CIS 415: Operating Systems

Threads

Spring 2012
Prof. Kevin Butler

- Last class:
  - Processes
- Today:
  - Threads

Why Threads?

- Think back to processes: "a program in execution"
  - memory address space containing code and data
  - other resources (e.g., open file descriptors)
  - state information (PC, register, SP) => PCB details
- Consider as *two* categories
  - *collection of resources* (code, addr space, open files, etc)
  - *thread of execution* (current state operating on resource)
- Can think about separately

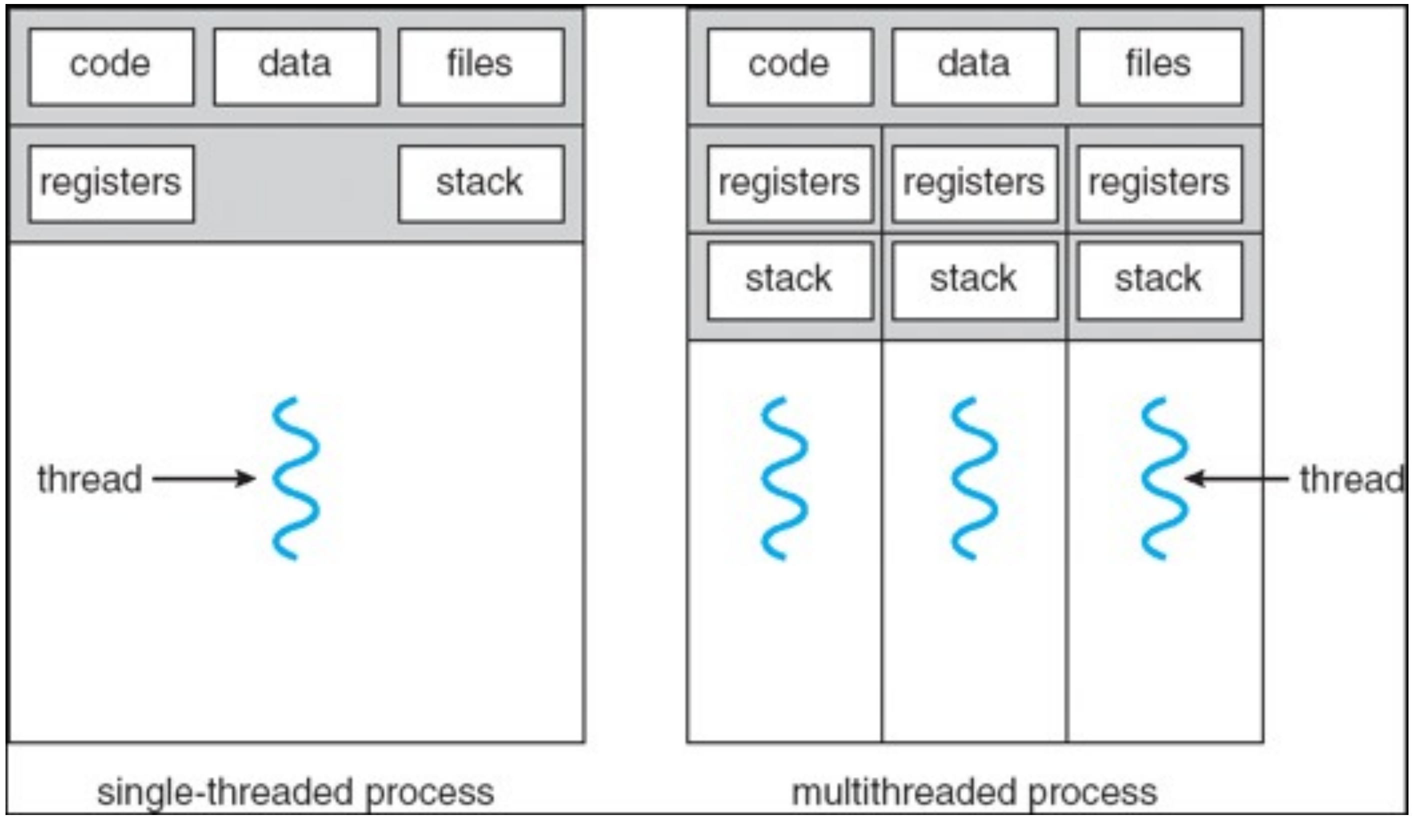
Process Model

- Recall from last day: much of OS job is keeping processes from interfering with each other
  - thread of execution associated with own resources
  - can't write over process address space
- Good for isolation, bad because of context switching required for changing threads
  - full process swap required, OS intervention, all the state involved in a context switch (what is involved?)
  - some apps could contain multiple threads of execution but only need one grouping of resources

Advantages of Threads

- Improve Responsiveness
    - Ideally, a thread is always ready
- Resource Sharing
    - All the stuff is easily accessible
- Economy of Resources
    - Thread resources are cheaper than process resources
- Utilization of Multiprocessors
    - Get all of them running

Multi-Threaded vs. Single-Threaded



Regular UNIX process can be thought of as a special case of a multithreaded process: a process that contains just one thread

Terminology

- Multiprogramming
  - Run multiple processes concurrently on a single processor
  - OS choose which process to run out of multiple
- Multiprocessing
  - Run multiple processes on multiple processors
  - OS manages mapping of processes to processors
- Multithreading
  - Define multiple execution contexts in a single address space
  - OS manages mapping of contexts (threads) to an address space
  - OS manages mapping of threads to processor(s)

Multithreaded Applications

- Multiple threads sharing a common address space
  - applications that:
    - need to share data structures among threads
    - don't need the OS to enforce resource separation (trust amongst the threads)
  - not for arbitrary code or general programs
- What are examples of multi-threaded applications?

What's a Thread?

UNIVERSITY
OF OREGON

- Thread of Execution on CPU
  - Program counter
  - Registers
- Memory
  - Address space (process)
  - Stack -- per thread
- I/O
  - Share files, sockets, etc. (process)

Working with Threads

- In a C program
  - `main()` procedure defines the first thread
  - C programs always start at `main`
- Create a second thread
  - Allocate resources to maintain a second execution context in same address space
    - Think about what process fields will be necessary for a thread
  - Supply a procedure name to start the new thread’s execution

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

Which is Cheaper?

- Create new process or create new thread (in existing process)
- Context switch between processes or threads
- Interprocess or inter-thread communication
- Sharing memory between processes or threads
- Terminate a process or terminate a thread (not last one)

Process creation method	Time (sec), elapsed (real)
fork()	22.27 (7.99)
vfork()	3.52 (2.49)
clone()	2.97 (2.14)

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

Implications?

- **0.22 ms** per fork
  - maximum of (1000 / 0.22) = 4545.5 connections per second
  - 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 billion) page views per day
      - guess ~1-20 HTTP connections per page
      - would need 3,000 -- 60,000 machines just to handle fork( ), i.e., without doing any work for each connection!

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
  - stack
  - signal mask
  - thread-specific data
  - alternate signal stack
  - error return value
  - scheduling policy/priority
  - Linux-specific (e.g., CPU affinity)

Threading Models

- Programming: *Library or system call interface*
    - User-Space Threading
        - Thread management support in user-space library
        - Linked into your program
    - Kernel Threading
        - Thread management support in the kernel
        - Invoked via system call
- Scheduling: *Application or kernel scheduling*
    - May create user-level or kernel-level threads
        - NOTE: CPU only runs kernel threads!

User-Space Threads

- Thread management support in user-space library
  - Sets of functions for creating, invoking, and switching among threads
- Linked into your program
  - Thread libraries
- Examples
  - POSIX Threads (PThreads)
  - Win32 Threads
  - Java Threads

Implementing Threading

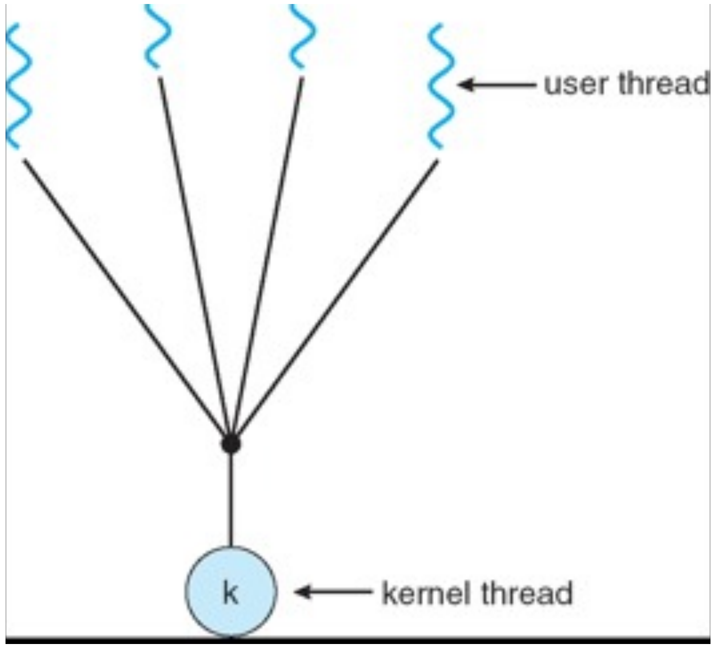
- Threads can perform operations in user mode that are usually handled by the OS
    - assumes cooperating threads so hardware enforcement of separation not required
- Idea: "dispatcher" subroutine in the process is called when a thread is ready to relinquish control to another thread
    - manages stack pointer, program counter
    - switches process's internal state among threads

Kernel Threads

- Thread management support in kernel
  - Sets of system calls for creating, invoking, and switching among threads
- Supported and managed directly by the OS
  - Thread objects in the kernel
- Nearly all OSes support a notion of threads
  - Linux -- thread and process abstractions are mixed
  - Solaris
  - Mac OS X
  - Windows XP
  - …

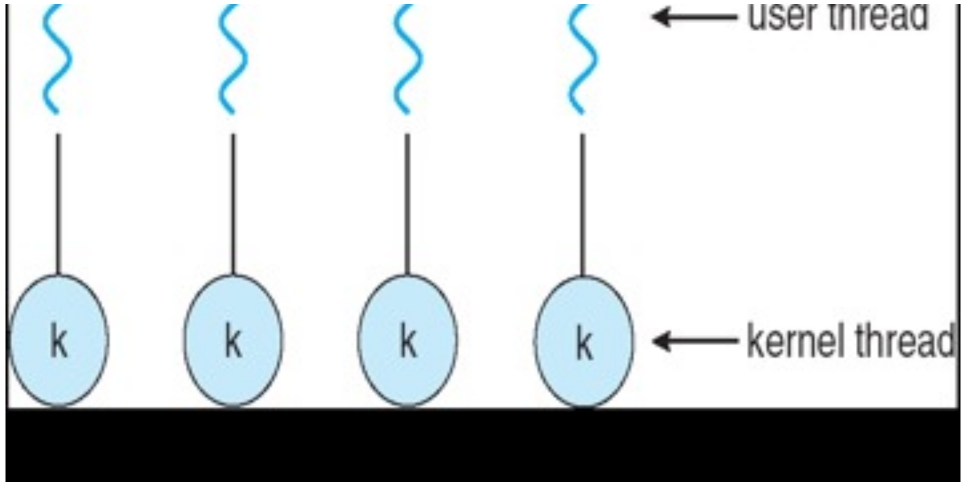
Many-to-one Thread Model

- Many user-level threads correspond to a single kernel thread
  - Kernel is not aware of the mapping
  - Handled by a thread library
- How does it work?
  - Create and execute a new thread
  - Upon yield, switch to another thread in the same process
    - Kernel is unaware
  - Upon wait, all threads are blocked
    - Kernel is unaware there are other options
    - Can't wait and run at the same time



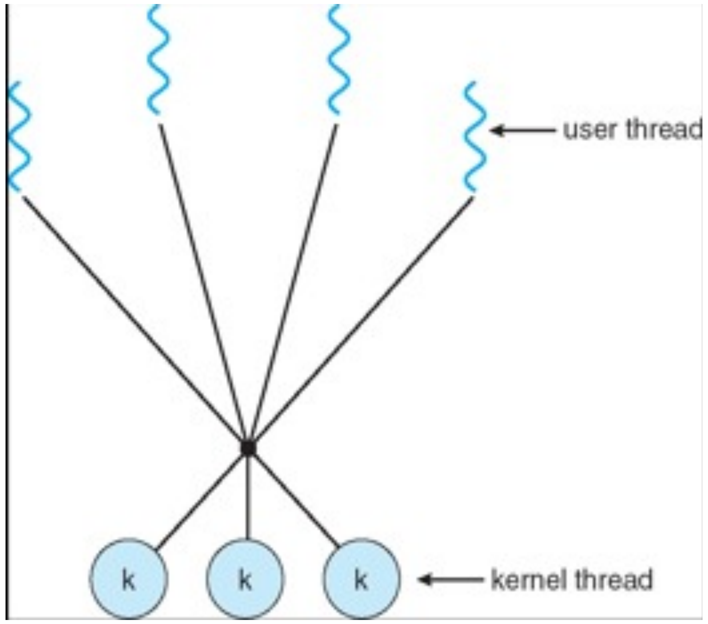
One-to-one Thread Model

- 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 new user-level thread
- How does it work?
  - Create new thread, including system call to kernel
  - Upon yield, switch to another thread in system
    - Kernel is aware
  - 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 pool of user-level threads maps to a pool of kernel threads
  - Pool sizes can be different (kernel pool is no larger)
  - A kernel thread is pool is allocated for every user-level thread
  - No need for the kernel to allocate resources for each new user-level thread
- How does it work?
  - Create new thread (may map to kernel thread dynamically)
  - Upon yield, switch to another thread in system
    - Kernel is aware
  - Upon wait, another thread in the process may run
    - If a kernel thread is available to be scheduled to that process
    - Kernel is aware of the mapping between process threads and kernel threads

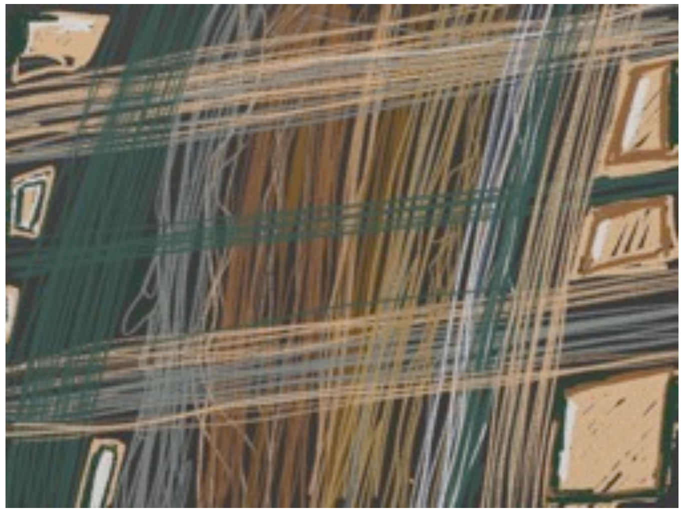


Problems solved with threads

- 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
  - What happens if the user decided to stop the request?
    - Mouse click on the stop button

Linux Threads

- Linux uses a one-to-one thread model
  - Threads are calls tasks
- Linux views threads are "contexts of execution"
  - Threads are defined separately from processes
  - I.e., a thread is assigned an address space



Linux Threads

- Linux system call
  - `clone(int (*fn)(), void **stack, int flags, int argc, … /*args */)`
  - Create a new thread (Linux task)
- May be created in the same address space or not
  - Flags: Clone VM, Clone Filesystem, Clone Files, Clone Signal Handlers
    - If `clone` with all these flags off, what system call is `clone` equal to?

POSIX Threads

- POSIX Threads or Pthreads is a thread API specification
  - Not directly an implementation
  - Could be mapped to libraries or system calls
- Supported by Solaris and Linux

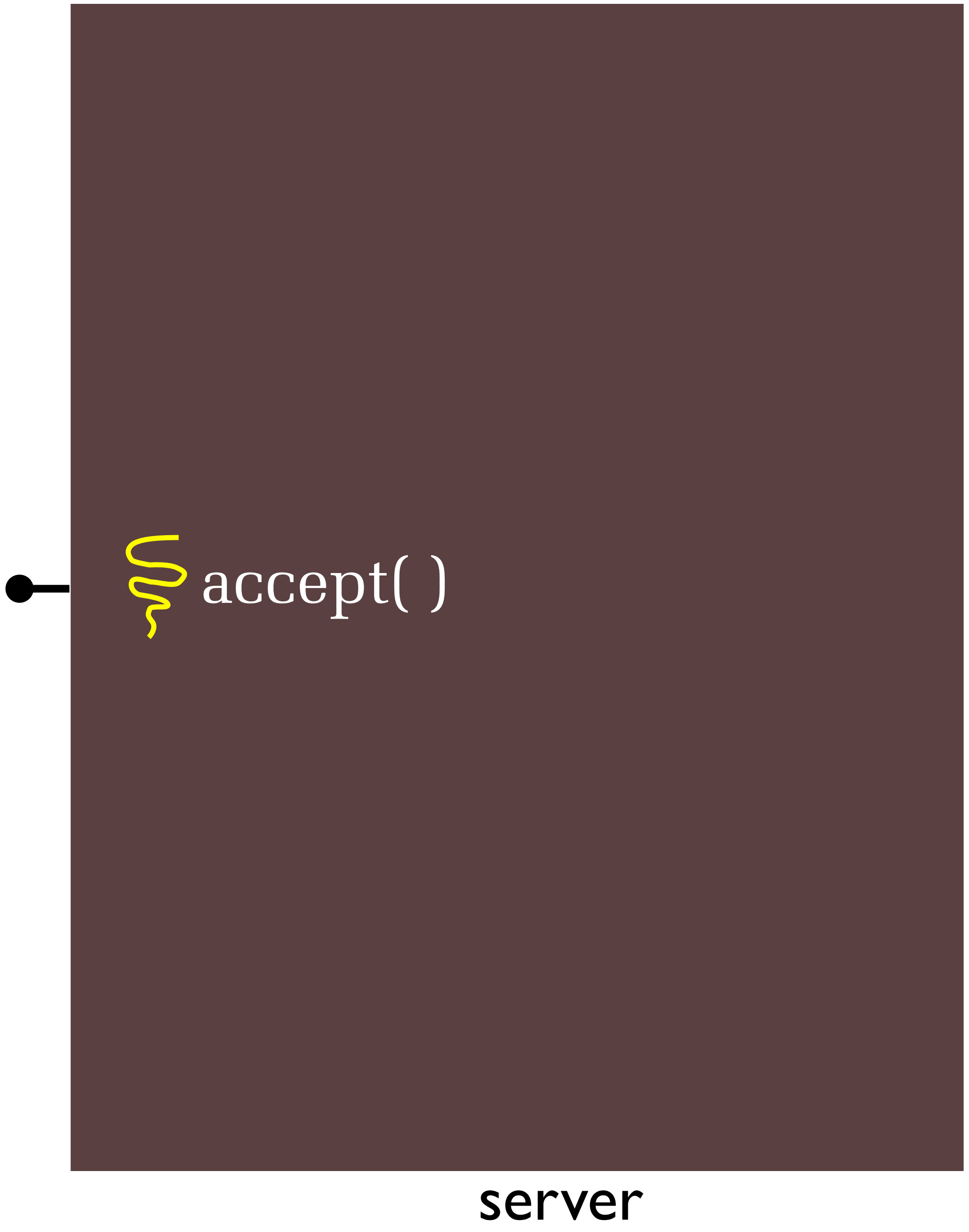
POSIX Threads

- `phtread_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

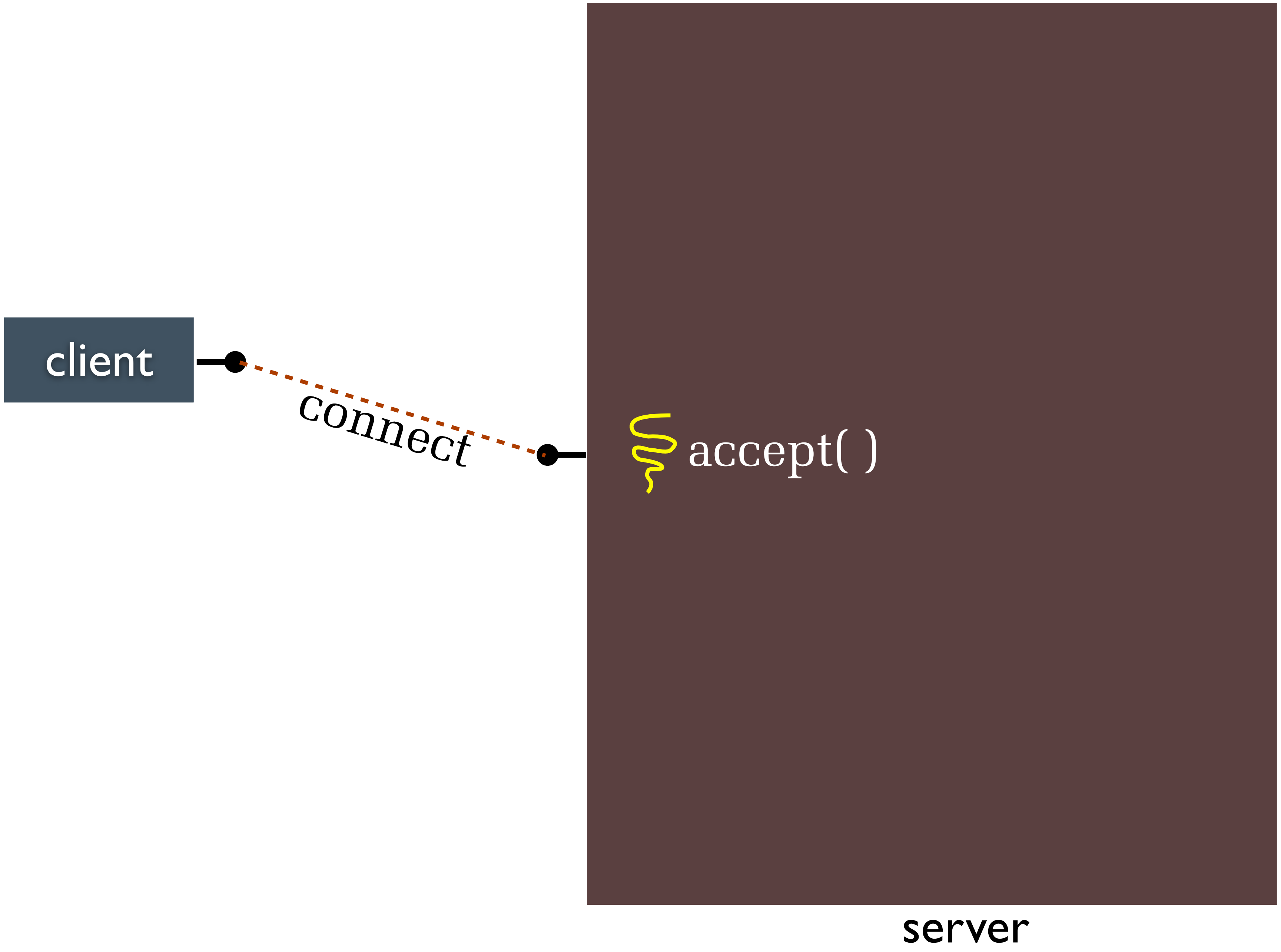
Concurrency with threads

- A single ***process*** handles all of the connections
  - but, a parent ***thread*** forks (or dispatches) a new thread to handle each connection
  - the child thread:
    - handles the new connection
    - exits when the connection terminates

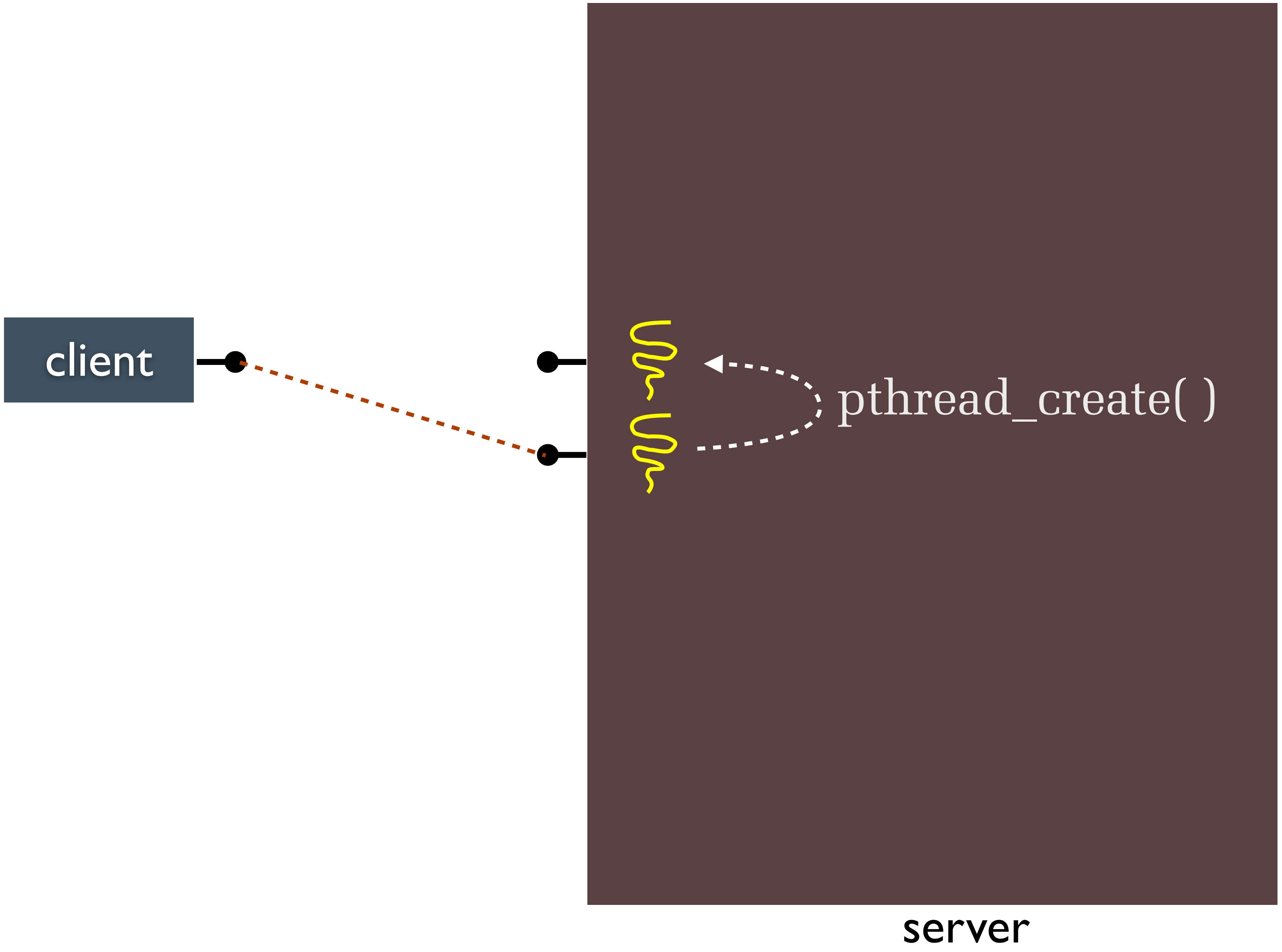
Graphically



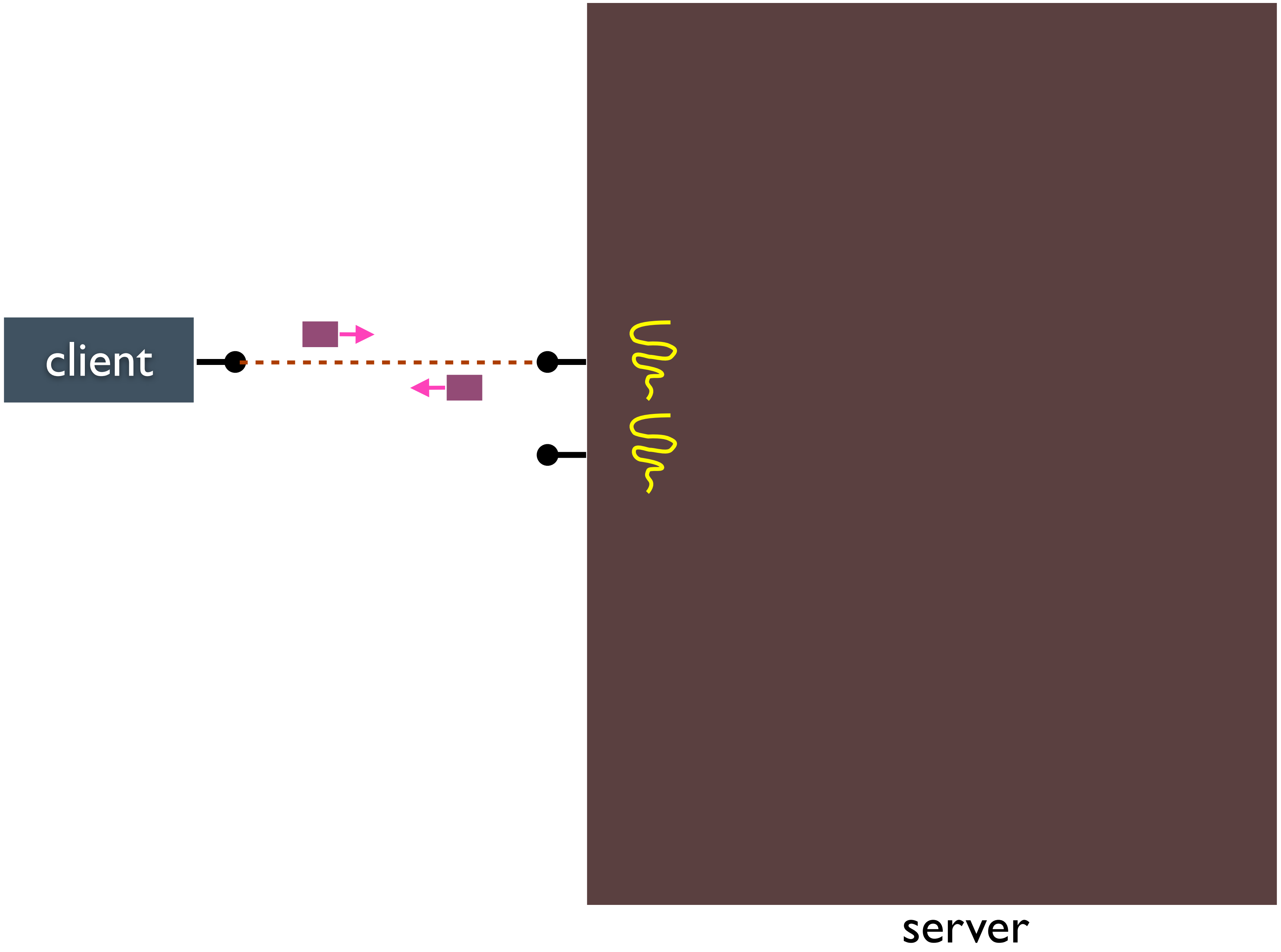
Graphically



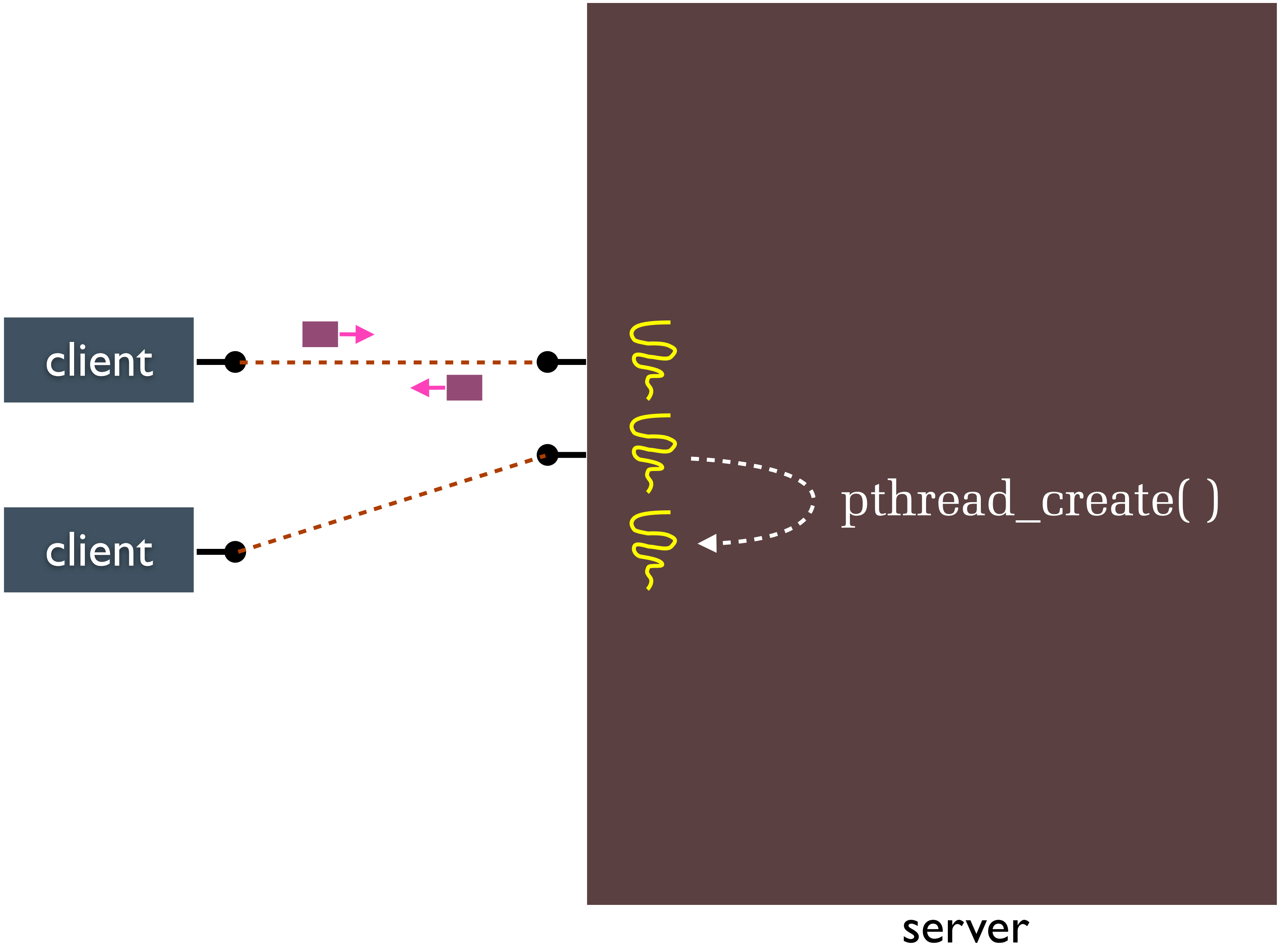
Graphically



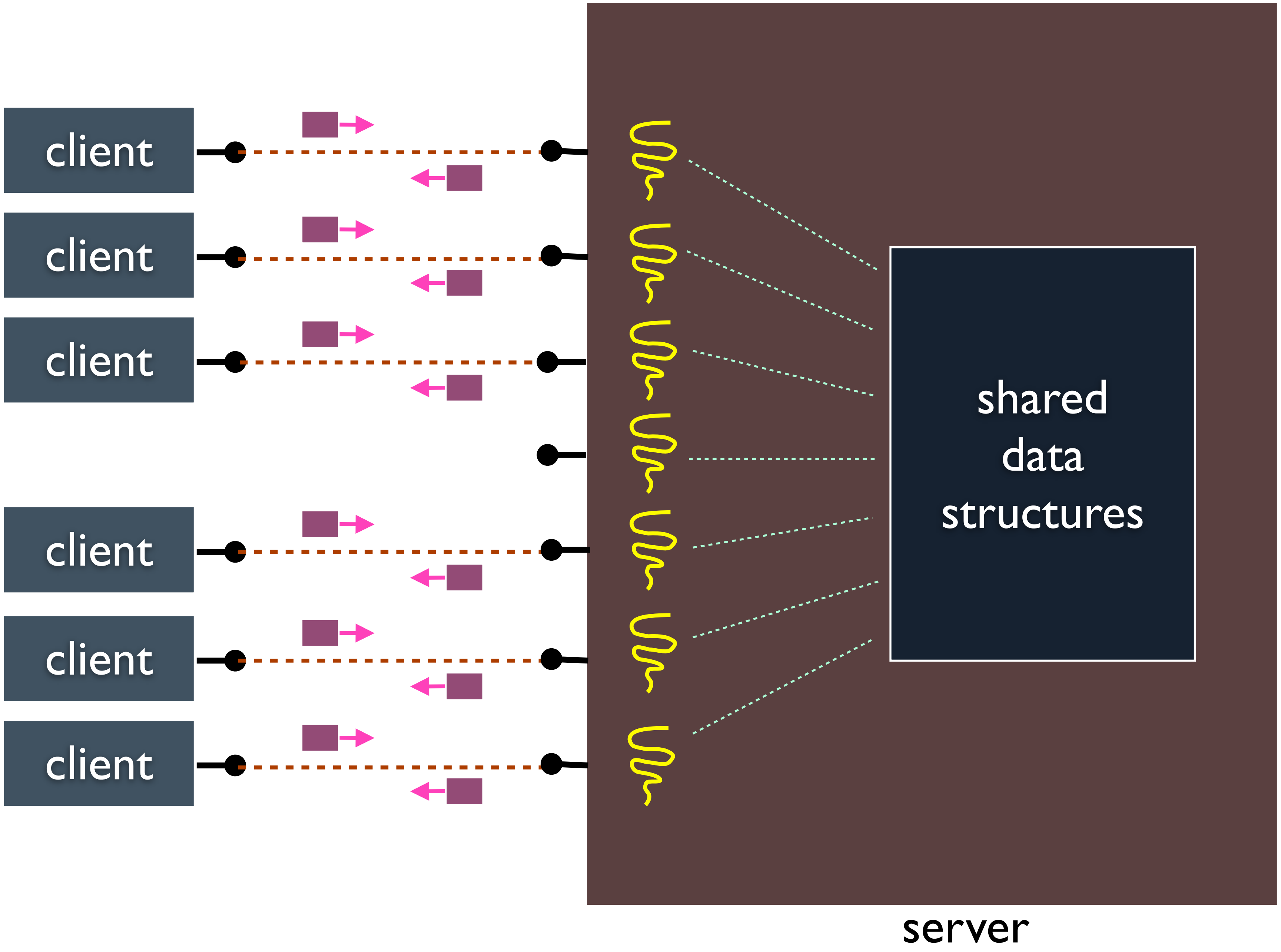
Graphically



Graphically



Graphically



Implications?

- **0.0297 ms** per thread create; 10x faster than process forking
  - maximum of (1000 / 0.0297) = ~33,670 connections per second
  - 3 billion connections per day per machine
    - much, much better

- But, writing safe multithreaded code can be complicated

Concurrent threads

- Benefits
    - straight-line code, line processes or sequential
        - still the case that much of the code is identical!
    - parallel execution; good CPU, network utilization
        - lower overhead than processes
    - shared-memory communication is possible
- Disadvantages
    - synchronization is complicated
    - shared fate within a process; one rogue thread can hurt you badly

Inter-Thread Communication

- Can you use shared memory?
  - Already have it
  - Just need to allocate memory in the address space
    - No need for shm
    - Programming to pipes provides abstraction
- Can you use message passing?
  - Sure
  - Would have to build infrastructure

Thread Cancellation

- So, you want to stop a thread from executing
  - Don’t need it anymore
    - Remember the browser ‘stop’ example

- Two choices
  - Synchronous 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)

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
  - E.g., 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…

Signal Handling and Threads

- UNIX signal model created decades before Pthreads: conflicts arise
- Synchronous vs. Asynchronous Cases
- Synchronous
  - Signal is delivered to the same process that caused the signal
  - Which thread(s) would you deliver the signal to?
- Asynchonous
  - Signal generated by another process
  - Which thread(s) in this case?

Thread Pools

- Problem: setup time
- Faster than setting up a process, but what is necessary?
    - How do we improve performance?

Thread Pools

UNIVERSITY
OF OREGON

- Pool of threads
  - Create (all) at initialization time
  - Assign task to a waiting thread
    - It's already made
  - Use all available threads
- What about when that task is done?
  - Suppose another request is in the queue…
  - Should we use running thread or another thread?

Reentrance and Thread-Safety

- Terms that you might hear
- *Reentrant Code*
  - Code that can be run by multiple threads concurrently
- *Thread-safe Libraries*
  - Library code that permits multiple threads to invoke the safe function
- Requirements
  - Rely only on input data
    - Or some thread-specific data
  - Must be careful about locking (later)

Why not threads?

- Threads can interfere with one another
  - Impact of more threads on caches
  - Impact of more threads on TLB
  - Bug in one thread...
- Executing multiple threads may slow them down
  - Impact of single thread vs. switching among threads
- Harder to program a multithreaded program
  - Multitasking hides context switching
  - Multithreading introduces concurrency issues

Summary of Threads

- Threads
    - Programming systems
    - Multi-threaded design issues
- Useful, but not a panacea
    - Slow down system in some cases
    - Can be difficult to program
- Multiprogramming and multithreading are vital concepts

- Next time: Scheduling
- Reminder: Assignment 1 due Thursday
- Project 1 due next Tuesday