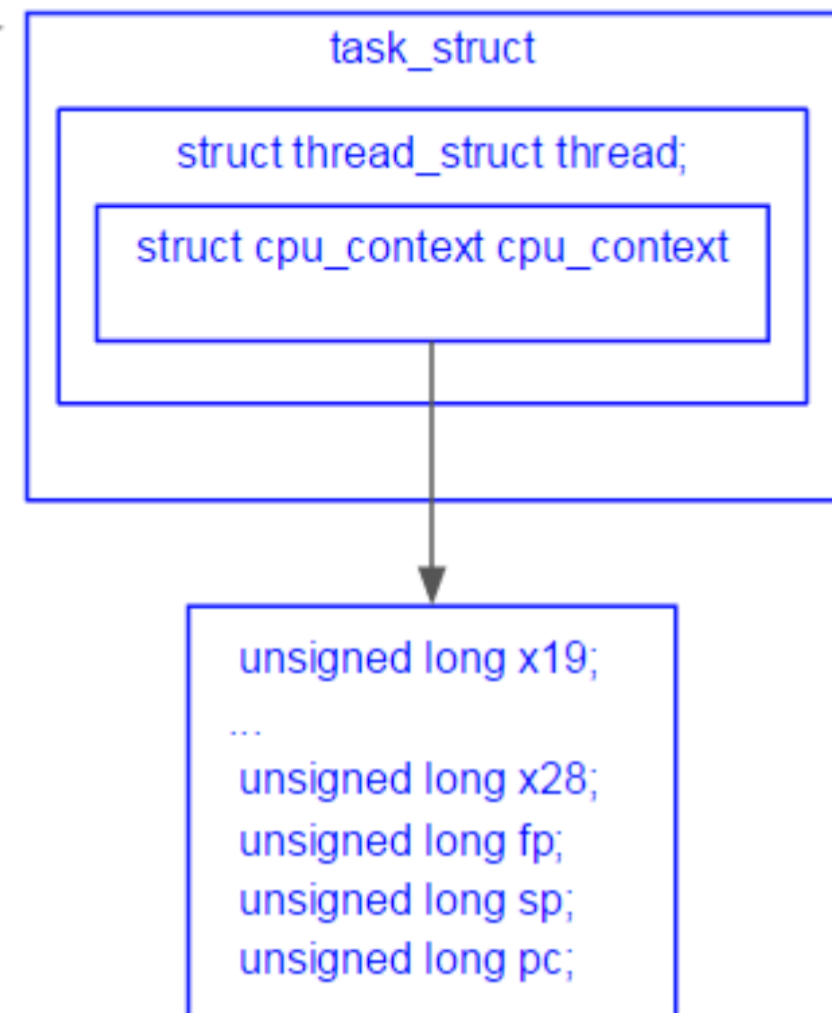
- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
  - The more complex the OS and the PCB → the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU → multiple contexts saved/loaded at once



# Context Switch

```
extern struct task_struct *cpu_switch_to(struct task_struct *prev,  
                                         struct task_struct *next);
```

```
ENTRY(cpu_switch_to)  
    mov     x10, #THREAD_CPU_CONTEXT  
    add     x8, x0, x10  
    mov     x9, sp  
    stp     x19, x20, [x8], #16  
    stp     x21, x22, [x8], #16  
    stp     x23, x24, [x8], #16  
    stp     x25, x26, [x8], #16  
    stp     x27, x28, [x8], #16  
    stp     x29, x9, [x8], #16  
    str     lr, [x8]  
    add     x8, x1, x10  
    ldp     x19, x20, [x8], #16  
    ldp     x21, x22, [x8], #16  
    ldp     x23, x24, [x8], #16  
    ldp     x25, x26, [x8], #16  
    ldp     x27, x28, [x8], #16  
    ldp     x29, x9, [x8], #16  
    ldr     lr, [x8]  
    mov     sp, x9  
    msr     sp_el0, x1  
    ret  
ENDPROC(cpu_switch_to)
```





# Context Switch

```
extern struct task_struct *cpu_switch_to(struct task_struct *prev,  
                                         struct task_struct *next);
```

```
ENTRY(cpu_switch_to)
    mov     x10, #THREAD_CPU_CONTEXT
    add     x8, x0, x10
    mov     x9, sp
    stp     x19, x20, [x8], #16
    stp     x21, x22, [x8], #16
    stp     x23, x24, [x8], #16
    stp     x25, x26, [x8], #16
    stp     x27, x28, [x8], #16
    stp     x29, x9, [x8], #16
    str     lr, [x8]
    add     x8, x1, x10
    ldp     x19, x20, [x8], #16
    ldp     x21, x22, [x8], #16
    ldp     x23, x24, [x8], #16
    ldp     x25, x26, [x8], #16
    ldp     x27, x28, [x8], #16
    ldp     x29, x9, [x8], #16
    ldr     lr, [x8]
    mov     sp, x9
    msr     sp_el0, x1
    ret
ENDPROC(cpu_switch_to)
```

Most important step is  
**STACK SWITCHING**

- 1) Move sp to general reg
- 2) Save old sp
- 3) Load new sp
- 4) Move new sp to sp reg

# Scheduling Algorithms

---

- First-Come, First-Served Scheduling
- Shortest-Job-First Scheduling
- Round-Robin Scheduling
- Priority Scheduling
- Multilevel Queue Scheduling
- Multilevel Feedback Queue Scheduling

Average Waiting Time  
Average Turnaround Time

# What have we studied so far

---

- Computer architecture
- OS overview
- OS structures
- Processes
- IPC
- Thread
- Scheduling
- Synchronization
- Deadlock

# Race Condition

---

- Several processes (or threads) access and manipulate the same data **concurrently** and the outcome of the execution depends on the **particular order** in which the access takes place, is called a **race-condition**

# Critical Section

---

- Consider system of  $n$  processes  $\{p_0, p_1, \dots, p_{n-1}\}$
- Each process has a critical section segment of code
  - e.g., to change common variables, update table, write file, etc.
- Only one process can be in the critical section
  - when one process in critical section, no other may be in its **critical section**
  - each process must ask permission to enter critical section in **entry section**
  - the permission should be released in **exit section**
  - **Remainder section**

# Solution to Critical-Section: Three Requirements

---

- **Mutual Exclusion**
  - only one process can execute in the critical section
- **Progress**
  - if no process is executing in its critical section and some processes wish to enter their critical section, then only those processes that are not executing in their retainer sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely
- **Bounded waiting**
  - There exists a bound, or limit, 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
  - It prevents starvation

# Hardware Support for Synchronization

---

- Many systems provide hardware support for critical section code
- **Uniprocessors: disable interrupts**
  - currently running code would execute without preemption
  - generally too inefficient on multiprocessor systems
    - need to disable all the interrupts
    - operating systems using this not scalable
- Solutions:
  - 1. **Memory barriers**
  - 2. **Hardware instructions**
    - **test-and-set**: either test memory word and set value
    - **compare-and-swap**: compare and swap contents of two memory words
  - 3. **Atomic variables**

# Synchronization

---

- Spinlock
  - Mutex lock with busy waiting (spin)
- Semaphore
  - introduces waiting queue



# Synchronization

---

- Spinlock
  - Mutex lock with busy waiting (spin)
  - Good for short waiting (short critical section)
- Semaphore
  - introduces waiting queue
  - Good for long waiting (long critical section)
  - Wait and signal operations are protected by spinlock

# Implementation with waiting queue


---

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}

signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

# Semaphore w/ waiting queue


```
Semaphore mutex;    // initialized to 1
do {
    _____
    wait (mutex);
    critical section
    signal (mutex);
    _____
    remainder section
} while (TRUE);      //while loop but not busy waiting
```




A red vertical double-headed arrow is positioned between the `wait (mutex);` and `signal (mutex);` lines, indicating a period of busy waiting.

**busy waiting**

```
Semaphore mutex;    // initialized to 1
do {
    _____
    wait (mutex);
    _____
    critical section
    _____
    signal (mutex);
    _____
    remainder section
} while (TRUE);      //while loop but not busy waiting
```



A red vertical double-headed arrow is positioned between the `wait (mutex);` and `critical section` lines, indicating a period of busy waiting.



A red vertical double-headed arrow is positioned between the `signal (mutex);` and `remainder section` lines, indicating a period of busy waiting.

**busy waiting**

**busy waiting**

# Priority Inversion

---

- **Priority Inversion:** a higher priority process is **indirectly** preempted by a lower priority task
  - e.g., three processes,  $P_L$ ,  $P_M$ , and  $P_H$  with priority  $P_L < P_M < P_H$
  - $P_L$  holds a lock that was requested by  $P_H \Rightarrow P_H$  is blocked
  - $P_M$  becomes ready and preempted the  $P_L$
  - It effectively "inverts" the relative priorities of  $P_M$  and  $P_H$
- Solution: **priority inheritance**
  - temporary assign the highest priority of waiting process ( $P_H$ ) to the process holding the lock ( $P_L$ )

# Classical Synchronization Problems

---

- Bounded-buffer problem
- Readers-writers problem
- Dining-philosophers problem

# Bounded-Buffer Problem

---

- Two processes, the producer and the consumer share **n** buffers
  - the producer generates data, puts it into the buffer
  - the consumer consumes data by removing it from the buffer
- The problem is to make sure:
  - **the producer won't try to add data into the buffer if it is full**
  - **the consumer won't try to remove data from an empty buffer**
  - also call producer-consumer problem
- Solution:
  - n buffers, each can hold one item
  - semaphore **mutex** initialized to the value **1**
  - semaphore **full-slots** initialized to the value **0**
  - semaphore **empty-slots** initialized to the value **N**

# Bounded-Buffer Problem

---

- The producer process:

```
do {  
  //produce an item  
  
  ...  
  
  wait(empty-slots);  
  
  wait(mutex);  
  
  //add the item to the buffer  
  
  ...  
  
  signal(mutex);  
  
  signal(full-slots);  
} while (TRUE)
```

# Bounded Buffer Problem

---

- The consumer process:

```
do {  
    wait(full-slots);  
    wait(mutex);  
    //remove an item from  buffer  
  
    ...  
    signal(mutex);  
    signal(empty-slots);  
    //consume the item  
  
    ...  
} while (TRUE);
```



# Bounded Buffer Problem

---

- The consumer process:

```
do {  
    wait(full-slots);  
    wait(mutex);  
    //remove an item from  buffer  
  
    ...  
    signal(mutex);  
    signal(empty-slots);  
    //consume the item  
  
    ...  
} while (TRUE);
```

# Dining-Philosophers Problem

---

- Philosophers spend their lives thinking and eating
  - they sit in a round table, but don't interact with each other
- They occasionally try to pick up 2 chopsticks (one at a time) to eat
  - one chopstick between each adjacent two philosophers
  - need both chopsticks to eat, then release both when done
  - Dining-philosopher problem represents **multi-resource synchronization**
- Solution (assuming **5 philosophers**):
  - semaphore **chopstick[5]** initialized to 1

# An asymmetrical solution

- only odd philosophers start left-hand first, and even philosophers start right-hand first. This does not deadlock.

```
void pickup(Phil_struct *ps)
{
    Sticks *pp;
    int phil_count;

    pp = (Sticks *) ps->v;
    phil_count = pp->phil_count;

    if (ps->id % 2 == 1) {
        pthread_mutex_lock(pp->lock[ps->id]);          /* lock up left stick */
        pthread_mutex_lock(pp->lock[(ps->id+1)%phil_count]); /* lock right stick */
    } else {
        pthread_mutex_lock(pp->lock[(ps->id+1)%phil_count]); /* lock right stick */
        pthread_mutex_lock(pp->lock[ps->id]);          /* lock up left stick */
    }
}

void putdown(Phil_struct *ps)
{
    Sticks *pp;
    int i;
    int phil_count;

    pp = (Sticks *) ps->v;
    phil_count = pp->phil_count;

    if (ps->id % 2 == 1) {
        pthread_mutex_unlock(pp->lock[(ps->id+1)%phil_count]); /* unlock right stick */
        pthread_mutex_unlock(pp->lock[ps->id]); /* unlock left stick */
    } else {
        pthread_mutex_unlock(pp->lock[ps->id]); /* unlock left stick */
        pthread_mutex_unlock(pp->lock[(ps->id+1)%phil_count]); /* unlock right stick */
    }
}
```

# Linux Synchronization

---

- Linux:
  - prior to version 2.6, disables interrupts to implement short critical sections
  - version 2.6 and later, fully preemptive
- Linux provides:
  - **atomic integers**
  - **spinlocks**
  - **semaphores**
    - on single-cpu system, spinlocks replaced by enabling/disabling kernel preemption
  - **reader-writer locks**

# What have we studied so far

---

- Computer architecture
- OS overview
- OS structures
- Processes
- IPC
- Thread
- Scheduling
- Synchronization
- **Deadlock**

# The Deadlock Problem

---

- **Deadlock:** a set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set
- Examples:
  - a system has 2 disk drives,  $P_1$  and  $P_2$  each hold one disk drive and each needs another one
  - semaphores A and B, initialized to 1

$P_1$	$P_2$
wait (A);	wait(B)
wait (B);	wait(A)

# Four Conditions of Deadlock

---

- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it, after it has completed its task
- **Circular wait:** there exists a set of waiting processes  $\{P_0, P_1, \dots, P_n\}$ 
  - $P_0$  is waiting for a resource that is held by  $P_1$
  - $P_1$  is waiting for a resource that is held by  $P_2 \dots$
  - $P_{n-1}$  is waiting for a resource that is held by  $P_n$
  - $P_n$  is waiting for a resource that is held by  $P_0$

# Handling deadlocks

---

- Deadlock prevention
- Deadlock avoidance
- Deadlock detection
- Deadlock recovery



# Resource-Allocation Graph

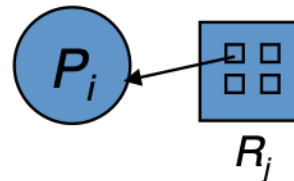
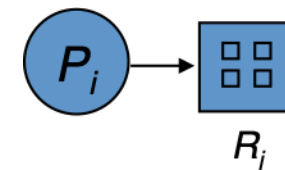
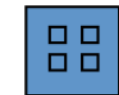
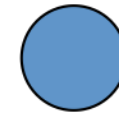
---

- Two types of nodes:
  - $P = \{P_1, P_2, \dots, P_n\}$ , the set of all the **processes** in the system
  - $R = \{R_1, R_2, \dots, R_m\}$ , the set of all **resource** types in the system
- Two types of edges:
  - **request edge**: directed edge  $P_i \rightarrow R_j$
  - **assignment edge**: directed edge  $R_j \rightarrow P_i$

# Resource-Allocation Graph

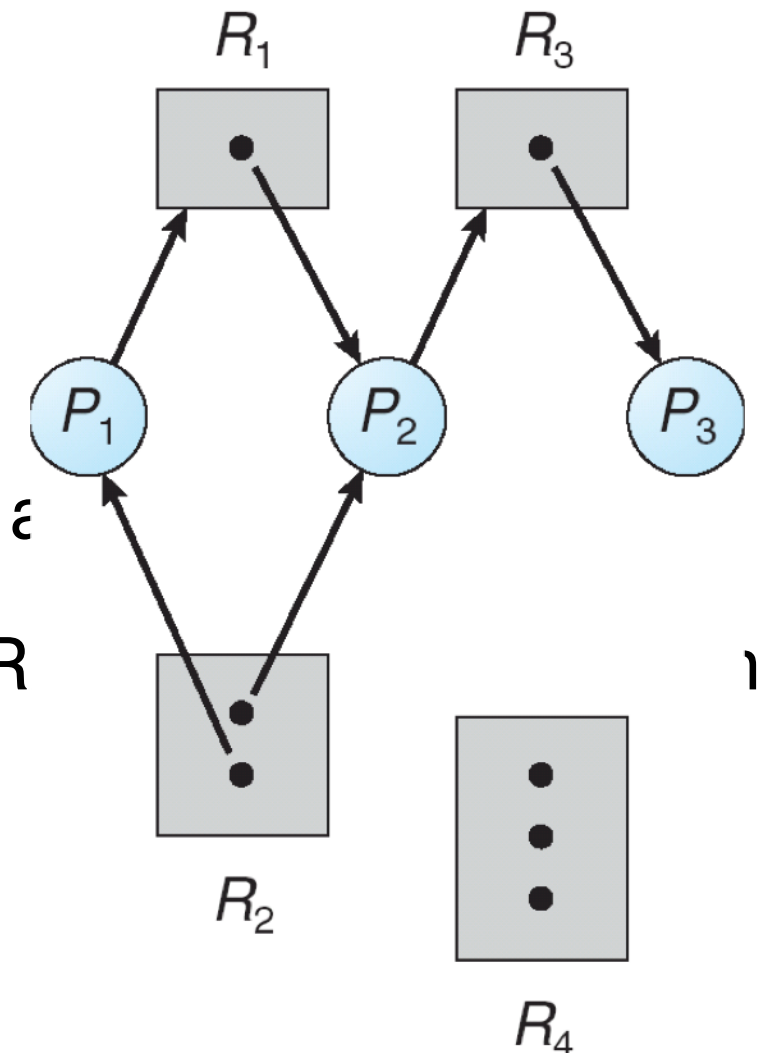
---

- Process
- Resource Type with 4 instances
- $P_i$  requests instance of  $R_j$
- $P_i$  is holding an instance of  $R_j$



# Resource Allocation Graph

- One instance of  $R_1$
- Two instances of  $R_2$
- One instance of  $R_3$
- Three instance of  $R_4$
- $P_1$  holds one instance of  $R_2$  and is waiting for  $\epsilon$
- $P_2$  holds one instance of  $R_1$ , one instance of  $R_2$  and one instance of  $R_3$
- $P_3$  is holds one instance of  $R_3$



# Basic Facts

---

- If graph contains **no cycles**  $\Rightarrow$  **no deadlock**
- If graph contains a cycle
  - if only **one instance per resource type**,  $\Rightarrow$  **deadlock**
  - if **several instances** per resource type  $\Rightarrow$  **possibility** of deadlock

# Deadlock Prevention

---

- How to prevent **mutual exclusion**
  - not required for sharable resources
  - must hold for non-sharable resources
- How to prevent **hold and wait**
  - whenever a process requests a resource, it doesn't hold any other resources
    - require process to request ***all*** its resources before it begins execution
    - allow process to request resources only when the process has none
  - low resource utilization; starvation possible

# Deadlock Prevention

---

- How to handle **no preemption**
  - if a process requests a resource not available
    - release all resources currently being held
    - preempted resources are added to the list of resources it waits for
    - process will be restarted only when it can get all waiting resources
- How to handle **circular wait**
  - impose a total ordering of all resource types
  - require that each process requests resources in an increasing order
  - Many operating systems adopt this strategy for some locks.

# Deadlock Avoidance

---

- Dead avoidance: require **extra information** about how resources are to be requested
  - **Is this requirement practical?**
- Each process declares a **max** number of resources it may need
- Deadlock-avoidance algorithm ensure there can **never** be a **circular-wait** condition
- Resource-allocation state:
  - the number of **available** and **allocated** resources
  - the **maximum demands** of the processes
- Single instance deadlock avoidance
- Banker Algorithm

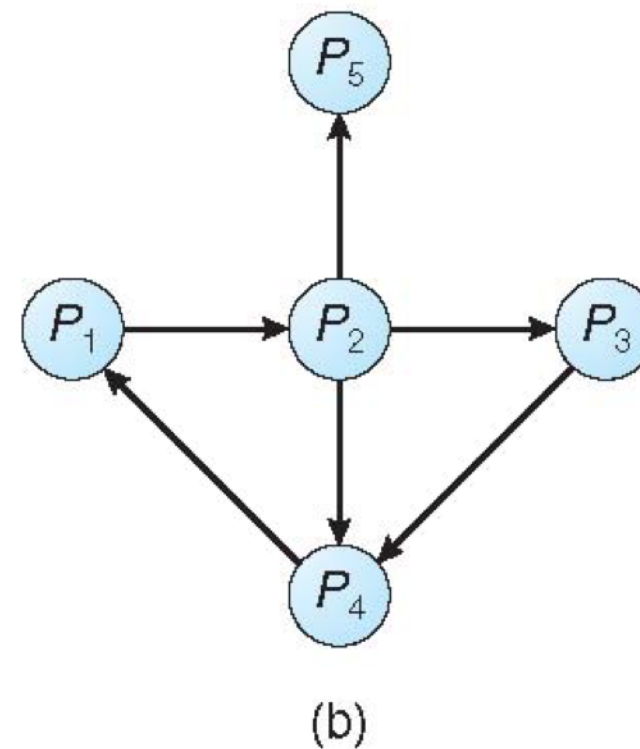
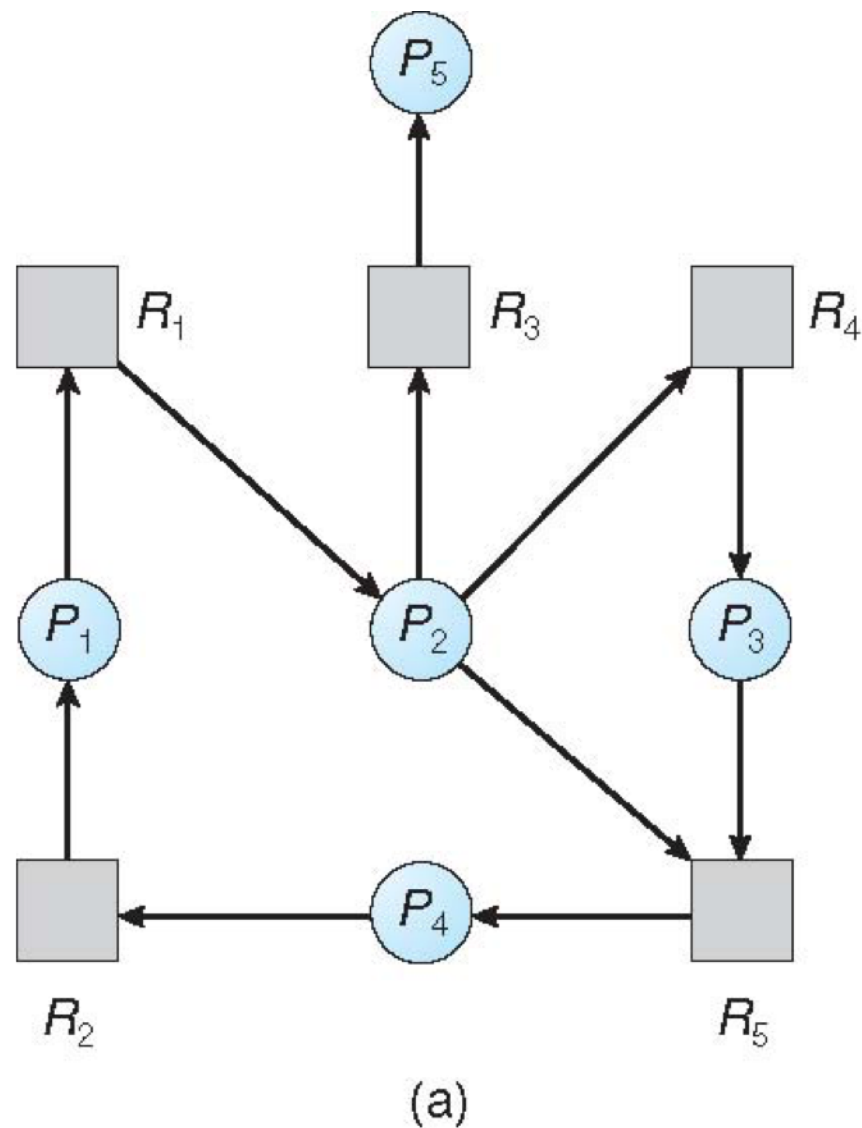
# Deadlock Detection: Single Instance Resources

---

- Maintain a wait-for graph, nodes are processes
- $P_i \rightarrow P_j$  if  $P_i$  is waiting for  $P_j$
- Periodically invoke an algorithm that searches for a cycle in the graph
  - if there is a cycle, there exists a deadlock
  - an algorithm to detect a cycle in a graph requires an order of  $n^2$  operations,
    - where  $n$  is the number of vertices in the graph



# Wait-for Graph Example



Resource-allocation Graph

wait-for graph

# Deadlock Detection: Multi-instance Resources

---

- Detection algorithm similar to Banker's algorithm's safety condition
  - to prove it is **not possible** to enter a **safe state**
- Data structure
  - **available**: a vector of length  $m$ , number of available resources of each type
  - **allocation**: an  $n \times m$  matrix defines the number of resources of each type currently allocated to each process
  - **request**: an  $n \times m$  matrix indicates the current request of each process
    - request  $[i, j] = k$ : process  $P_i$  is requesting  $k$  more instances of resource  $R_j$
  - **work**: a vector of  $m$ , the allocatable instances of resources
  - **finish**: a vector of  $m$ , whether the process has finished
    - if  $\text{allocation}[i] \neq 0 \Rightarrow \text{finish}[i] = \text{false}$ ; otherwise,  $\text{finish}[i] = \text{true}$

# Deadlock Recovery: Option I

---

- Terminate deadlocked processes. options:
  - abort all deadlocked processes
  - abort one process at a time until the deadlock cycle is eliminated
  - In which order should we choose to abort?
    - priority of the process
    - how long process has computed, and how much longer to completion
    - resources the process has used
    - resources process needs to complete
    - how many processes will need to be terminated
    - is process interactive or batch?

# Deadlock Recovery: Option II

---

- Resource preemption
  - Select a victim
  - Rollback
  - Starvation
    - How could you ensure that the resources do not preempt from the same process?