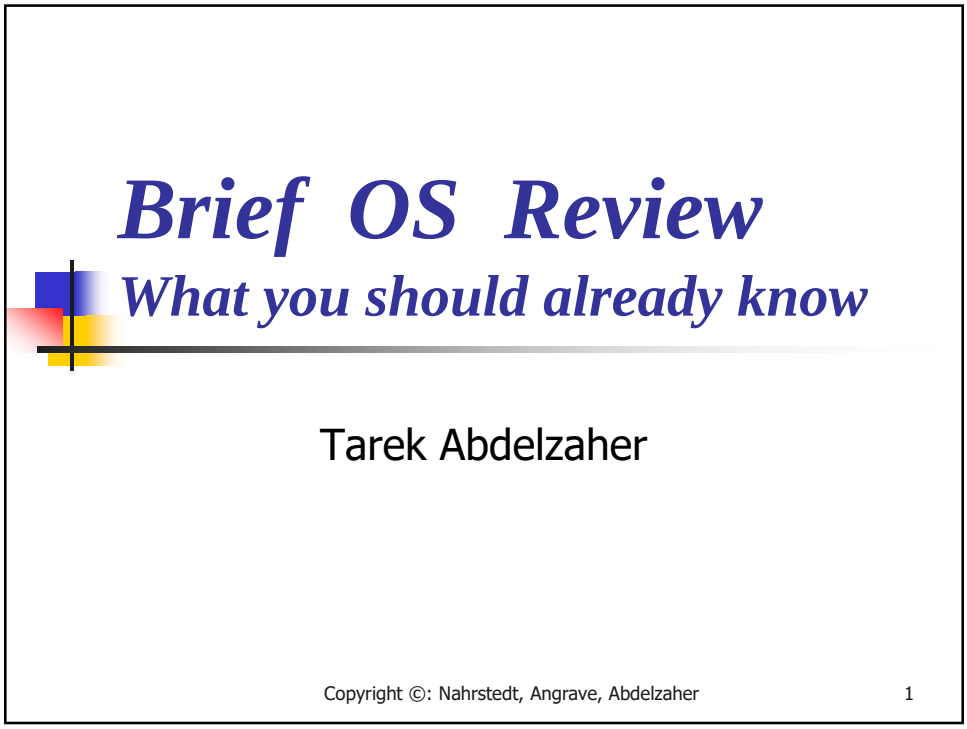
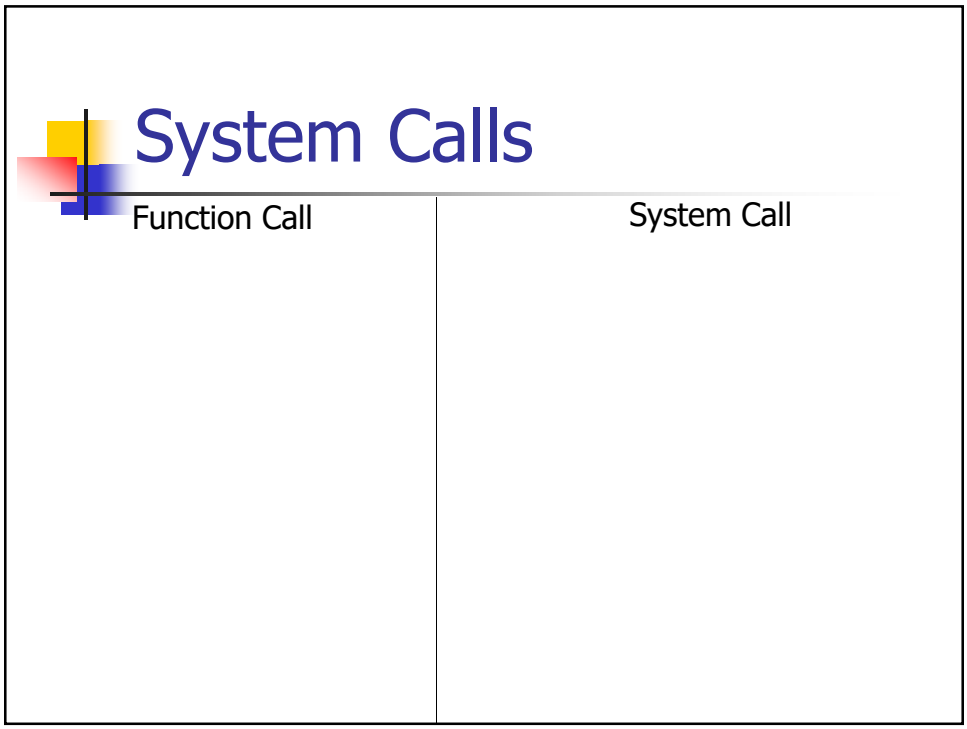
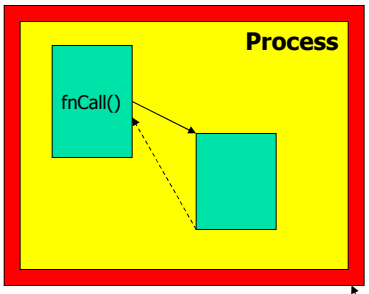

Brief OS Review
What you should already know
Tarek Abdelzaher
Copyright ©: Nahrstedt, Angrave, Abdelzaher
1


System Calls
Function Call
System Call

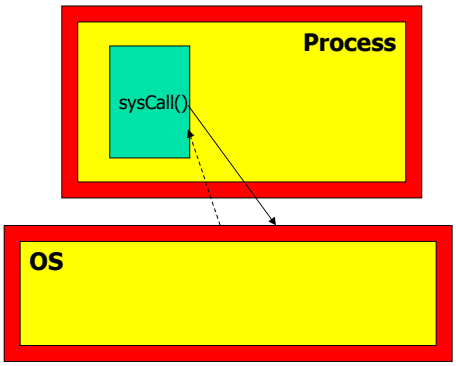
System Calls

Function Call



Caller and callee are in the same Process
- Same user
- Same "domain of trust"

System Call

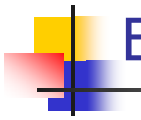


- OS is trusted; user is not.
- OS has super-privileges; user does not
- Must take measures to prevent abuse

3

Examples of System Calls

4



Examples of System Calls

- Example:
  - getuid() //get the user ID
  - fork() //create a child process
  - exec() //executing a program
- Don't mix system calls with standard library calls
  - Differences?
  - Is printf() a system call?
  - Is rand() a system call?

5



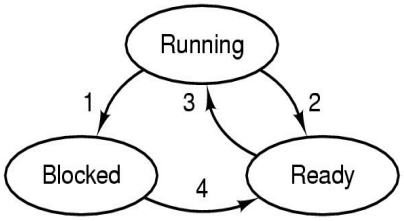
I/O Library Calls versus System Calls

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

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

6

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?

7

Creating a Process

- What UNIX call creates a process?

8



Creating a Process – fork()

- 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()

9



Creating a Process – fork()

- 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?

10



Creating a Process – fork()

- 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()
- fork() returns
  - 0 if child
  - -1 if fork fails
  - Child's PID if parent process

11



Creating a Process – fork()

- 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()
- fork() returns
  - 0 if child
  - -1 if fork fails
  - Child's PID if parent process
- If the parent code changes a global variable, will the child see the change?

12



Creating a Process – fork()

- 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()
- fork() returns
  - 0 if child
  - -1 if fork fails
  - Child's PID if parent process
- Child gets new program counter, stack, file descriptors, heap, globals, pid!

13



How to Change the Child Image?

14



exec() Function

- 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

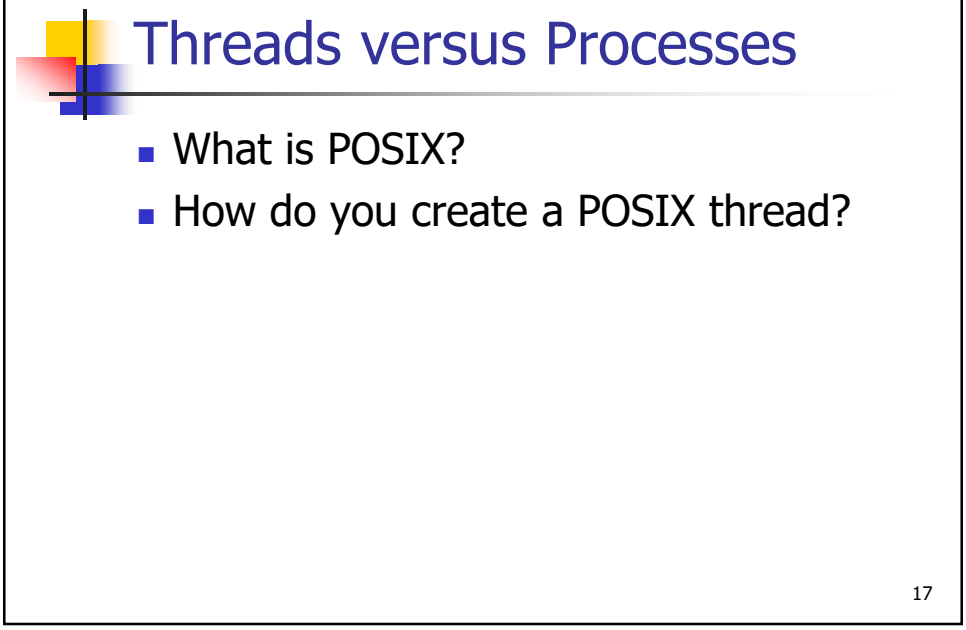
15

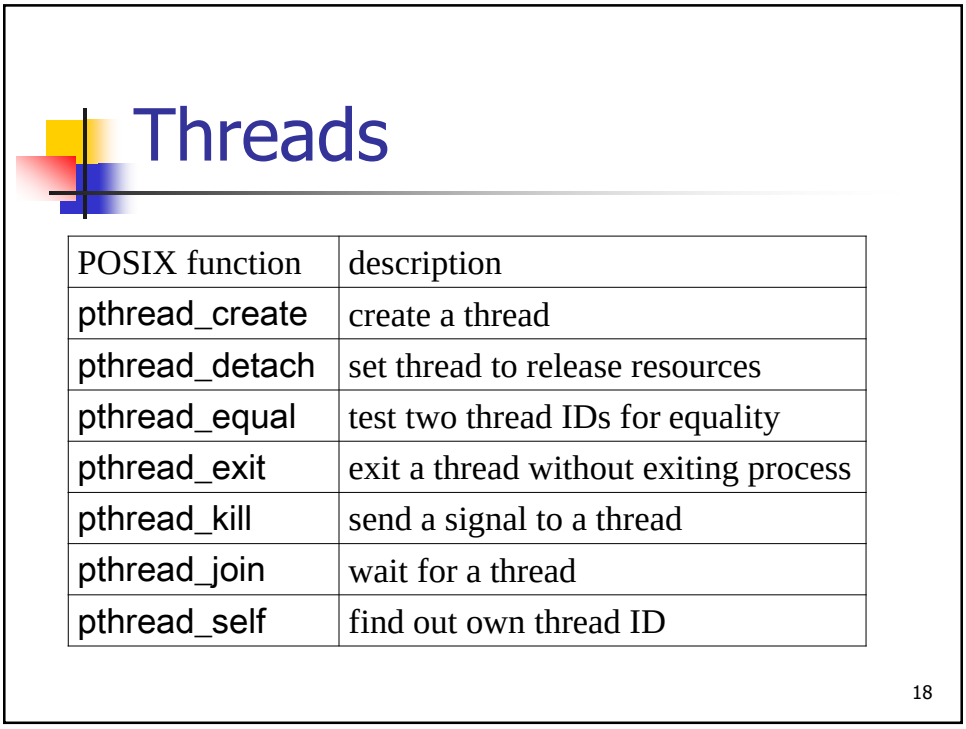


Threads versus Processes

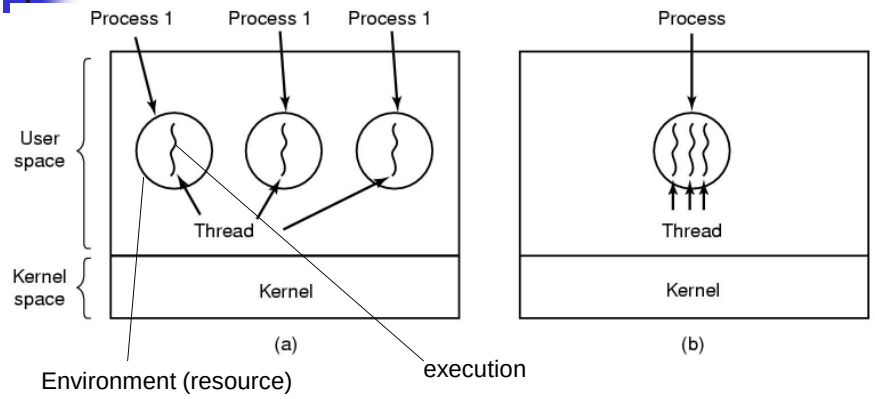
- What is the difference between a thread and a process?

16


Threads versus Processes
What is POSIX?
How do you create a POSIX thread?
17


Threads
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
18

Threads: Lightweight Processes



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

19

Kernel versus User Threads:

- What is the difference between kernel and user threads?

20



Kernel versus User Threads:

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

21

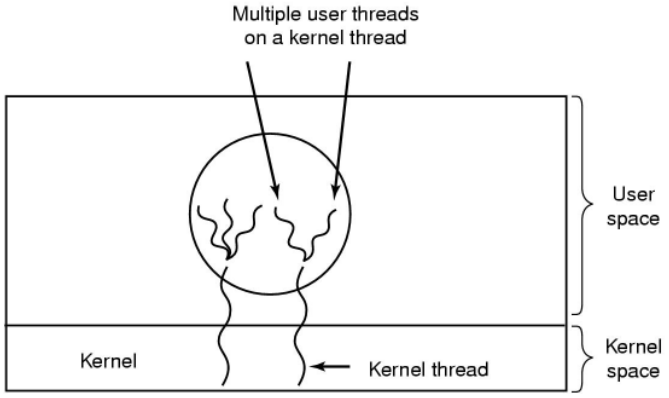


Kernel versus User Threads: Trade-offs?

- 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

22

Hybrid Implementations (Solaris)



- Multiplexing user-level threads onto kernel-level threads

23

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:	**Thread 2:**
Read X	**Read X**
Add 1	**Add 1**
Write X	**Write X**

24



Critical Region (Critical Section)

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

25

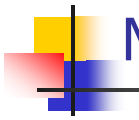


Semaphores

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

signal (sem_t *sp)
   if (sp->list != NULL)
        remove next process from sp->list;
   else sp->value++;
```

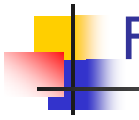
26



Mutex

- 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.

27



POSIX Mutex-related 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);

28



Scheduling

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

29



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?

30

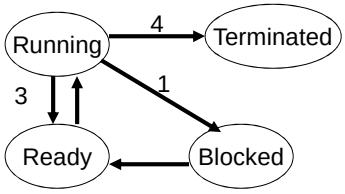
Scheduling

- 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)?

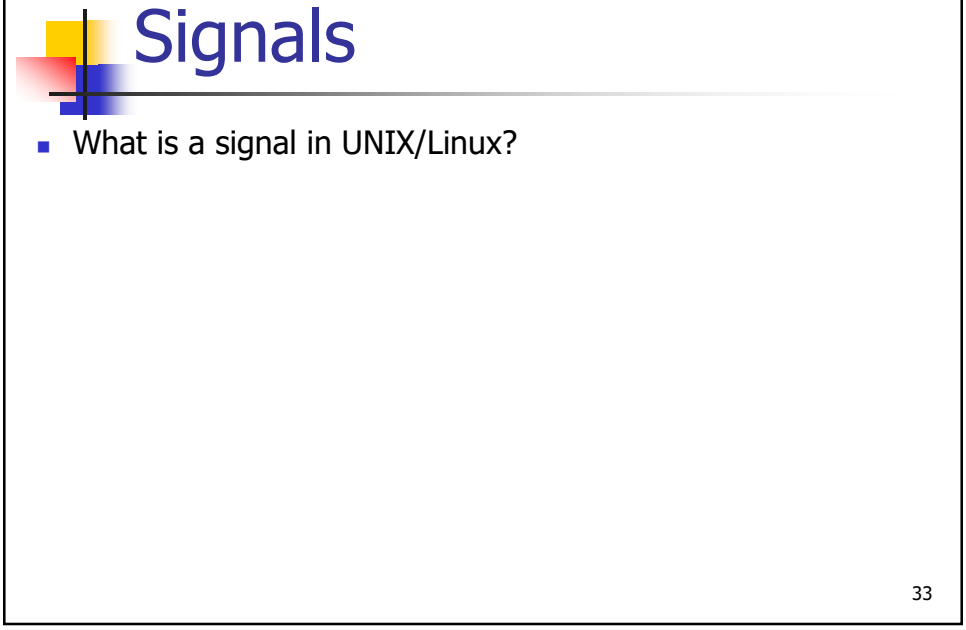
31

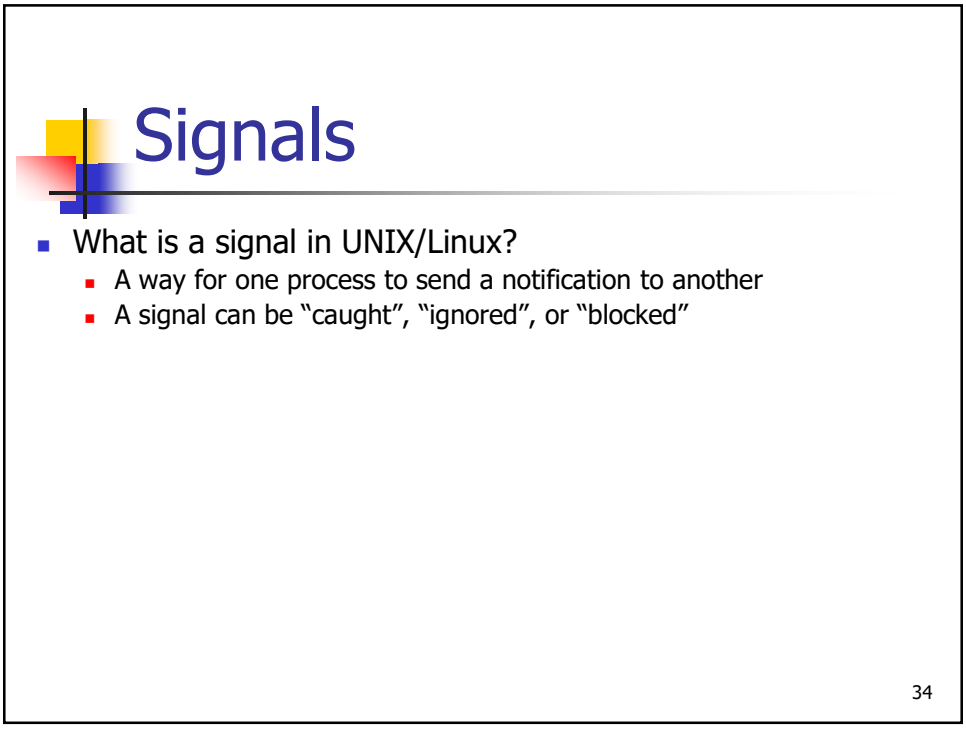
Preemptive vs. Non-preemptive scheduling

- **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)



32


Signals
What is a signal in UNIX/Linux?
33


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"
34



Signals

- 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.

35



Examples of POSIX Required Signals

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

36

Examples of POSIX Required Signals

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

37

Command Line Generates Signals

- You can send a signal to a process from the command line using **kill**
- kill -l 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.

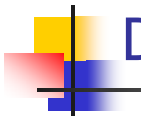
38



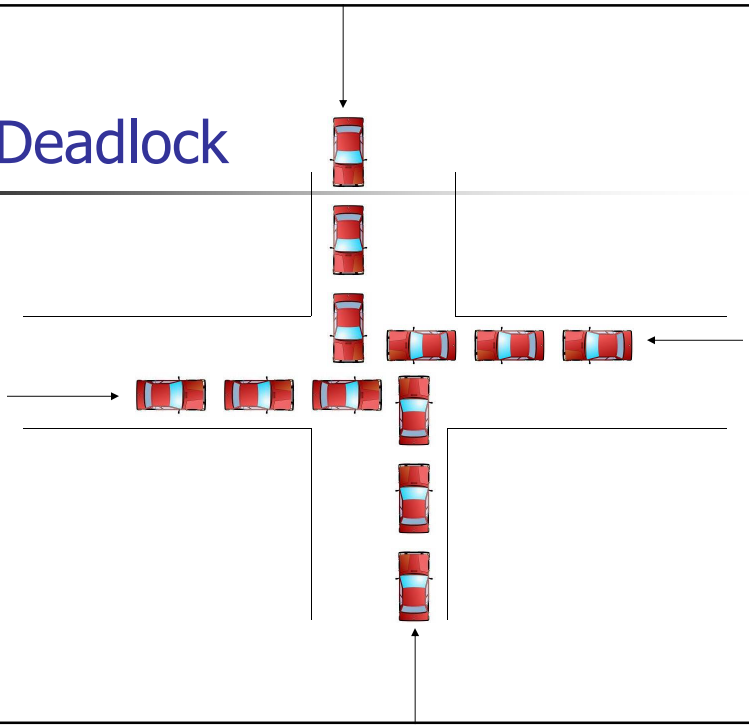
Signal Masks

- Process can temporarily prevent 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.

39



Deadlock



40



Deadlock

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

41

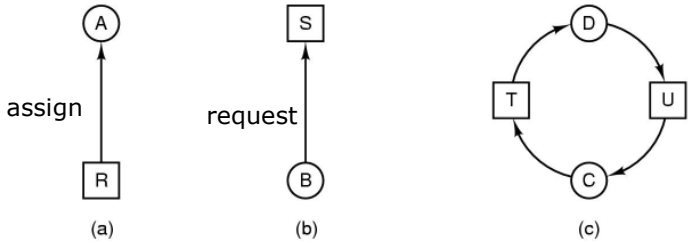


Deadlock

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

42

Resource Allocation Graph



- 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

43

Strategies

- Strategies for dealing with Deadlocks
  - detection and recovery
  - dynamic avoidance (at run-time)
  - prevention (by off-line design)
    - negating one of the four necessary conditions

44



Other Issues

- Memory management
- Files
- I/O

45