



# CS 423

# Operating System Design

<https://cs423-uiuc.github.io>

Tianyin Xu

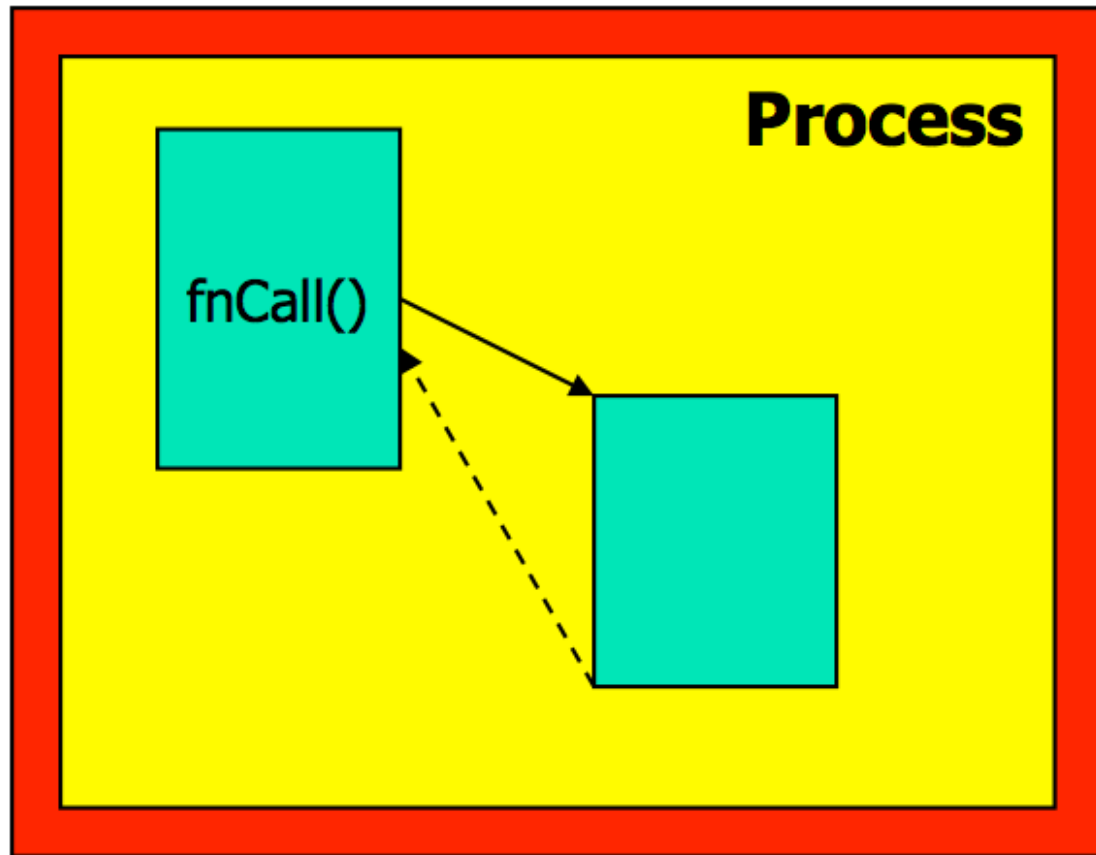
[tyxu@illinois.edu](mailto:tyxu@illinois.edu)

\* Thanks Adam Bates for the slides.

# System Calls



## Function Calls



Caller and callee are in the same Process

- Same user
- Same "domain of trust"

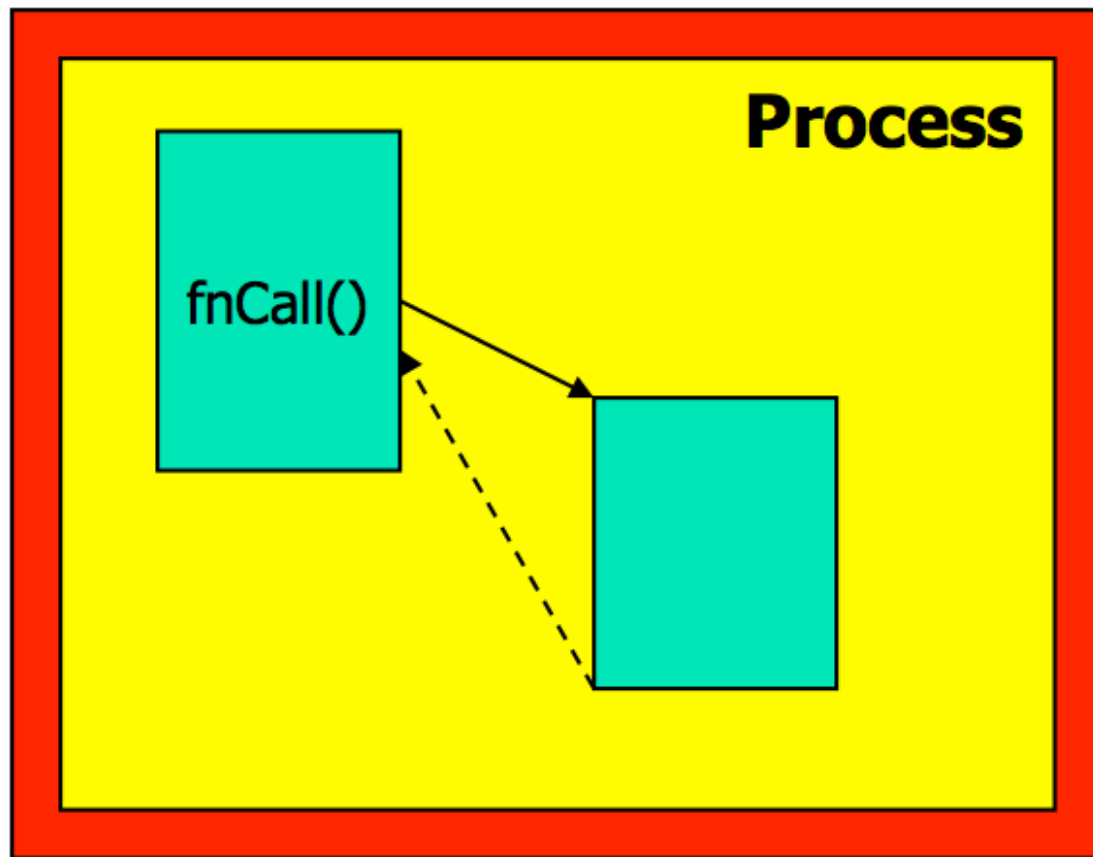
## System Calls



# Review: System Calls



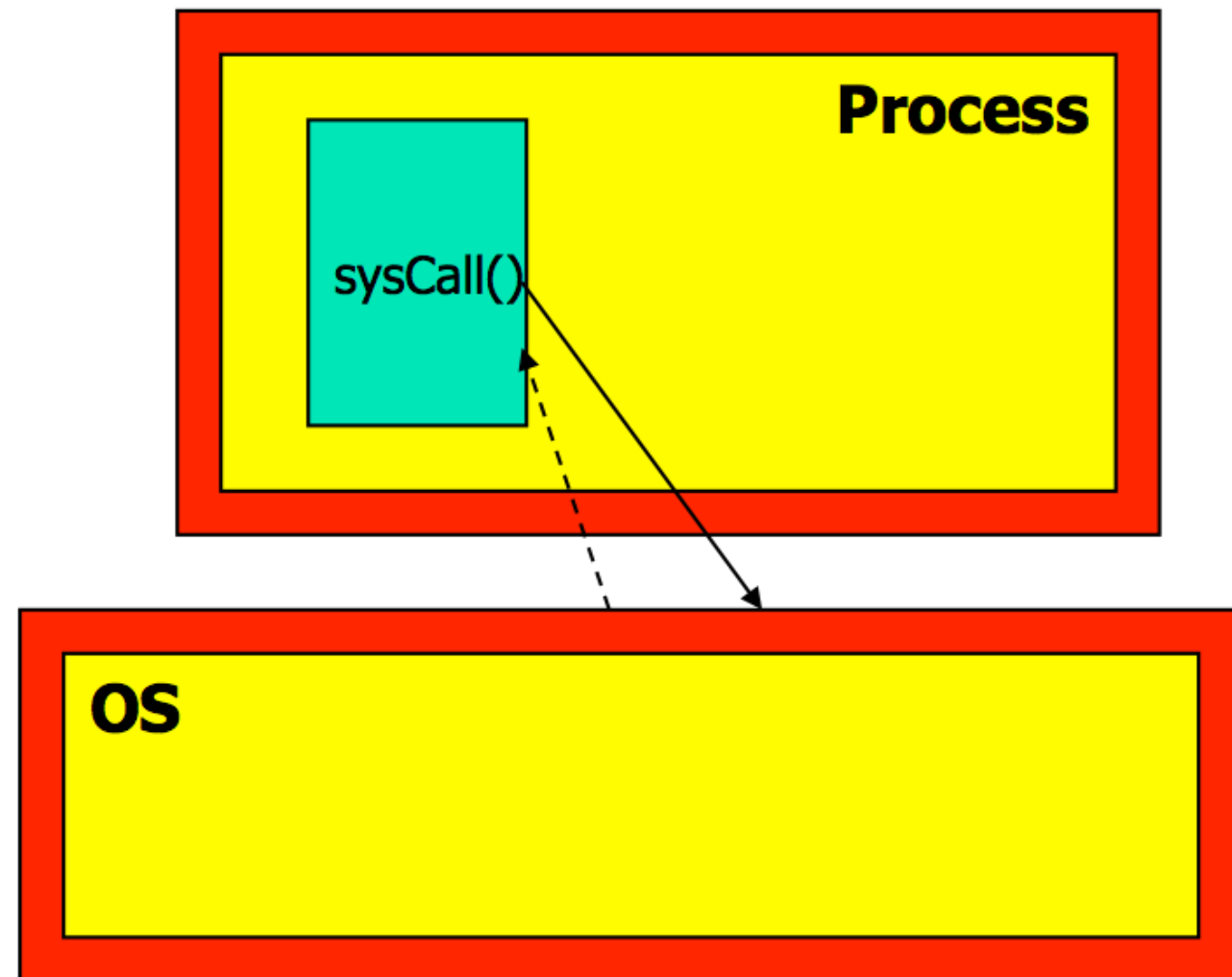
## Function Calls



Caller and callee are in the same Process

- Same user
- Same "domain of trust"

## System Calls



- OS is trusted; user is not.
- OS has super-privileges; user does not
- Must take measures to prevent abuse

# Example System Calls?



# Example System Calls?



## Example:

<code>getuid()</code>	<code>//get the user ID</code>
<code>fork()</code>	<code>//create a child process</code>
<code>exec()</code>	<code>//executing a program</code>

# Example System Calls?



## Example:

getuid() //get the user ID  
fork() //create a child process  
exec() //executing a program

Don't confuse system calls with stdlib calls

Differences?

Is printf() a system call?

Is rand() a system call?

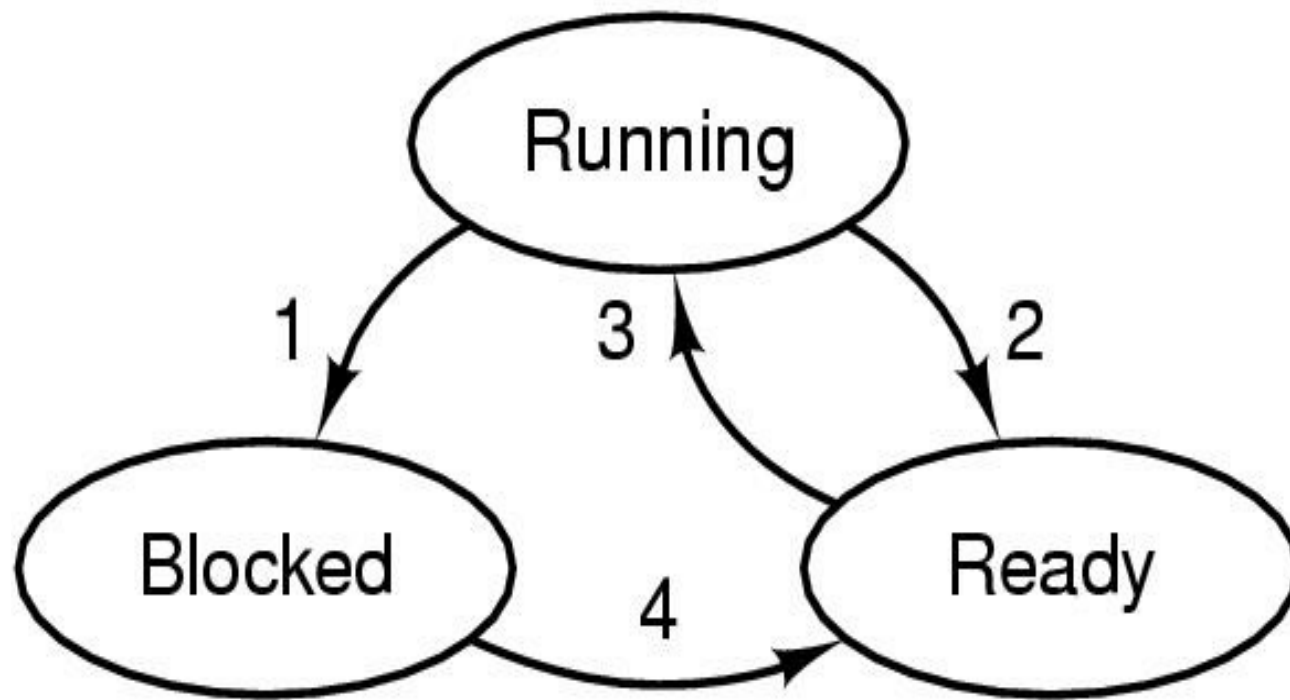
# Syscalls vs. I/O Lib Calls



**Each system call has analogous procedure calls from the standard I/O library:**

<u>System Call</u>	<u>Standard I/O call</u>
open	fopen
close	fclose
read/write	getchar/putchar getc/putc fgetc/fputc fread/fwrite
	gets/puts fgets/fputs scanf/printf fscanf/fprintf
lseek	fseek

# Processes

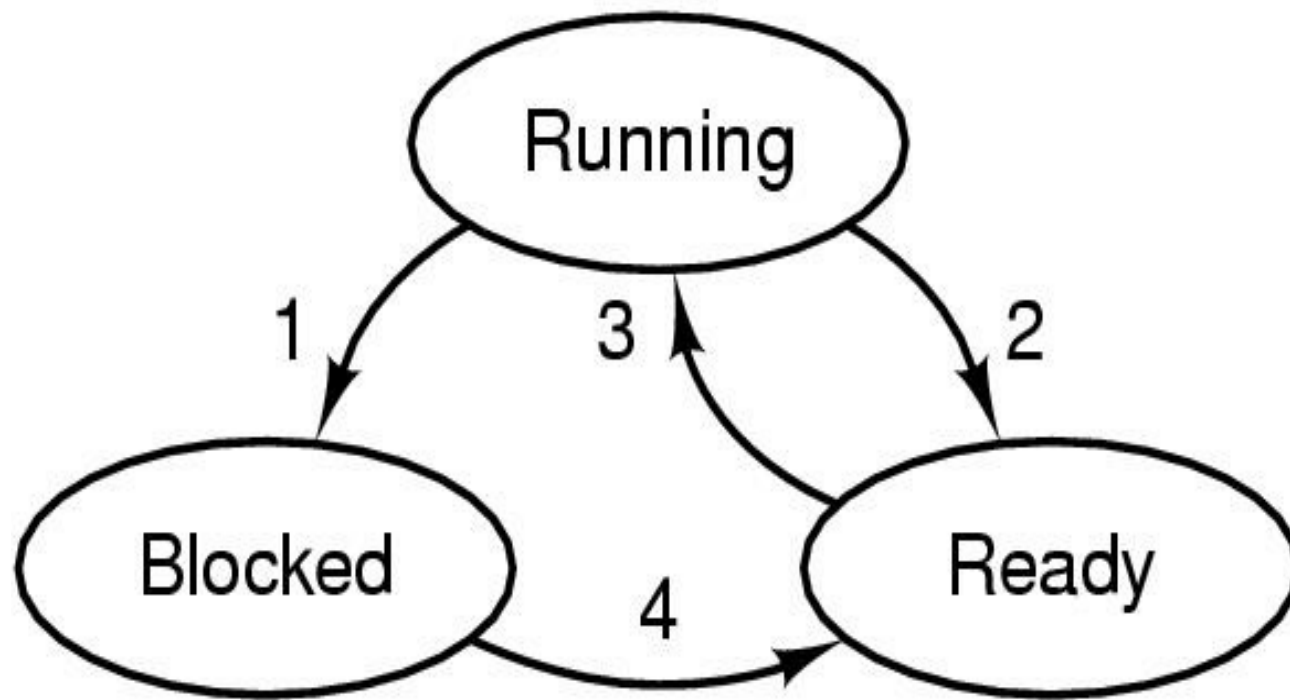


1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

- Possible process states
  - Running (occupy CPU)
  - Blocked
  - Ready (does not occupy CPU)
  - Other states: suspended, terminated



# Processes



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

- Possible process states
  - Running (occupy CPU)
  - Blocked
  - Ready (does not occupy CPU)
  - Other states: suspended, terminated

Question: in a single processor machine, how many process can be in running state?

# Creating a Process



- What UNIX call creates a process?

# Creating a Process - fork()



- What UNIX call creates a process?
- fork() duplicates a process so that instead of one process you get two.
  - The new process and the old process both continue in parallel from the statement that follows the fork()

# Creating a Process - fork()



- What UNIX call creates a process?
- fork() duplicates a process so that instead of one process you get two.
  - The new process and the old process both continue in parallel from the statement that follows the fork()
- How can you tell the two processes apart?

# Creating a Process - fork()



- What UNIX call creates a process?
- fork() duplicates a process so that instead of one process you get two.
  - The new process and the old process both continue in parallel from the statement that follows the fork()
- How can you tell the two processes apart?
  - fork() returns
    - 0 to child
    - -1 if fork fails
    - Child's PID to parent process

# Creating a Process - fork()



- What UNIX call creates a process?
- fork() duplicates a process so that instead of one process you get two.
  - The new process and the old process both continue in parallel from the statement that follows the fork()
- How can you tell the two processes apart?
  - fork() returns
    - 0 to child
    - -1 if fork fails
    - Child's PID to parent process
- If the parent code changes a global variable, will the child see the change?

# Creating a Process - fork()



- What UNIX call creates a process?
- fork() duplicates a process so that instead of one process you get two.
  - The new process and the old process both continue in parallel from the statement that follows the fork()
- How can you tell the two processes apart?
  - fork() returns
    - 0 to child
    - -1 if fork fails
    - Child's PID to parent process
- If the parent code changes a global variable, will the child see the change?
  - Nope! On fork, child gets new program counter, stack, file descriptors, heap, globals, pid!

# Creating a Process



- What if we need the child process to execute different code than the parent process?



# Creating a Process - exec()



- What if we need the child process to execute different code than the parent process?
  - Exec function allows child process to execute code that is different from that of parent
  - Exec family of functions provides a facility for overlaying the process image of the calling process with a new image.
  - Exec functions return -1 and sets errno if unsuccessful

# Threads vs. Processes



- What is the difference between a thread and a process?

# Threads vs. Processes



- What is the difference between a thread and a process?
  - Both provided independent execution sequences, but...
  - Each process has its own memory space
    - Remember how child processes can't see changes to parent's global variable??
  - Threads run in a shared memory space

# Threads vs. Processes



- What is POSIX?
- How do you create a POSIX thread?

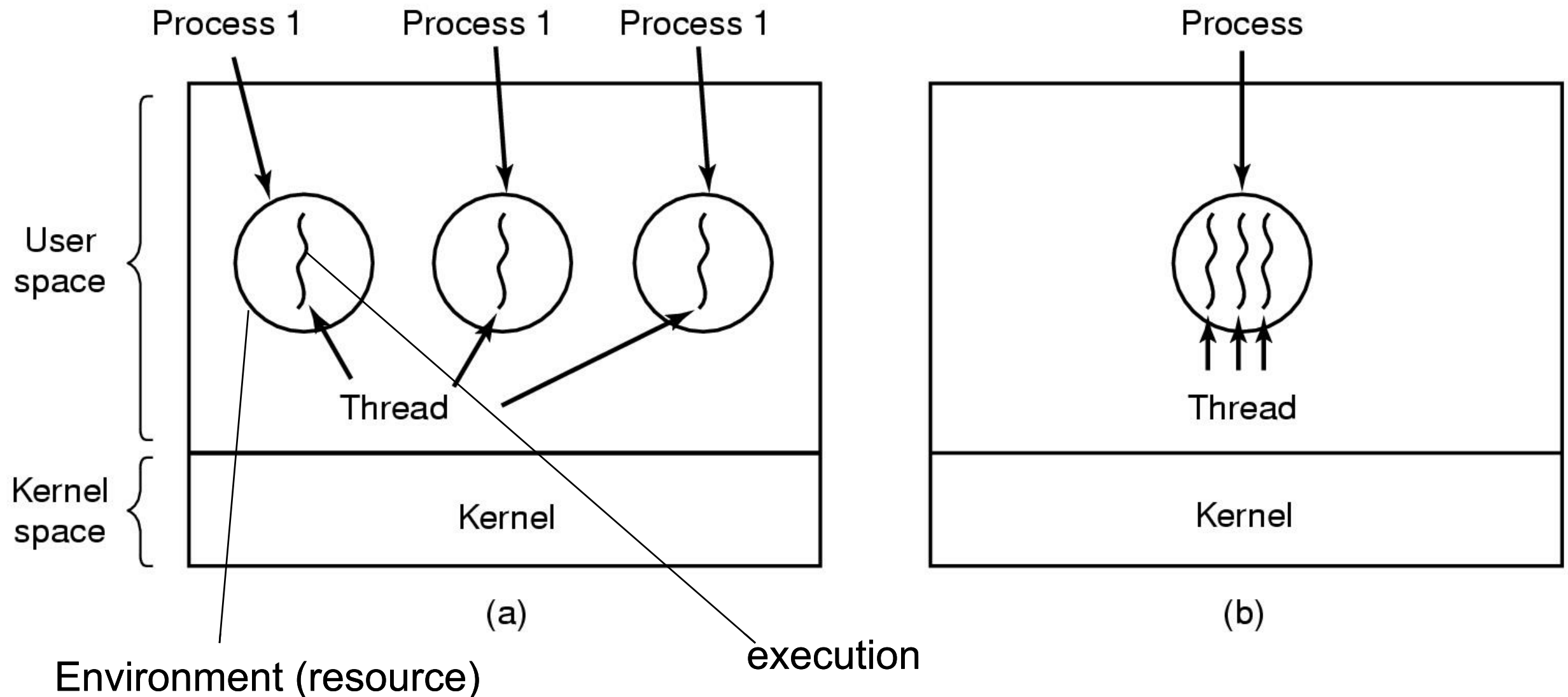
# Threads vs. Processes



- What is POSIX?
- How do you create a POSIX thread?

POSIX function	description
pthread_create	create a thread
pthread_detach	set thread to release resources
pthread_equal	test two thread IDs for equality
pthread_exit	exit a thread without exiting process
pthread_kill	send a signal to a thread
pthread_join	wait for a thread
pthread_self	find out own thread ID

# Threads: Lightweight Proc's



- (a) Three processes each with one thread
- (b) One process with three threads

# Threads: Kernel v. User



- What is the difference between kernel and user threads? Pros and Cons?

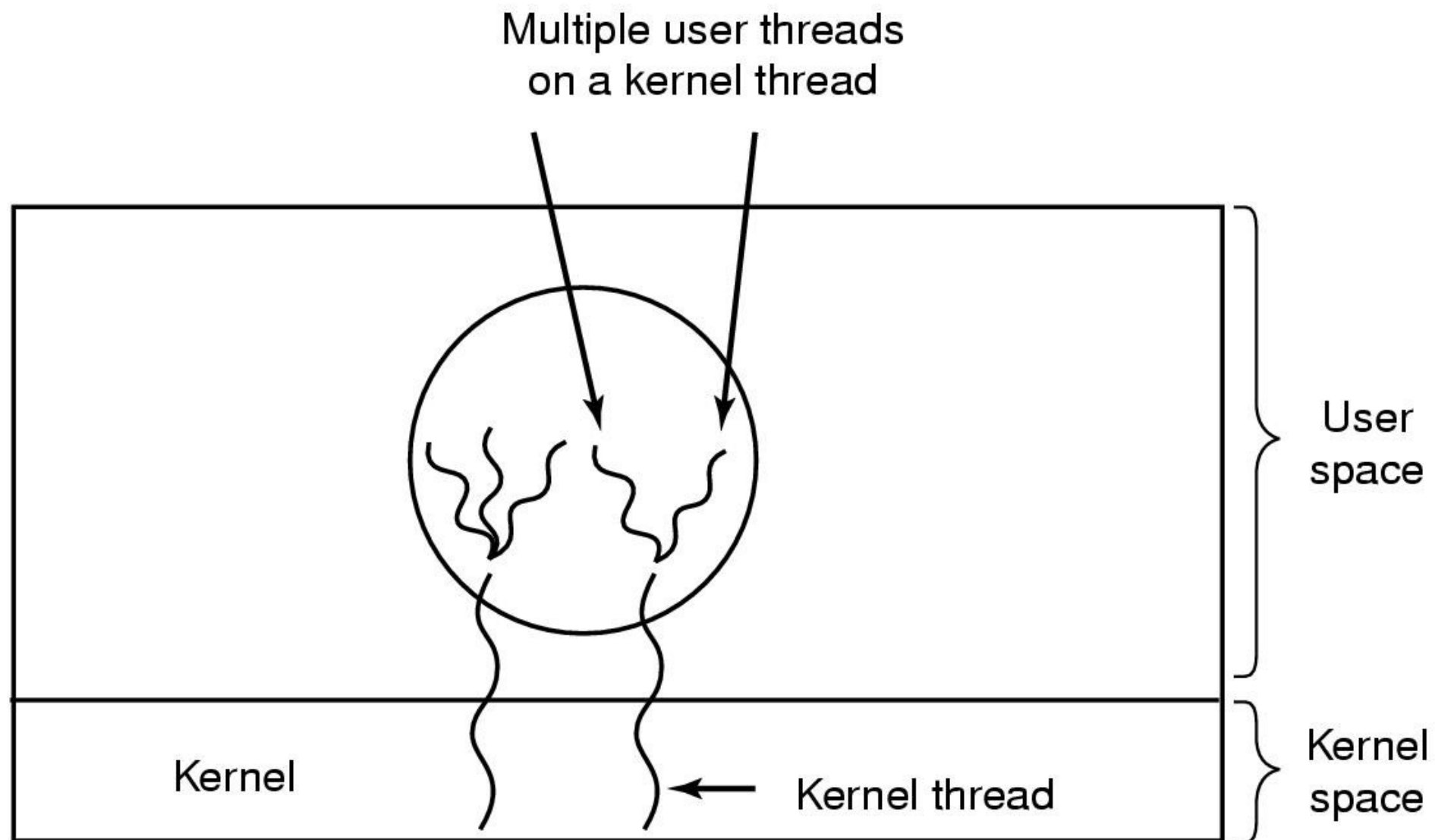
# Threads: Kernel v. User



- What is the difference between kernel and user threads? Pros and Cons?
- Kernel thread packages
  - Each thread can make blocking I/O calls
  - Can run concurrently on multiple processors
- Threads in User-level
  - Fast context switch
  - Customized scheduling



# Hybrid Threads (Solaris)



M:N model multiplexes N user-level threads onto M kernel-level threads

Good idea? Bad Idea?

# Synchronization



- Processes and threads can be preempted at arbitrary times, which may generate problems.
- Example: What is the execution outcome of the following two threads (initially  $x=0$ )?

**Thread 1:**

**Read X**  
**Add 1**  
**Write X**

**Thread 2:**

**Read X**  
**Add 1**  
**Write X**

How do we account for this?

# Critical Regions/Sections



```
Process {  
    while (true) {  
        ENTER CRITICAL SECTION  
        Access shared variables;  
        LEAVE CRITICAL SECTION  
        Do other work  
    }  
}
```



- Simplest and most efficient thread synchronization mechanism
- A special variable that can be either in
  - **locked state**: a distinguished thread that holds or owns the mutex; or
  - **unlocked state**: no thread holds the mutex
- When several threads compete for a mutex, the losers block at that call
  - The mutex also has a queue of threads that are waiting to hold the mutex.
- POSIX does not require that this queue be accessed FIFO.
- Helpful note — Mutex is short for “Mutual Exclusion”

# POSIX Mutex Functions



- `int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);`
  - Also see `PTHREAD_MUTEX_INITIALIZER`
- `int pthread_mutex_destroy(pthread_mutex_t *mutex);`
- `int pthread_mutex_lock(pthread_mutex_t *mutex);`
- `int pthread_mutex_trylock(pthread_mutex_t *mutex);`
- `int pthread_mutex_unlock(pthread_mutex_t *mutex);`

# Semaphores



Pseudocode for a blocking implementation of semaphores:

```
void wait (semaphore_t *sp)
    if (sp->value > 0) sp->value--;
    else {
        <Add this process to sp->list>
        <block>
    }

void signal (semaphore_t *sp)
    if (sp->list != NULL)
        <remove a process from sp->list,
        put it in ready state>
    else sp->value++;
```



- Basic scheduling algorithms
  - FIFO (FCFS)
  - Shortest job first
  - Round Robin
  - Priority Scheduling



- Basic scheduling algorithms
  - FIFO (FCFS)
  - Shortest job first
  - Round Robin
  - Priority Scheduling
  
- What is an optimal algorithm in the sense of maximizing the number of jobs finished?

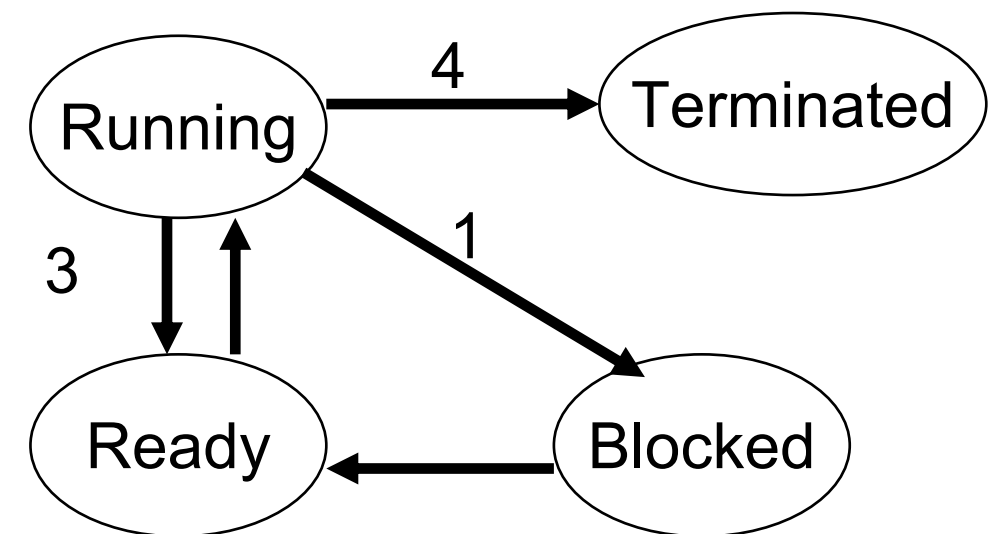




- Basic scheduling algorithms
  - FIFO (FCFS)
  - Shortest job first
  - Round Robin
  - Priority Scheduling
  
- What is an optimal algorithm in the sense of meeting the most deadlines (of real time tasks)?

- **Non-preemptive scheduling:**

- The running process keeps the CPU until it **voluntarily** gives up the CPU
  - process exits
  - switches to blocked state
  - 1 and 4 only (no 3)



- **Preemptive scheduling:**

- The running process can be interrupted and must release the CPU (can be **forced** to give up CPU)

# Signals



- What is a signal in UNIX/Linux?

# Signals



- What is a signal in UNIX/Linux?
  - A way for one process to send a notification to another
  - A signal can be “caught”, “ignored”, or “blocked”



- What is a signal in UNIX/Linux?
  - A way for one process to send a notification to another
  - A signal can be “caught”, “ignored”, or “blocked”
- Signal is **generated** when the event that causes it occurs.
- Signal is **delivered** when a process receives it.
- The **lifetime** of a signal is the interval between its generation and delivery.
- Signal that is generated but not delivered is **pending**.
- Process **catches** signal if it executes a **signal handler** when the signal is delivered.
- Alternatively, a process can **ignore** a signal when it is delivered, that is to take no action.
- Process can temporarily prevent signal from being delivered by **blocking** it.
- **Signal Mask** contains the set of signals currently blocked.

# POSIX-required Signals\*



Signal	Description	default action
SIGABRT	process abort	implementation dependent
<b>SIGALRM</b>	<b>alarm clock</b>	<b>abnormal termination</b>
SIGBUS	access undefined part of memory object	implementation dependent
SIGCHLD	child terminated, stopped or continued	ignore
SIGILL	invalid hardware instruction	implementation dependent
<b>SIGINT</b>	<b>interactive attention signal (usually ctrl-C)</b>	<b>abnormal termination</b>
<b>SIGKILL</b>	<b>terminated (cannot be caught or ignored)</b>	<b>abnormal termination</b>

*\* Not an exhaustive list*

# POSIX-required Signals\*



Signal	Description	default action
<b>SIGSEGV</b>	<b>Invalid memory reference</b>	<b>implementation dependent</b>
SIGSTOP	Execution stopped	stop
<b>SIGTERM</b>	<b>termination</b>	<b>Abnormal termination</b>
SIGTSTP	Terminal stop	stop
SIGTTIN	Background process attempting read	stop
SIGTTOU	Background process attempting write	stop
<b>SIGURG</b>	<b>High bandwidth data available on socket</b>	<b>ignore</b>
<b>SIGUSR1</b>	<b>User-defined signal 1</b>	<b>abnormal termination</b>

*\* Not an exhaustive list*

# User-generated Signals



- How can you send a signal to a process from the command line?



# User-generated Signals



- How can you send a signal to a process from the command line?
- **kill** 🥵

# User-generated Signals



- How can you send a signal to a process from the command line?
- **kill** 🥵
- `kill -1` will list the signals the system understands
- `kill [-signal] pid` will send a signal to a process.
  - The optional argument may be a name or a number (default is SIGTERM).
- To unconditionally kill a process, use:
  - `kill -9 pid` which is  
`kill -SIGKILL pid`.

# Signal Masks



- A process can temporarily prevent a signal from being delivered by **blocking** it.
- **Signal Mask** contains a set of signals currently blocked.
- **Important!** Blocking a signal is different from ignoring signal. Why?

# Signal Masks



- A process can temporarily prevent a signal from being delivered by **blocking** it.
- **Signal Mask** contains a set of signals currently blocked.
- **Important!** Blocking a signal is different from ignoring signal. Why?
- When a process blocks a signal, the OS does not deliver signal until the process unblocks the signal
  - A **blocked** signal is not delivered to a process until it is unblocked.
- When a process ignores signal, signal is delivered and the process handles it by throwing it away.



# Deadlocks





# Deadlocks



When do deadlocks occur (hint: 4 preconditions)?





# Deadlocks



**When do deadlocks occur (hint: 4 preconditions)?**

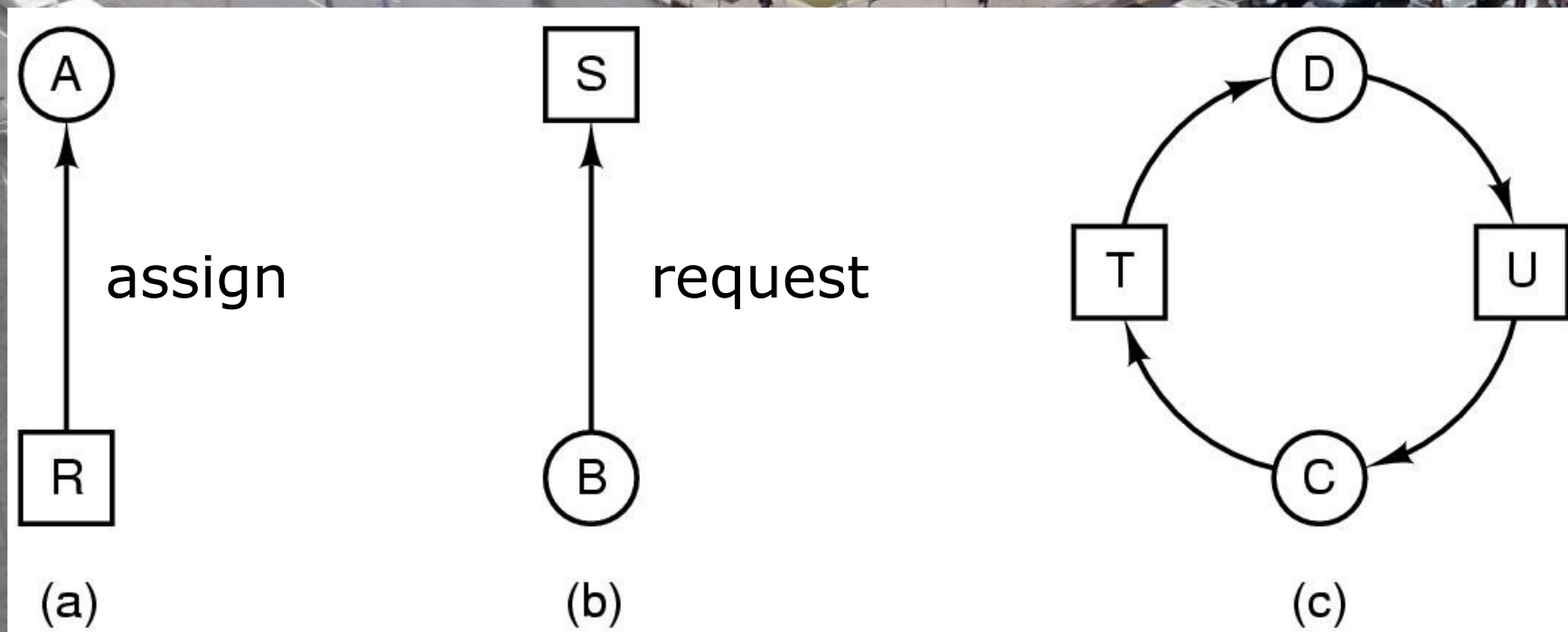
- 
- Mutual exclusion
  - Hold and wait condition
  - No preemption condition
  - Circular wait condition



# Deadlocks



## Resource Allocation Graphs



- resource R assigned to process A
- process B is requesting/waiting for resource S
- process C and D are in deadlock over resources T and U



## Strategies for Dealing with Deadlocks

- ~~shouting~~
- detection and recovery
- dynamic avoidance (at run-time)
- prevention (by offline design)
  - by negating one of the four necessary conditions

