



CS 415

Operating Systems

Threads

Prof. Jieyang Chen

Department of Computer and Information Science

Fall 2025



UNIVERSITY OF OREGON

Logistics

- Keep working hard on Project 1
 - Due 11:59pm on Friday
 - Please work individually
- Labs this week focus on answering questions
- Take advantage of TA office hours
 - Keep checking Canvas announcements
- Read Chapter 4 on threads
- Encourage you to come to office hours
 - Few people are coming now

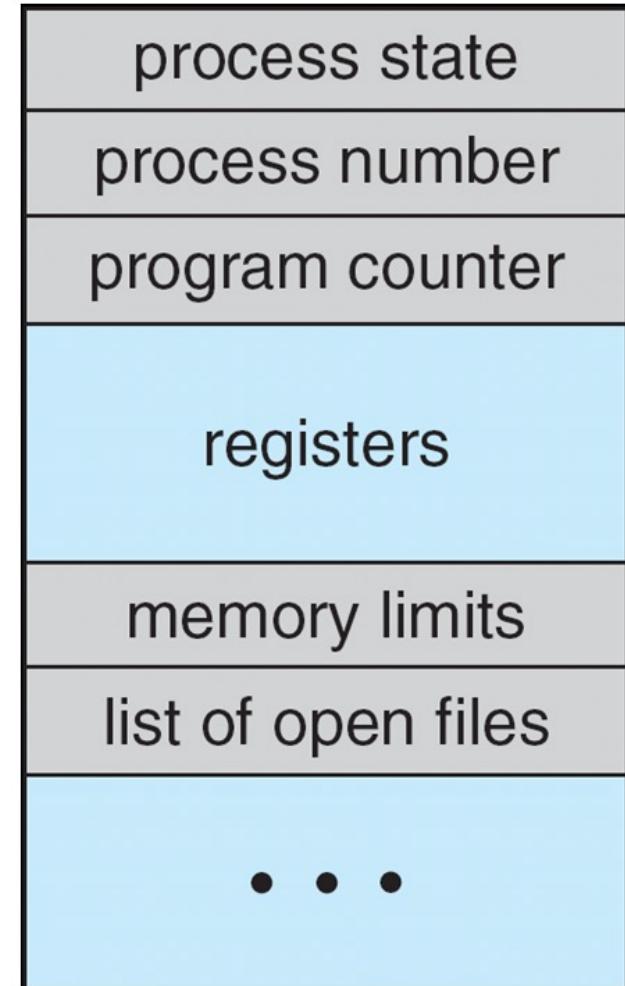
Outline

- Processes revisited
- Threads

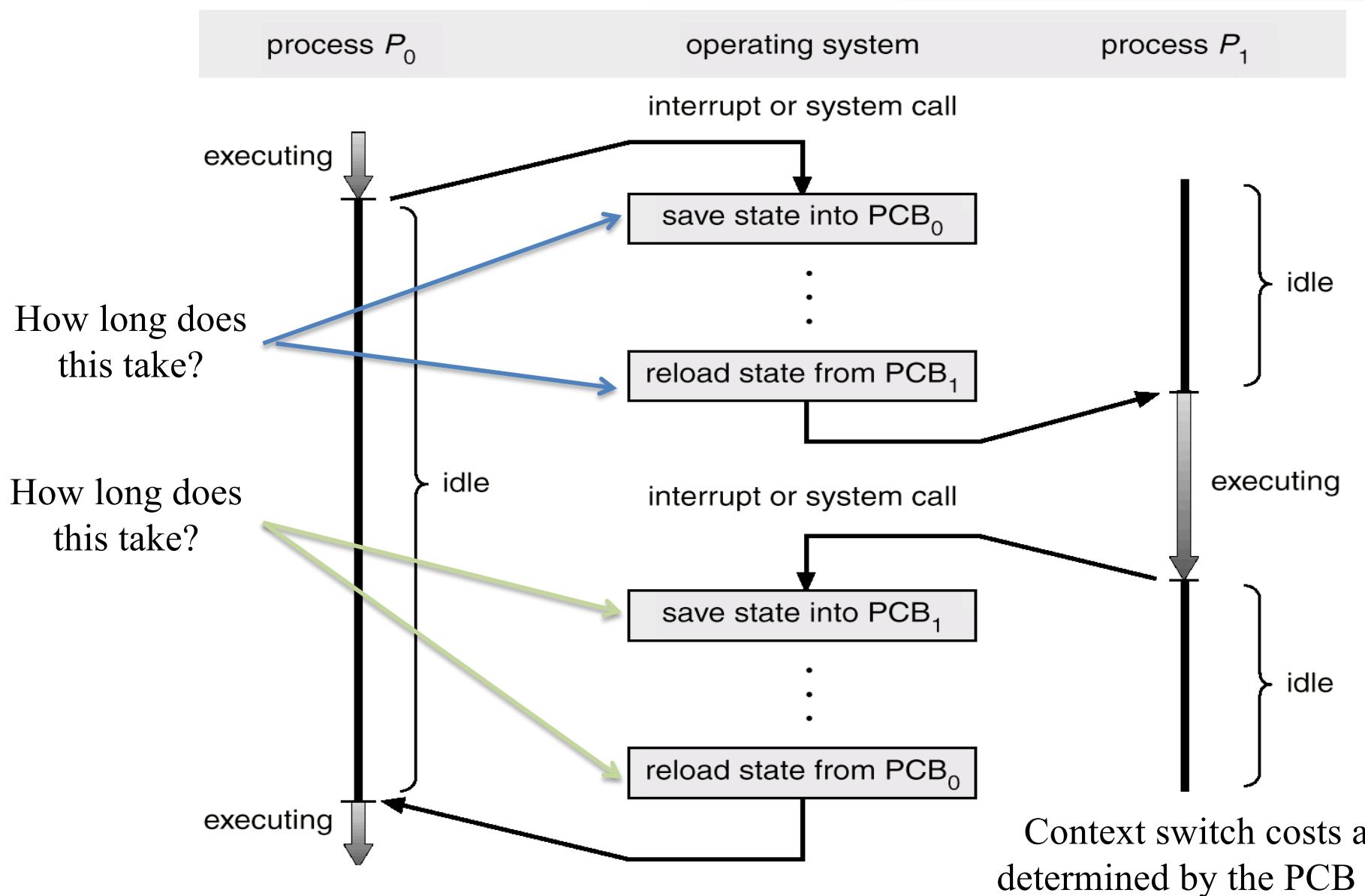
Process Control Block

- Information associated with each process (aka *task control block*)
- Process state
 - Running, waiting, ...
- Program counter
 - Location of instruction to next execute
- CPU registers
 - Contents of all process-centric registers
- CPU scheduling information
 - Priorities, scheduling queue pointers, ...
- Memory-management information
 - Memory allocated to the process
 - Memory limits and other associated information
- Accounting information
 - CPU used, elapsed time, time limits, ...
- I/O status information
 - I/O devices allocated to process, list of open files

PCB



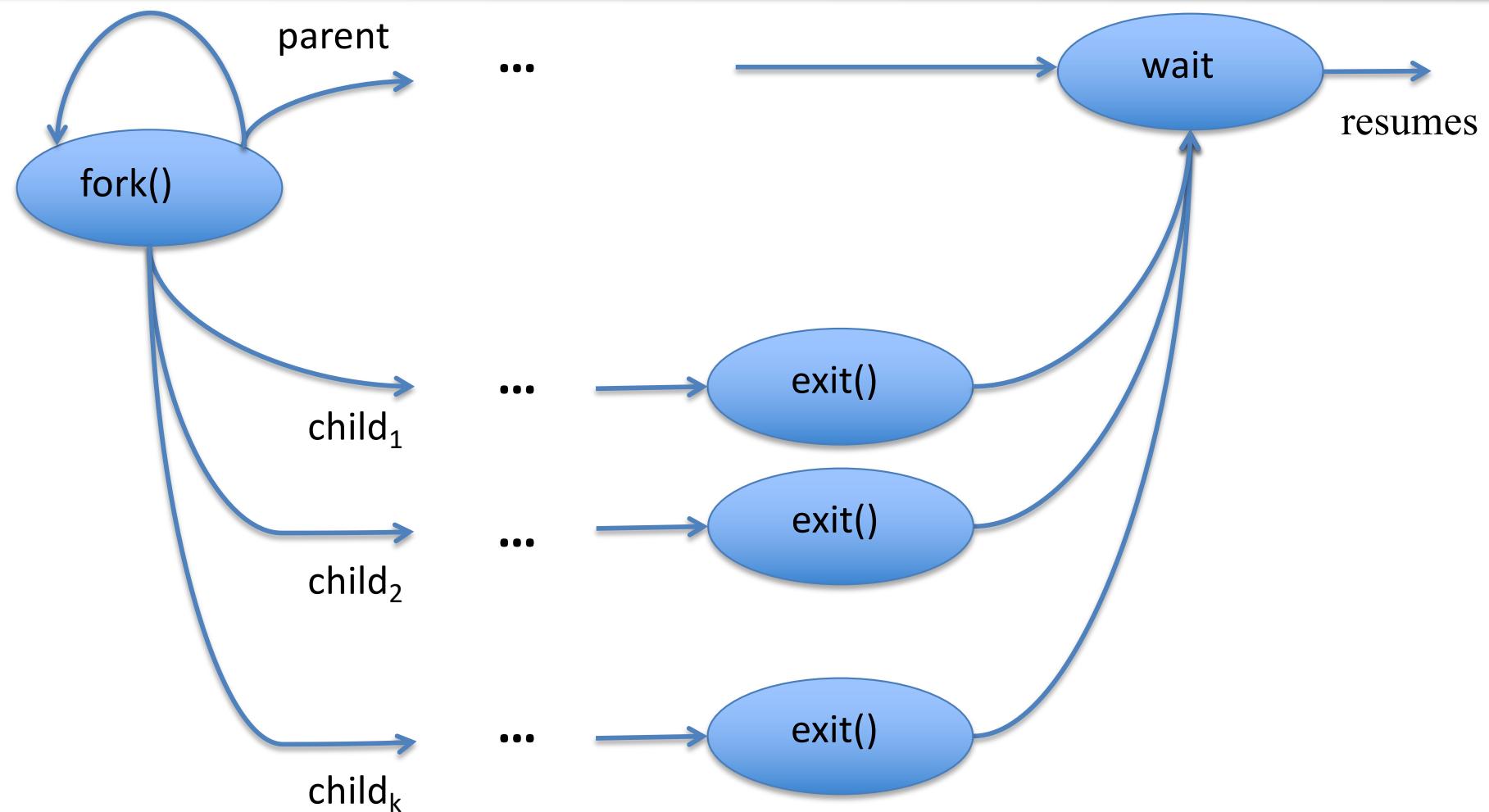
Process Context Switch



Process Model

- Much of the OS's job is keeping processes from interfering with each other ... Why?
- Each process has its OWN resources to use
 - Program code to execute, address space, files, ...
- Processes are good for *isolation (protection)*
 - Prevent one process from affecting another process
- Processes are *heavyweight* ... Why?
 - Pay a price for isolation
 - A full “process swap” is required for multiprocessing
 - There is lots of process state to save and restore
 - OS must context switch between them
 - ◆ intervene to save/restore all process state
- Is there an alternative?

Program with Multiple Processes



Program Creation System Calls

- *fork()*
 - Copy address space of parent and data (copy on write) to child process
 - *vfork()*
 - Share the parent's address space
 - Shares address space AND physical memory between parent and child
 - Parent blocks and waits for child to exit or call *exec()*
 - *exec()*
 - Load new program and replace address space (for new child code)
 - Some resources may be transferred (open file descriptors)
 - Specified by arguments
 - *clone()*
 - Like *fork()* BUT child shares some process context
 - More explicit control over what is shared
 - Calls a function pointed to by argument
 - Process address space and physical memory can be shared!
- Child process is independently executing
- Child process is like in *fork()*, but it is much lighterweight!

Process Reimagined

- A process is “a program in execution”
 - Memory address space containing code and data
 - State information (PC, register, SP) => PCB details
 - Allocated physical memory
 - Other resources (e.g., open file descriptors)
- Consider a process in 2 respects (categories)
 - 1) A collection of resources required for execution
 - ◆ code, address space, memory, open files, ...
 - 2) A “*thread of execution*”
 - ◆ current state of execution (CPU state)
 - ◆ where the execution is now (PC)
- Suppose we think about these separately!
 - Resources
 - Execution state



Terminology

- ***Multiprogramming***
 - Running *multiple programs* on a computer at the same time
 - ◆ Concurrent (logically) execution
 - Each program is a process (or set of processes)
 - Processes of different programs are independent
- ***Multiprocessing***
 - Running *multiple processes* on a computer at the same time
 - Physical concurrently
 - OS schedules processes on processor(s)
- ***Multithreading***
 - Multiple *threads of execution (execution contexts)* in a single address space (of a process)
 - OS executes threads on processor(s)
 - Can be logical or physical concurrent

What Makes up a (Multithreaded) Process?

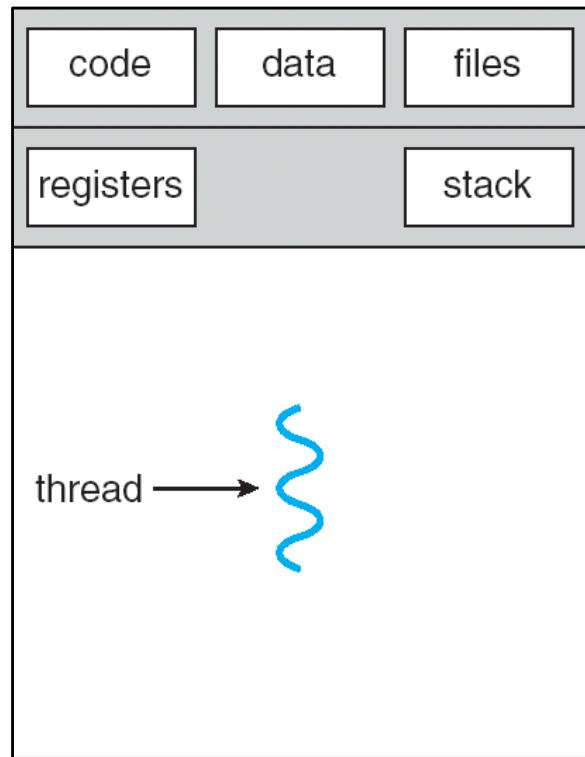
- Thread of execution (sequence of instructions) on a CPU
 - Program counter *per thread*
 - Registers *per thread*
- Memory
 - Address space (code and data)
 - ◆ address space and physical memory assigned is shared *process*
 - Stack
 - ◆ each thread has its own stack pointer
 - ◆ separately located in the address space*per thread*
 - Heap
 - + private dynamic memory
 - ◆ separately located in the address space*process*
per thread
- I/O
 - Share files, sockets, ... *process*
- PCB
 - A thread lives in a process and uses the PCB *process*

Single-Threaded vs. Multithreaded

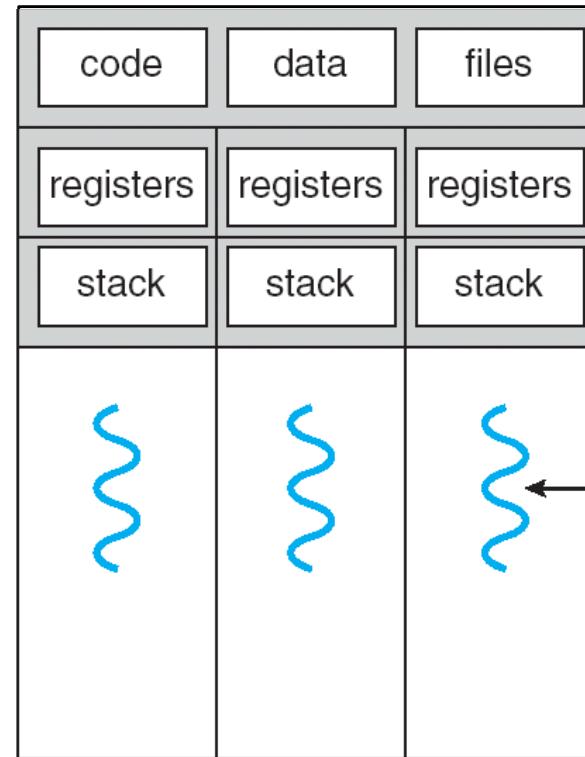
- A normal process is a special case of a multithreaded process
 - A process that contains **just one thread!**
- A multithreaded process has multiple threads
 - Where do they live? How do they get created?

process
context

thread
context



single-threaded process



multithreaded process

process
context

thread
context
(each thread
has its own)

Why Multithreaded Applications?

- Multiple threads share a common address space
 - More correctly, they share process resources, including memory!
- Why would you want to do that?
- Concurrency!
 - How do you get concurrency?
 - One way is to create multiple processes
 - Use IPC to support process-level concurrency
 - Issues?
- Some applications want to share data structures among concurrently executing parts of the computation
 - Is this possible with processes? => shared segments
 - Is it difficult with processes? => well, it is not that easy
 - Again, use IPC for process-level data sharing
- What is the problem? What is the solution?

Advantages of Threads

- Suppose we do not care if the OS enforces resource separation
 - This is a trust issue, in part (back off on isolation / protection)
 - Will introduce issues with respect to concurrency ... What kind?
- Improve responsiveness
 - Possible to have a thread of execution that just responds to events
- Resource sharing is facilitated
 - All threads in a process have equal access to resources
- Economy of resources
 - Thread-level resources are “cheaper” than process resources!
 - Threads are “lighter weight”
- Utilization of multiprocessors
 - Run multiple threads on multiple processes without the overhead of running multiple processes

Working with Threads

- In a C program
 - *main()* procedure defines the first thread
 - C programs always start at *main()*
- Now you want to create a second thread
 - Allocate resources to maintain a second execution context in **the same address space**
 - ◆ think about what state will be needed for a thread
 - Want something similar to *fork()* but simpler
 - ◆ supply **a function** when starting the new thread
 - Remember this creates another thread of execution

Can a thread call exec() to run a new program?

- Yes, but the results are often NOT what you want
- exec() replaces the entire current process image with a new program.
 - All previous code, data, heap, stack, and threads are destroyed.
 - All other threads are terminated immediately.
 - Only the thread that called exec() survives long enough to run the new program's entry point (main()).
 - The process doesn't become “multi-threaded” anymore — it becomes the new program, running in a single thread.

Threads vs. Processes

- Easier to create than a new process
- Less time to terminate a thread than a process
- Less time to switch between two threads
 - Within the same process
- Less communication overheads
 - Communicating between the threads of one process is simple because the threads share everything
 - Address space is shared ...
 - ... thus memory is shared

Which is Cheaper?

- Create new process or create new thread (in existing process)?
- Context switch between processes or threads?
- Interprocess or interthread communication?
- Sharing memory between processes or threads?
- Terminating a process or terminating a thread (not the last one)?

Process creation method	Time (sec), elapsed (real)
<i>fork()</i>	22.27 (7.99)
<i>vfork() (faster fork)</i>	3.52 (2.49)
<i>clone()</i>	2.97 (2.14)

Time to create 100,000 processes (Linux 2.6 kernel, x86-32 system)

clone() creates a lightweight Linux process (thread)

Implications?

- Consider a web server on a Linux platform
- Measure 0.22 ms per *fork()*
 - Maximum of $(1000 / 0.22) = 4545.5$ connections/sec
 - 0.45 billion connections per day per machine
 - ◆ fine for most servers
 - ◆ too slow for a few super-high-traffic front-line web services
- Facebook serves $O(750 \text{ billion})$ page views per day
 - Guess ~1-20 HTTP connections per page
 - Would need 3,000 – 60,000 machines just to handle *fork()*, without doing any work for each connection!
- What is the problem here?

Thread Attributes

- Global to process:
 - memory
 - PID, PPID, GID, SID
 - controlling term
 - process credentials
 - record locks
 - FS information
 - timers
 - resource limits
 - and more...
- Local to specific thread:
 - thread ID
 - CPU state and stack
 - signal mask
 - thread-specific data
 - alternate signal stack
 - error return value
 - scheduling policy/priority
 - Linux-specific
(e.g., CPU affinity)

thread state is small
versus
process state
(stored in PCB)

Thread Programming and Execution

- Every process is created with 1 “thread of execution”
 - Starts executing program at main()
- Additional threads are created using a threading programming library
 - A new “thread of execution” is created
 - Thread runs in the context of the process
 - Multiple threads can be created
- A process is “executing” when 1 or more of its threads are executing
- Suppose there are N threads and only 1 CPU
 - 1 thread can be running
 - what’s going on with the other $N-1$ threads?

Kernel Threads

- There is thread management support in the OS
 - Kernel operations for executing process threads
 - Thread “objects” in the kernel to hold thread context
 - Called *kernel threads*
- A process thread must have a kernel thread to run
 - Kernel thread acts like an “**execution container**”
- Nearly all OSes support a notion of threads
 - Linux -- thread and process abstractions are mixed
 - Mac OS X, Windows XP, ...

Something Subtle Going on Here (1)

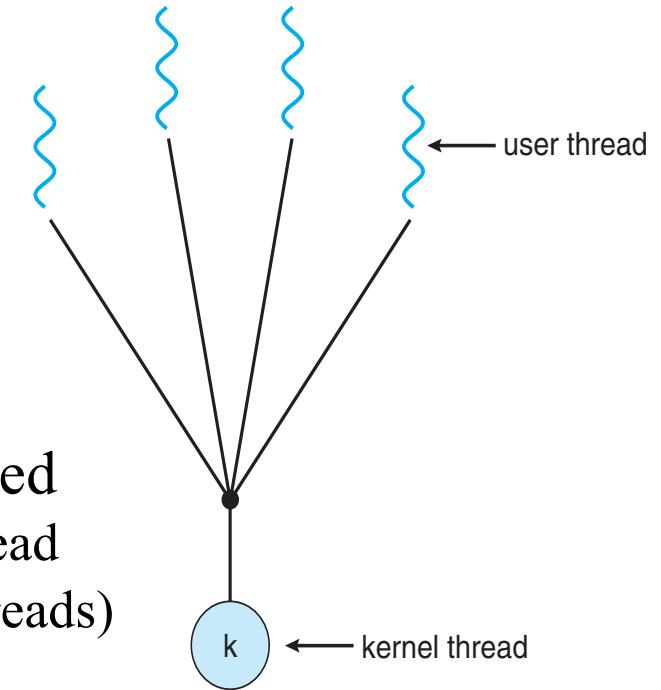
- A thread is just a (logical) unit of concurrency
 - It has a stack, a PC, a CPU state, and ... that's it!
 - It is a “thread of execution” in the process code
- A thread can “execute” when it is assigned to a “resource” which is then scheduled on a CPU
 - That resource is a **kernel thread**
- Where is the thread when it is not executing?
 - Two parts
 - ◆ (user) thread control block (TCB) + kernel thread control block (KTCB)
 - Condition 1: no kernel thread assigned
 - ◆ Only TCB is saved in the memory of the process!
 - ◆ KTCB for that user thread does not exist.
 - Condition 2: kernel thread is currently assigned
 - ◆ TCB is saved in the memory of the process, AND
 - ◆ KTCB is saved in kernel memory space

Something Subtle Going on Here (2)

- In essence, ALL threads are process threads!
 - Called *user-level threads* to distinguish from kernel threads
- Kernel threads are just containers for user-level threads in order for them to execute
- There can be *MANY* more user-level threads than actual kernel threads ... Why?
- They live “virtually” in the process until they are assigned a kernel thread and execute
 - User-level threads do not take up OS resources!
 - Only kernel threads do (in fact, they ARE the resources)
- Different models are used to assign user-level threads to kernel threads

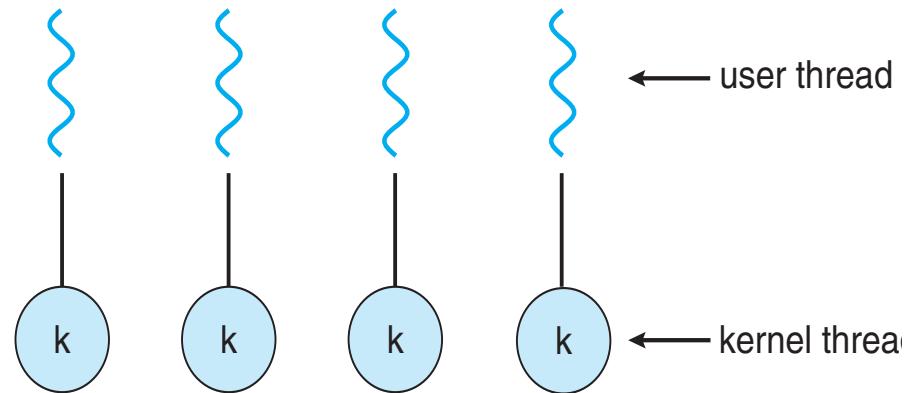
Many-to-One Thread Model

- Many user-level threads correspond to a *single* kernel thread
 - Kernel/OS is **not** aware of the mapping (OS only sees 1 thread)
 - Handled by a **user-level thread library**
- How does it work?
 - User-level thread library manages:
 - ◆ Create and execute a new thread
 - ◆ Maintain TCBs in process memory (user space)
 - ◆ Ready queue, waiting queues, etc.
 - ◆ Context switch between user-level threads
 - Upon yield, switch to another user thread in the same process
 - ◆ kernel is unaware
 - Upon wait (system calls), all threads are blocked
 - ◆ executing in kernel space by the single kernel thread
 - ◆ kernel is unaware there are other options (user threads)
 - ◆ cannot wait and run at the same time



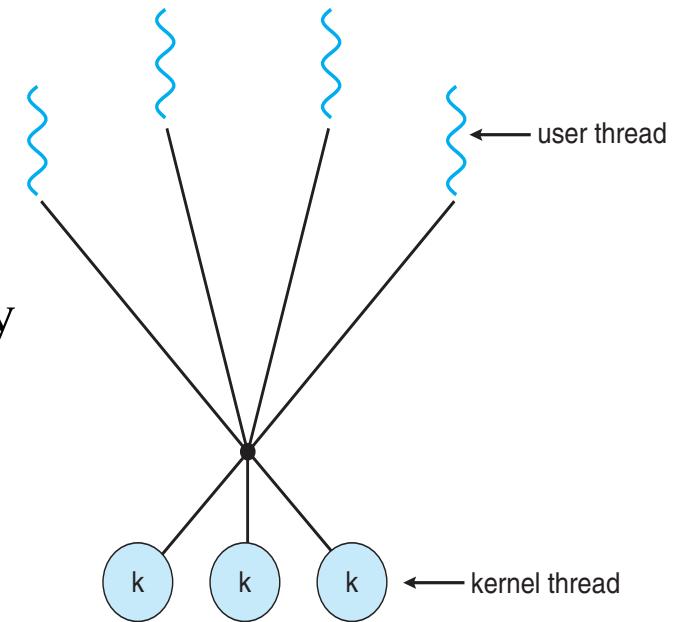
One-to-One Thread Model

- One user-level thread per kernel thread
 - A kernel thread is allocated for **every** user-level thread
 - Must get the kernel to allocate resources for **each** user-level thread
- How does it work?
 - User-level thread library manages:
 - ◆ system call to kernel to create threads
 - ◆ provide user APIs
 - Upon yield, switch to another kernel thread in system
 - ◆ kernel is aware and handle context switch entirely
 - Upon wait, another thread in the process may run
 - ◆ only the single kernel thread is blocked
 - ◆ kernel is aware there are other options in this process



Many-to-Many Thread Model

- A set of user-level threads maps to a set of kernel threads
 - Sizes can be different (kernel set is no larger)
 - No need for the kernel to allocate resources for each new user-level thread
- How does it work?
 - User-level thread library manages:
 - ◆ map user threads to kernel thread dynamically
 - ◆ TCB, queues (same as N:1)
 - Upon yield, switch to another thread
 - ◆ User-level thread library does context switch
 - Upon wait, another thread in the process may run
 - ◆ if a kernel thread is available, user-level can assign a ready user thread to it



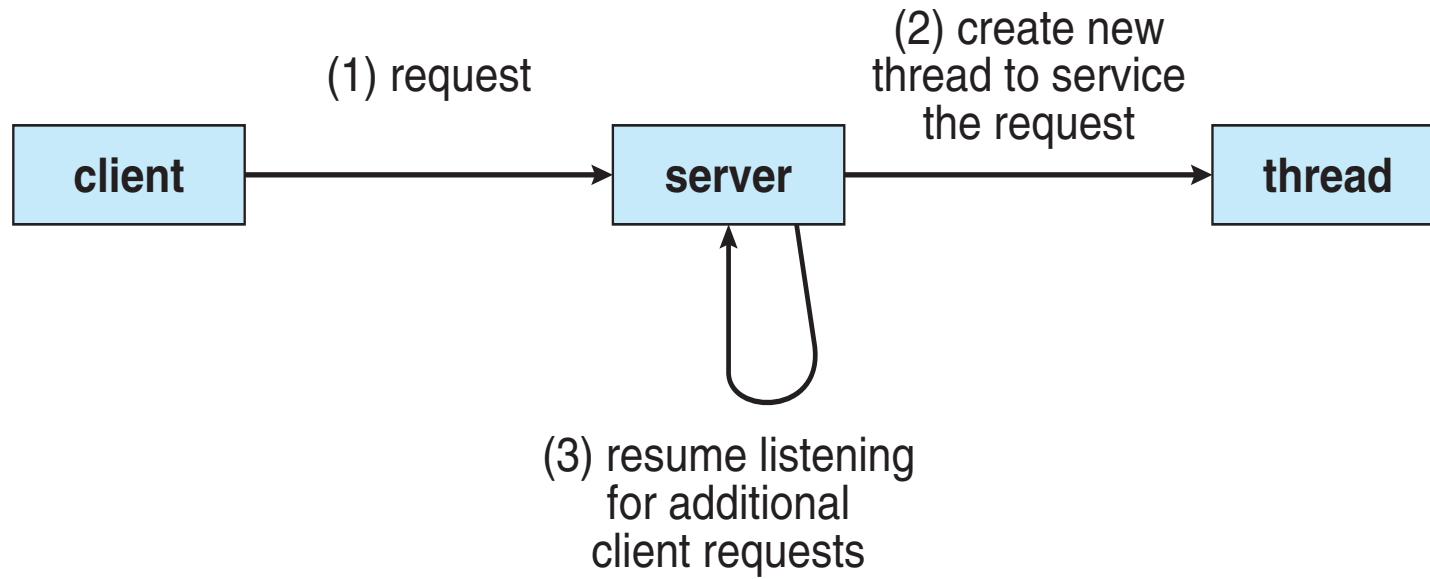
Thread Assignment

- How many kernel threads should a process have?
 - In an $M:N$ model there can be significantly more user-level threads (M) than kernel-level threads (N)
 - Kernel threads are the “real” threads that can be allocated to a CPU (core) and run
 - Want to keep enough kernel threads to satisfy the desired level of concurrency in a program and activity in the system
- Suppose that all kernel threads except 1 are blocked
 - What happens if the last kernel thread blocks?
 - Does it matter if there are more user threads “ready” to run?
- In multiprocessing, *thread affinity* is the notion of assigning a thread to run on a particular CPU (core)
 - Help to improve the thread’s performance by keeping its execution resources (CPU, cache, memory) local to CPU

Problems Solved with Threads

- Imagine you are building a web server
 - You could allocate a pool of threads, one for each client
 - ◆ thread would wait for a request, get content file, return it
 - How would the different thread models impact this?
- Imagine you are building a web browser
 - You could allocate a pool of threads
 - ◆ some for user interface
 - ◆ some for retrieving content
 - ◆ some for rendering content

Multithreaded Server Architecture



Linux Threads

- Linux uses a **one-to-one** thread model
 - Threads are called *tasks*
 - Limit (`cat /proc/sys/kernel/threads_max`)
- Linux views threads as “contexts of execution”
 - Threads are defined separately from processes
 - There is flexibility in what is private and shared
 - So, the line is blur in Linux implements
- Linux system call
 - `clone(int (*fn)(), void **stack, int flags, int argc, ...)`
 - Create a new thread (Linux task)
 - Start running at entry to `(*fn)()`
- May be created in the same address space or not
 - Flags (on means “share”): clone VM, clone filesystem, clone files, clone signal handlers
 - If all these flags off, what system call is clone equal to?

POSIX Threads

- POSIX Threads is a thread API specification
 - Does not define directly the implementation!
 - Could be implemented differently
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
 - Specification, not implementation
 - Provided either as user-level or kernel-level (very interesting)
 - API specifies behavior of the thread library
 - Implementation is up to development of the library
- Common in UNIX operating systems
 - Solaris, Linux, Mac OS X
- POSIX Threads is also known as *Pthreads*



POSIX Threads

- *pthread_create()*
 - start the thread
- *pthread_self()*
 - return thread ID
- *pthread_equal()*
 - for comparisons of thread ID's
- *pthread_exit()*
 - or just return from the start function
- *pthread_join()*
 - wait for another thread to terminate & retrieve value from *pthread_exit()*
- *pthread_cancel()*
 - terminate a thread, by TID
- *pthread_detach()*
 - thread is immune to join or cancel & runs independently until it terminates
- *pthread_attr_init()*
 - thread attribute modifiers

POSIX Threads: pthread_create()

- Interface
- *pthread_create(...)*
 - Thread ID {of the new thread}
 - ◆ set on return
 - Attributes {of the new thread}
 - ◆ stack size, scheduling information, etc.
 - Function {pointer}
 - ◆ start function for thread
 - Arguments {for function}
 - Return value, status {of the system call}

9

POSIX Threads FAQ

- How to pass multiple arguments to start a thread?
 - Build a struct and pass a pointer to it
- Is the pthreads ID unique to the system?
 - No, just process – Linux task ids are system-wide
- After *pthread_create()*, which thread is running?
 - It acts like fork in that both threads are running
- How many threads terminate when ...
 - *exit()* is called? – all in the process
 - *pthread_exit()* is called? – only the calling thread
- How are variables shared by threads?
 - Globals, local static, dynamic data (heap)

Pthreads Example

```
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of thread attributes */

    if (argc != 2) {
        fprintf(stderr,"usage: a.out <integer value>\n");
        return -1;
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
        return -1;
    }
}
```

Pthreads Example (Cont.)

```
/* get the default attributes */
pthread_attr_init(&attr);
/* create the thread */
pthread_create(&tid,&attr,runner,argv[1]);
/* wait for the thread to exit */
pthread_join(tid,NULL);

printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;

    for (i = 1; i <= upper; i++)
        sum += i;

    pthread_exit(0);
}
```

Code for Creating/Joining 10 Threads

```
#define NUM_THREADS 10
/* declare "worker" threads*/
pthread_t workers[NUM_THREADS]

. . .

/* create "worker" threads */
for (int i=0; i<NUM_THREADS; i++)
    pthread_create(&(workers[i]), &attr, runner, argv[1]);

. . .

/* join "worker" threads, one by one */
for (int i=0; i<NUM_THREADS; i++)
    pthread_join(workers[i], NULL);
```

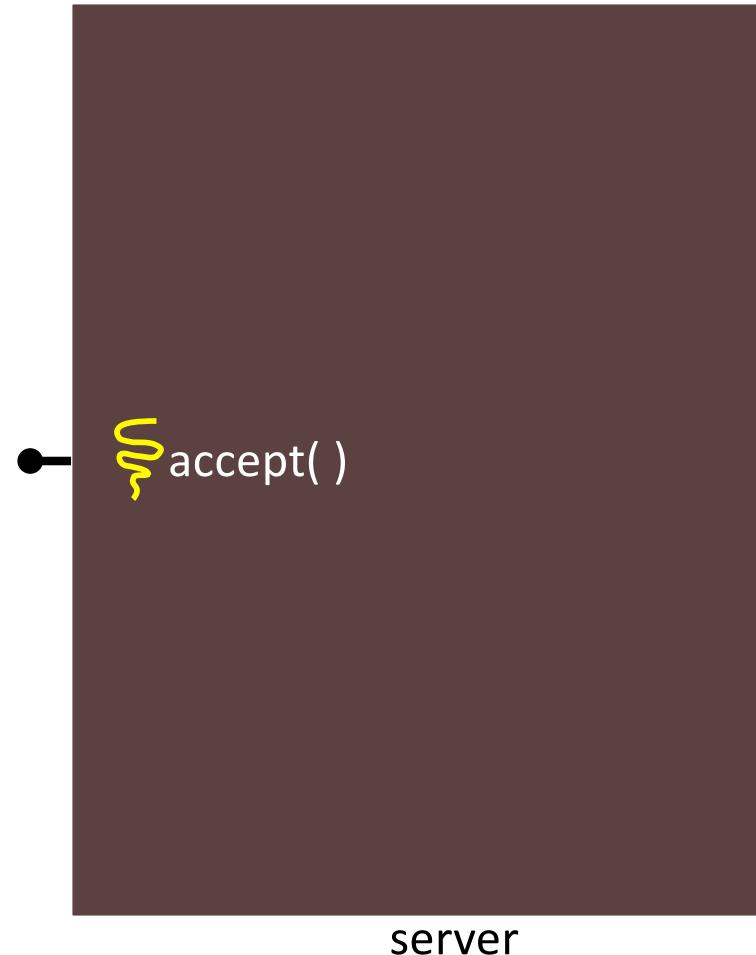


See code example
pthread-simple.c

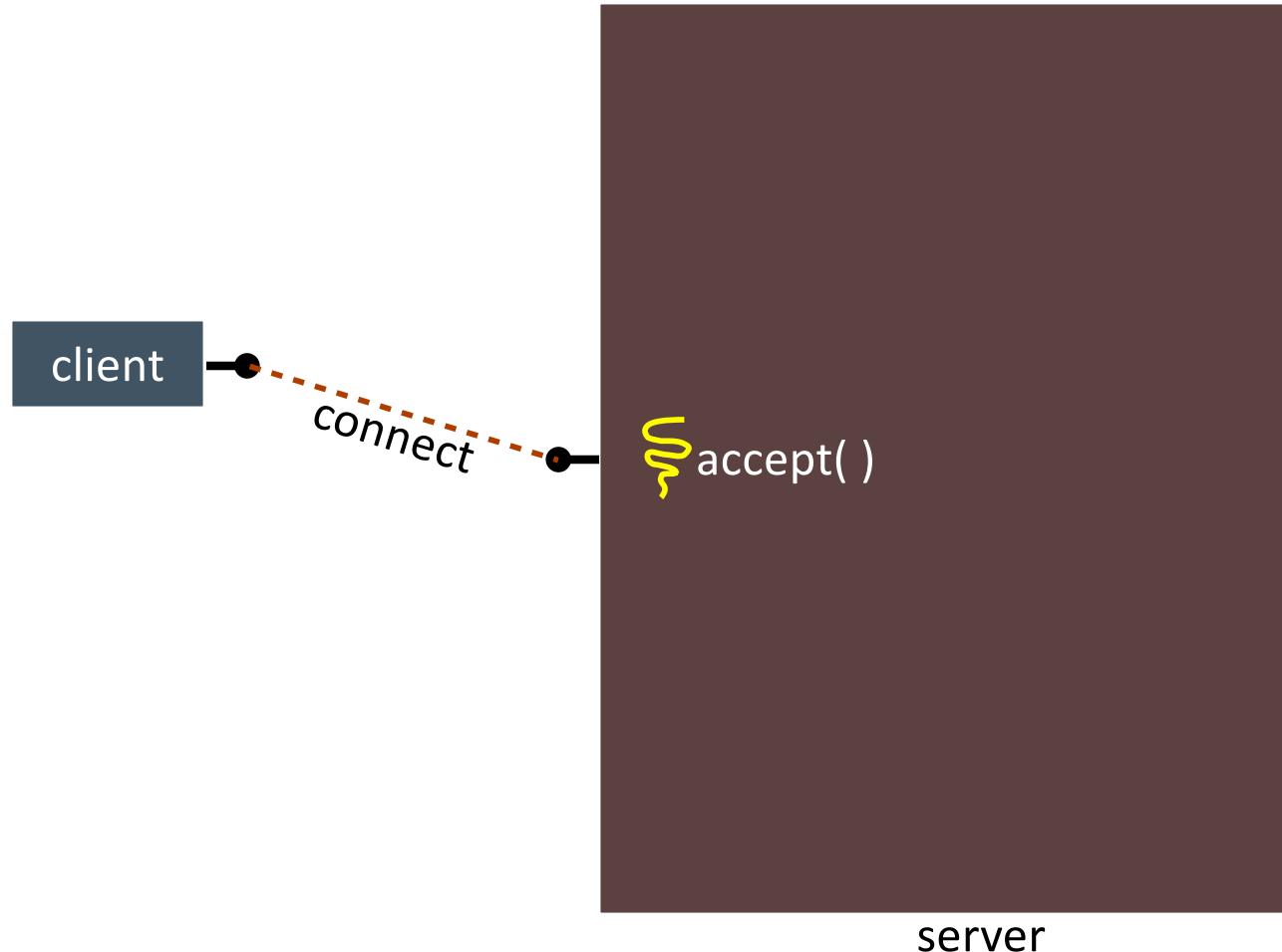
Concurrency with Threads

- Consider the client-server example again
- Now want to run with just a single process
 - Need to use threads to get concurrency
- Process “main” (parent) thread waits for clients
 - Parent thread forks (or dispatches) a new thread to handle each new client connection
 - Responsibilities of the child thread:
 - ◆ handles the new connection
 - ◆ exits when the connection terminates

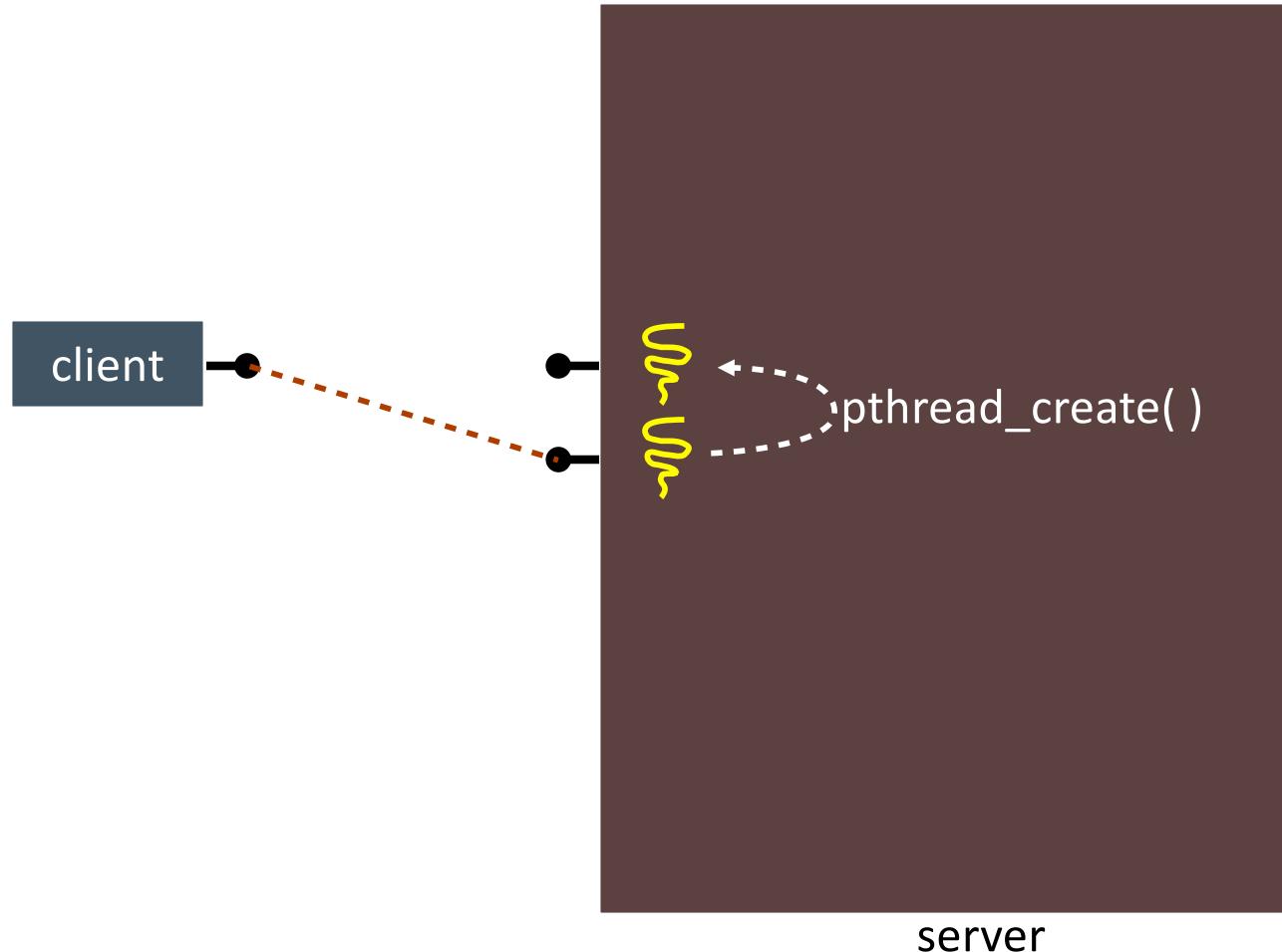
Client-Server with Pthreads (1)



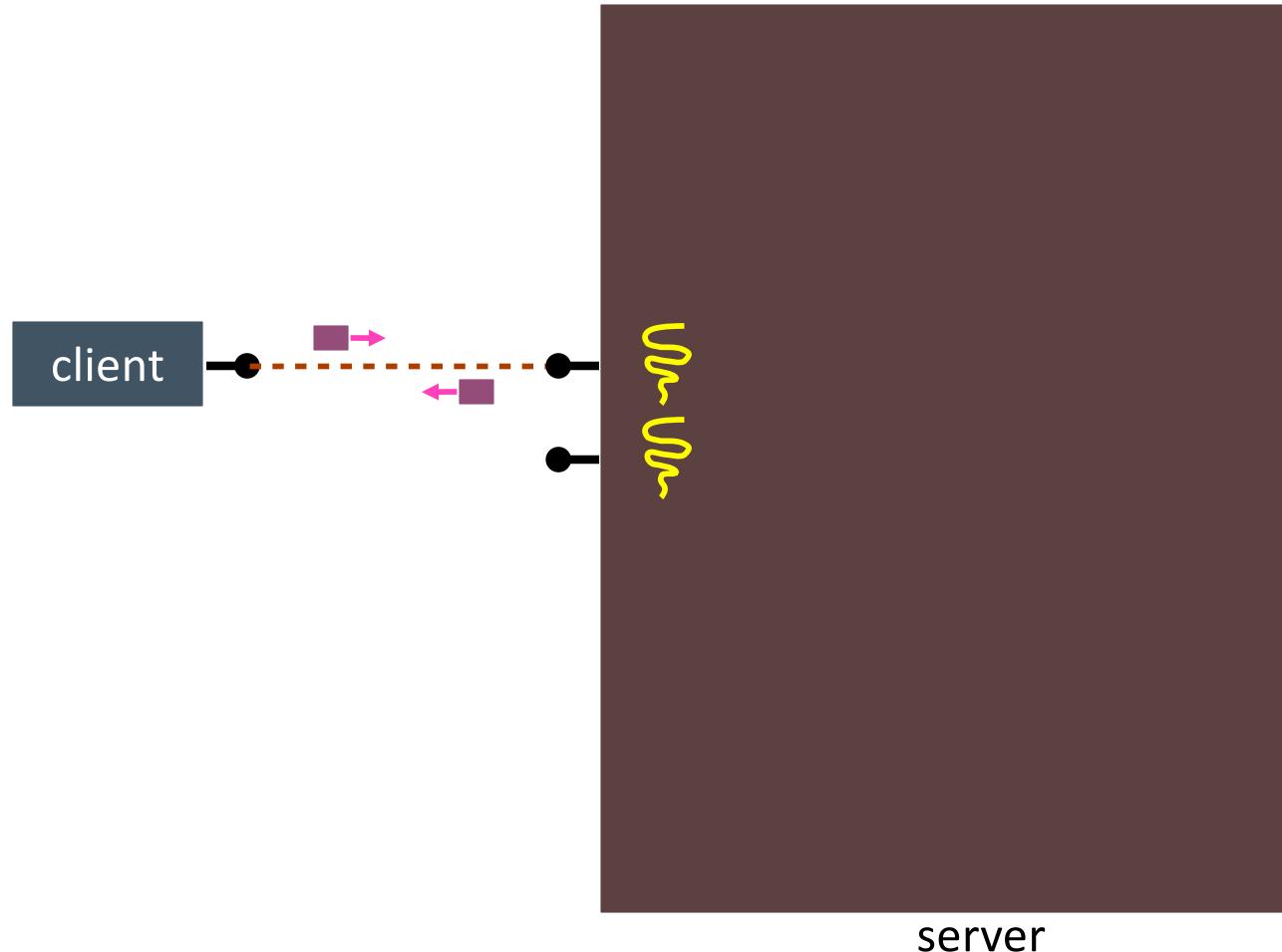
Client-Server with Pthreads (2)



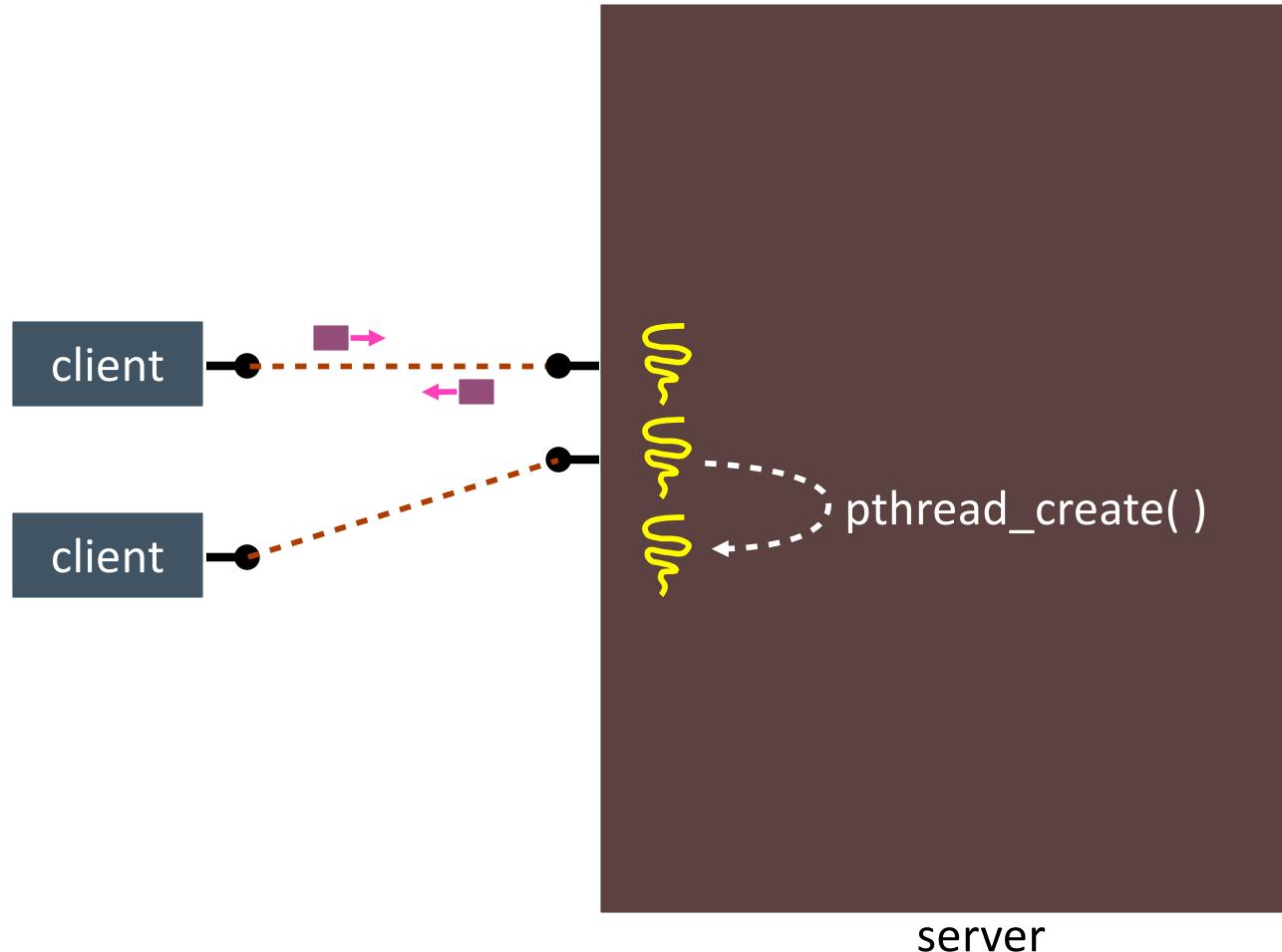
Client-Server with Pthreads (3)



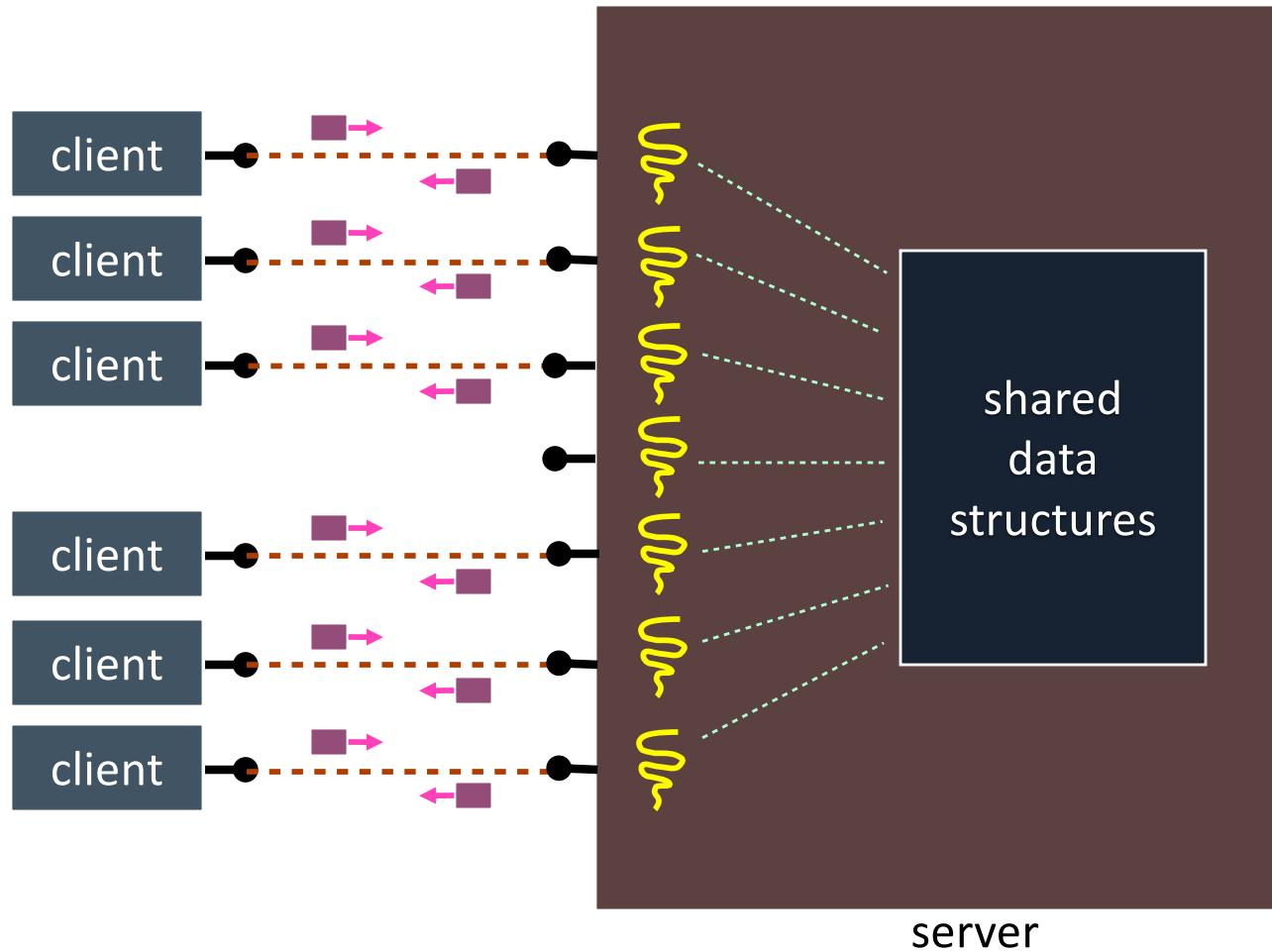
Client-Server with Pthreads (4)



Client-Server with Pthreads (5)



Client-Server with Pthreads (5)



How is this different
from processes?

Implications?

- Consider a web server on a Linux platform
- 0.0297 ms per thread create (time for *clone()*)
 - 10x faster than process forking
 - Maximum of $(1000 / 0.0297) = \sim 33,670$ connections/sec
 - 3 billion connections per day per machine
 - ◆ much, much better
- Facebook would need only 500 machines
- So, why do we need processes at all?
- Just write everything using threads
 - Writing safe multithreaded code can be complicated
 - Why? What are the issues?

Contrasting Processes and Threads

- A process MUST have all necessary information for its execution stored in the OS (process context)
 - Stored in the PCB and takes space in the OS memory
- Each process has a default “thread of execution”
- All processes can execute independently
- A thread is part of (lives in) a process
- All threads in a process share the process context
 - Each thread has a context: PC, stack, CPU state
- All threads are “user” threads in the sense that they have their own PC and a stack
 - This requires very little storage
- For a thread to execute, it must be assigned to a “kernel” thread that will be managed by the OS
 - It can then be context switched (thread switch)

Compare Overhead of Processes and Threads

- Write a program to create N processes or threads

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

pid_t pid;
int status;

int main(int argc, char *argv[])
{
    int i, numprocesses;

    if(argc != 2) {
        fprintf(stderr, "USAGE: processes <INT>\n");
        exit(1);
    }

    numprocesses = atoi(argv[1]);           // # child processss to create

    for (i=0; i<numprocesses; i++) {
        pid = fork();
        if (pid == 0)                  // child process
            exit(0);
        else
            waitpid(pid, &status, 0);
    }
    return 0;
}
```

Processes

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <sys/time.h>
#include <unistd.h>

struct threadargs {
    int id;
};

struct threadargs      threadARGS;

pthread_t tid;           // thread ID for publishers
pthread_attr_t attr;     // thread attributes
void **retval;

void *routine(void *param);

void *routine(void *args) {
    int tid;           // thread id
    tid = ((struct threadargs *) args)->id;

} // routine

int main(int argc, char *argv[])
{
    int i, numthreads;

    if(argc != 2) {
        fprintf(stderr, "USAGE: threads <INT>\n");
        exit(1);
    }

    numthreads = atoi(argv[1]);           // # child threads to create

    for(i=0; i<numthreads; i++) {
        tid = i;
        pthread_create(&tid, &attr, routine, (void *) &threadARGS);
        pthread_join(tid, retval);
    }
} // main()
```

Threads

- Time for different N

★ See program example

Processes versus Threads Performance

```
ix 95% foreach i (1000 10000 100000 1000000)
foreach? echo Time the creation and shutdown of $i processes
foreach? time processes $i
foreach? echo Time the creation and shutdown of $i threads
foreach? time threads $i
foreach? end

Time the creation and shutdown of 1000 processes
0.209u 0.145s 0:00.33 103.0%    0+0k 46+0io 0pf+0w
Time the creation and shutdown of 1000 threads
0.000u 0.090s 0:00.08 112.5%    0+0k 48+0io 0pf+0w

Time the creation and shutdown of 10000 processes
2.233u 1.452s 0:03.49 105.4%    0+0k 6+0io 0pf+0w
Time the creation and shutdown of 10000 threads
0.086u 0.684s 0:00.67 113.4%    0+0k 8+0io 0pf+0w

Time the creation and shutdown of 100000 processes
22.277u 14.535s 0:35.22 104.4%  0+0k 6+0io 0pf+0w
Time the creation and shutdown of 100000 threads
0.633u 7.107s 0:06.85 112.8%    0+0k 8+0io 0pf+0w

Time the creation and shutdown of 1000000 processes
223.840u 144.968s 5:50.76 105.1%   0+0k 6+0io 0pf+0w
Time the creation and shutdown of 1000000 threads
6.392u 71.093s 1:08.34 113.3%    0+0k 8+0io 0pf+0w
```

N	Processes	Threads
1K	0.33 sec	0.08 sec
10K	3.49 sec	0.67 sec
100K	35.22 sec	6.85 sec
1M	5:50.75 sec	1:08.34 sec

Concurrent Threads

- All threads of a process are concurrent
- Benefits
 - All threads are running the same code (of the process)
 - Threads interact directly through shared memory
 - Lower overhead than processes
 - ◆ thread switching is faster
 - ◆ better utilization of the processor, especially with multiple cores
- Disadvantages
 - Maintaining shared memory consistency
 - ◆ synchronization is complicated (see later lecture)
 - One errant thread can affect the whole process
- More to come on this topic ...

Thread Pools

- Pool of threads
 - Create (all threads) at initialization time
 - Assign tasks to run to a waiting thread
 - ◆ it is already made so it should be fast
 - Use all available threads if enough tasks
- What about when a task is done?
 - Should we use terminate the thread executing it?
 - Suppose there is another task to run
 - Should we create another thread for this task?
- Concern for the time to create another thread
 - It is faster than setting up a process
 - But is it necessary?
- How could we improve performance?
 - Just have the current thread wait in a pool



Thread Interaction and Communication

- Can you use shared memory?
 - Sure, it is there, might as well use it
 - Just need to allocate memory in the address space
 - ◆ no need for fancy IPC shared memory
- Can you use message passing between threads?
 - Of course, there is nothing to prevent it
 - Would have to build messaging infrastructure to do it
 - ◆ likely would implement in shared memory
 - ◆ not really necessary to use OS mechanisms
- Hmm, can threads utilize IPC mechanisms
 - Certainly, if they were in different processes
 - Would need to make sure only 1 thread uses at a time to avoid consistency problems
 - Strange to use IPC for threads within the same process

Thread Cancellation

- So, you want to stop a thread from executing
 - Do not need it anymore
 - It is just hanging around and you want to get rid of it
- Two choices
 - Synchronous (deferred) cancellation
 - ◆ wait for the thread to reach a point where cancellation is permitted
 - ◆ no such operation in Pthreads, but can create your own
 - Asynchronous cancellation
 - ◆ terminate it now
 - ◆ *pthread_cancel(thread_id)*

Scheduling

- How many kernel threads to create for a process?
 - In M:N model
- If last kernel thread for an application is to be blocked
 - What happens?
 - Recall the relationship between kernel and user threads
- It would be nice if the kernel told the application and the application had a way to get more kernel threads
 - Scheduler activation
 - ◆ at thread block, the kernel tells the application
 - Application can then get a new thread created
 - ◆ see lightweight threads

Scheduler Activation

- It would be nice if the kernel told the application that a kernel thread was blocking
 - *Scheduler activation*
 - ◆ at thread block, the kernel tells the application via an *upcall*
 - ◆ an *upcall* is a general term for an invocation of application function from the kernel
 - User-level thread scheduler can then get a new user-level thread created
- Way of conveying information between the kernel and the user-level thread scheduler regarding the disposition of:
 - # user-level threads (increase or decrease)
 - User-level thread state
 - ◆ running to waiting, waiting to ready

Fork/Exec Issues

- Semantics are ambiguous for multithreaded processes
- *fork()*
 - How does it interact with threads?
 - Child process created with only 1 thread?
- *exec()*
 - What happens if process has more than 1 thread?
- *fork* then *exec*
 - Should all threads be copied?



Re-entrance and Thread-Safety

- *Re-entrant* code
 - Code that can be run by multiple threads concurrently
 - Can also be safely interrupted and called again (“re-entered”) before the previous execution is finished
- *Thread-safe* libraries
 - Library code that permits multiple threads to invoke the safe function concurrently
 - Mainly concerned with variables that should be private to individual threads
 - Requires moving some global variables to local variables
- Requirements
 - Rely only on input data to the function or data that is already specific to a thread
 - ◆ can not (should not) use any shared data (e.g., global variables) that might be modified by a thread executing the function
 - Must be careful about locking (see later lectures)

Signal Handling

- What's a signal?
 - A form of IPC
 - Send a particular signal to another process
- Receiver's signal handler processes signal on receipt
- Example
 - Tell the Internet daemon (*inetd*) to reread its config file
 - Send signal to inetd: *kill -SIGHUP <pid>*
 - inetd's signal handler for the SIGHUP signal re-reads the config file
- Note: some signals cannot be handled by the receiving process, so they cause default action (kill the process)



Signal Handling

- Synchronous signals
 - Generated by the kernel for the process
 - Due to an exception -- divide by 0
 - ◆ events caused by the thread receiving the signal
- Asynchronous signals
 - Generated by another process
- Asynchronous signals are more difficult for multithreading

Signal Handling and Threads

- So, you send a signal to a process
 - Which thread should it be delivered to?
- Choices
 - Thread to which the signal applies
 - Every thread in the process
 - Certain threads in the process
 - A specific signal receiving thread
 - It depends...
- UNIX signal model created decades before Pthreads standard
 - Conflicts arise as a result

Why not threads?

- Threads can interfere with one another
 - Need to be careful to maintain consistency
 - Impact of more threads on memory and caches
 - Bug in one thread can cause process problems
- Executing multiple threads may slow them down
 - Impact of single thread vs. switching among threads
- Harder to program a multi-threaded program
 - Multitasking hides context switching
 - Multi-threading introduces concurrency issues

Summary of Threads

- Threads
 - A mechanism to improve performance and CPU utilization
- Kernel-space and user-space threads
 - Kernel threads are real, schedulable threads
 - User-space may define its own threads (but not real)
- Threading models and implications
 - Programming systems
 - Multi-threaded design issues
- Threads are useful, but not always the right solution
 - Could slow down system in some cases
 - Can be difficult to program
- Multiprogramming and multithreading are important concepts in modern operating systems

Next Class

□ Scheduling