csci 3411: Operating Systems

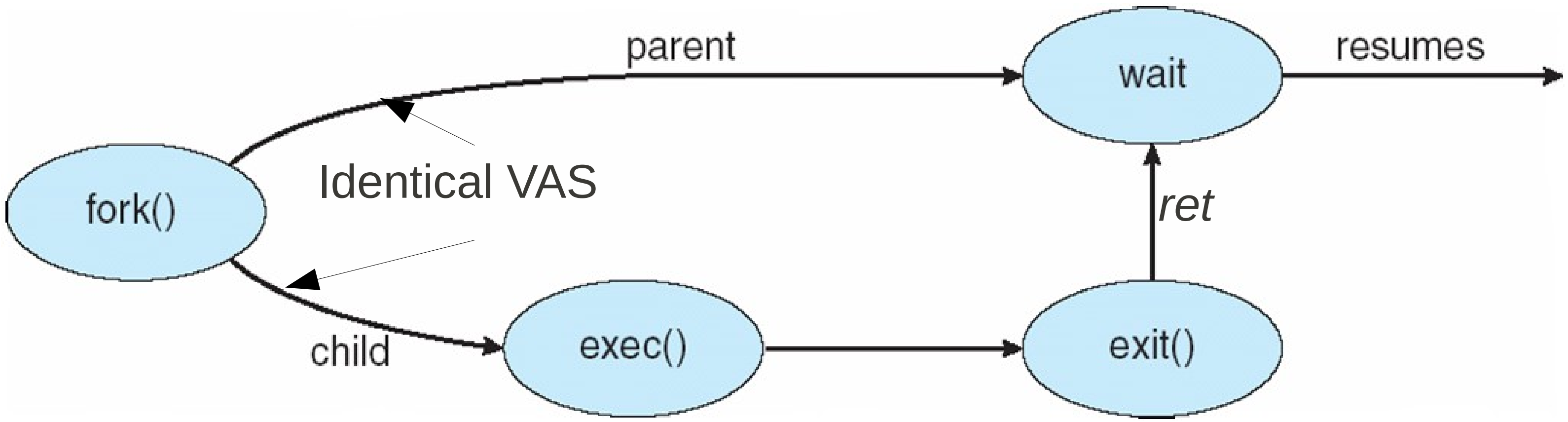
Threads and Communication

Gabriel Parmer

Slides evolved from Silberschatz and West

UNIX Process System Calls

- *fork* – create a new process identical to this one, but in a new virtual address space (VAS)
  - Return “child” process id
  - *exec* system call – load a new program into this VAS
- *exit(ret)* – stop this process
- *ret = wait(child_id)* – parent can wait for child to exit



Process Creation: fork()

- *Parent* process may fork() a *child* process
- Parent can wait(): stop executing till child exit()s
- Parent can kill() its children

- Process hierarchy
  - *Which is the first process?*
  - *Where does a “shell” fit in?*
  - *When does a “shell” wait()?*
  - *What does cntl-C in a “shell” do?*

Process Creation: fork() II

- fork() creates a copy of the parent's address space for the child
  - Copying all memory can be expensive!
- Often intention is to *exec*ute new program
  - exec() or execve() system calls load program from disk into current process
    - *The* way to run a new program
- *So why copy all memory?*
  - vfork() – stop parent's execution till we exec()
  - COW – copy on write memory sharing

Process Termination: exit()

- Release current process' resources back to the system, discontinue execution
- Takes argument: child return value
  - same as returning integer from main function
- Process might stick around with status/return value until parent wait()'s
  - wait() returns the status of the child process
  - “zombie” process – new process state

C Example of Fork Usage

```
int main()
{
     pid_t  pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
            fprintf(stderr, "Fork Failed");
            exit(-1);
    }
    else if (pid == 0) { /* child process: execute “ls” */
            execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
           int status;
            /* parent will wait for the child to complete */
            wait(&status); /* or wait_pid(pid, &status, 0) */
            printf ("Child Complete");
            exit(0);
    }
    return 0;
}
```

Process Cooperation

- fork/exit/wait provide simple cooperation

- Need other means for process coordination?
  - *Can you think of situations where this would be useful?*
  - *Is all IPC via fork/wait?*

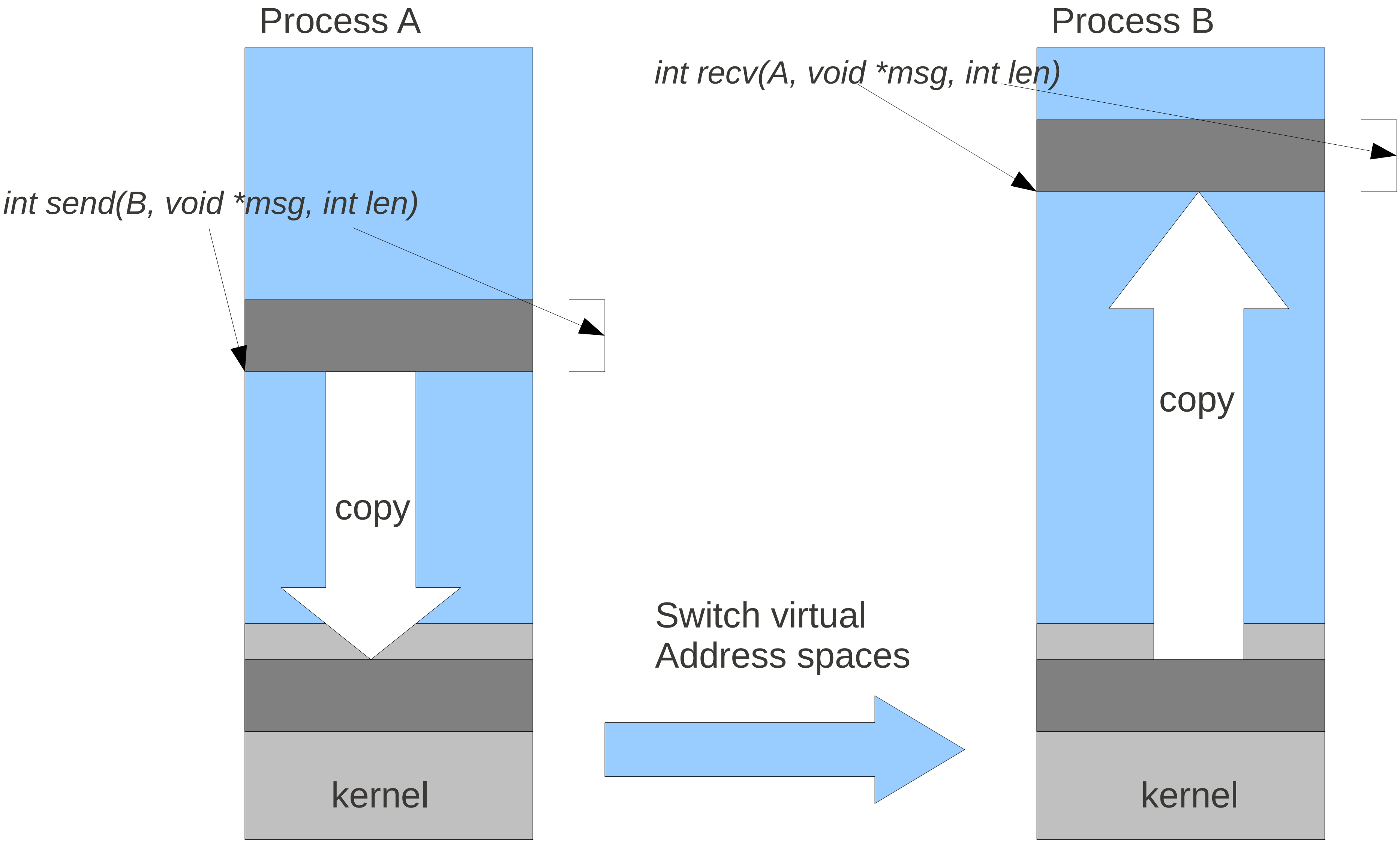
Process Cooperation II

- Concurrency – execution order of two processes is not predetermined
  - Multiple concurrently executing apps
  - Coordination between I/O bound processes
    - e.g. bittorrent, video streaming
- Parallelism – on multi-processor systems, two processes can execute *at the same time*
  - *How can a single application utilize multicore machines?*

Inter-Process Communication (IPC)

- Exchange information/data between procs
  - Process A: *int send(int pid, void *msg, int len)*
  - Process B: *int recv(int pid, void *msg, int len)*
- Data transfer models
  - Shared memory
  - Message passing
- Synchronization models
  - blocking/synchronous
  - non-blocking/asynchronous

Message Passing



Process A

Process B

*int recv(A, void *msg, int len)*

*int send(B, void *msg, int len)*

copy

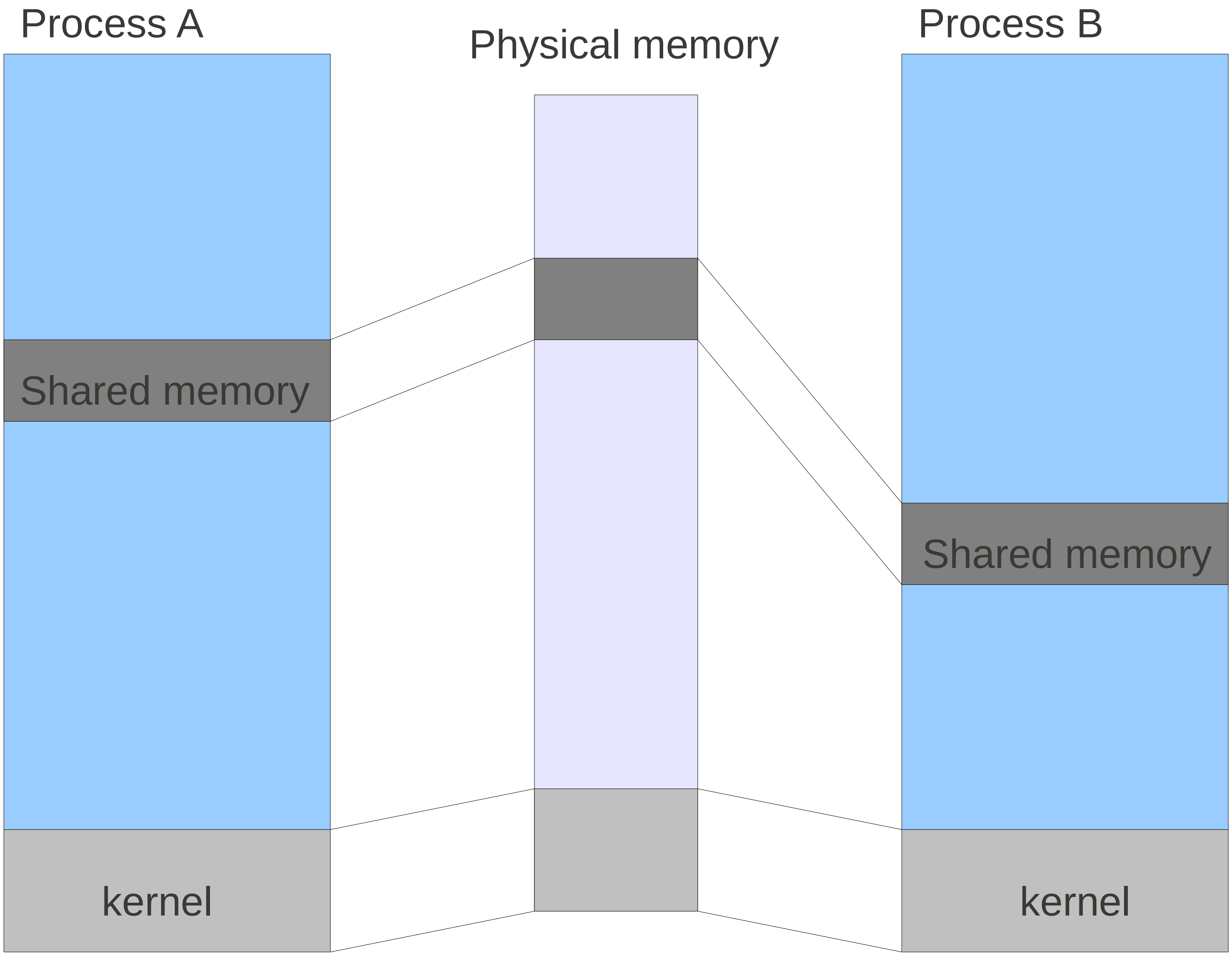
copy

Switch virtual
Address spaces

kernel

kernel

Shared Memory



Message Passing vs. Shared Mem

- Message passing
  - Must copy data
  - Must involve kernel
  - Easy to implement
- Shared memory
  - Copying data optional
  - Parallel processes can avoid invoking kernel

IPC Synchronization: Blocking OPs

- Blocking/Synchronous operations (*send*, *recv*)
  - Process put on process communication queue
  - Data transferred only when other process is also *sends* or *recvs*
  - .. then placed back into runqueue

1) Proc A: *recv(m)*

2) Kernel: remove from runqueue, placed on comm queue

3) Kernel: switch to B

4) Proc B: *send(m)*

5) Kernel: move A to runqueue

6) Kernel: later switch to A

IPC Synchronization: Nonblocking

- Nonblocking/ Asynchronous Ops
  - *send* and *recv* don't block the process
- No data to *recv*?
  - return 0 (bytes read)
- *Proc B sends in inf loop, Proc A never recvs. Problem?*

- Proc A: *recv(m)*
  - If data to be read, return it
  - Else return 0, continue computation
- Proc B: *send(m)*
  - Add data to queue to be read (later) by A
  - If cannot add to queue, return 0

IPC Synchronization: Buffering

- Buffering
  - Communication channel can buffer *N* items
    - Write *N* items to channel → nonblocking *and* data sent
    - Write *N+1* items → block *OR* return 0 (blocking vs. non)
  - Communication channel has *M* items (*M* <= *N*)
    - Read *M* items → nonblocking *and* data read
    - Read *M+1* items → block *OR* return 0 (blocking vs. non)
- *N = 0* → normal blocking

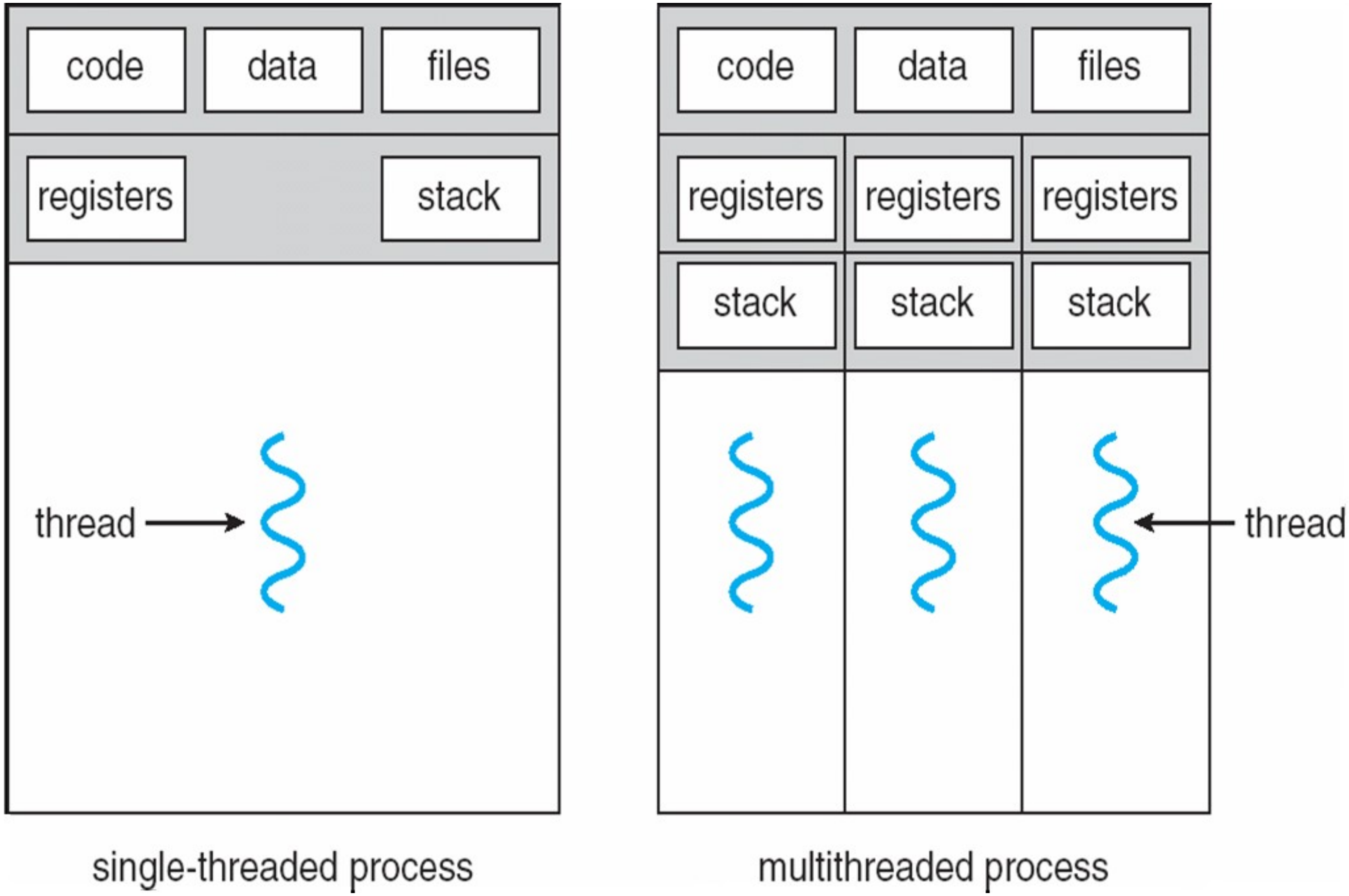
Blocking vs. Nonblocking: Example

- Handing in homework to Prof.
- Need timestamp
  1) Take homework to Prof's office, knock
  2) “block” waiting for Prof. to arrive or open door
  3) Prof. opens door, takes message, you unblock/leave
- Don't need timestamp
  1) Take homework to Prof's office
  2) Slide HW under door and leave
- 10,000 students, 1 prof on vacation. What happens to office?
- Blocking/Nonblocking applies to I/O requests too!

Threads: Alternative for Concurrency/Parallelism

- Each processes has a *flow of control*
  - The sequential execution through the processes' code
- Each of these is a *thread* which consists of
  - Register state (including instruction counter)
  - Execution stack
- A process can have *multiple* threads
  - Multi-threaded application
  - Share data, code, process resources

Threads II

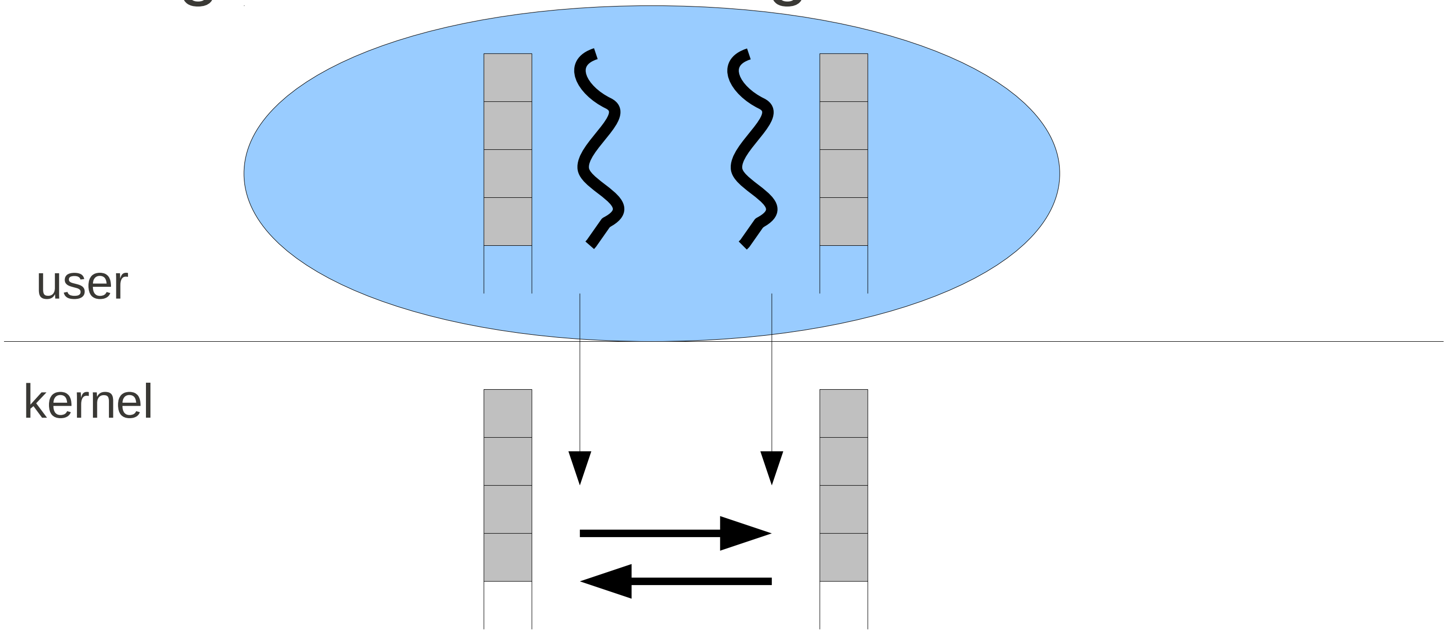

code
data
files
registers
stack
thread
single-threaded process
code
data
files
registers
registers
registers
stack
stack
stack
thread
multithreaded process

Kernel Threads

- Scheduled by the kernel
  - Only execute in kernel!
- Each has its own execution state (blocked, running, ready)
  - Migrates between system queues (run, I/O)

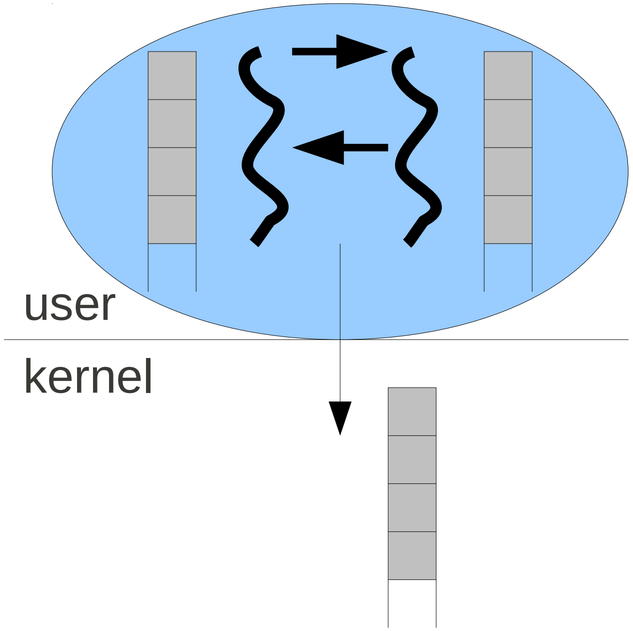
One-to-one/User-Kernel Threads

- Scheduled by the kernel
  - Executes at user-level, make syscalls to call kernel
  - Kernel thd ctxt switch cheaper than proc switch, why?
- Each thread backed by kernel thread
  - blocking/context switching

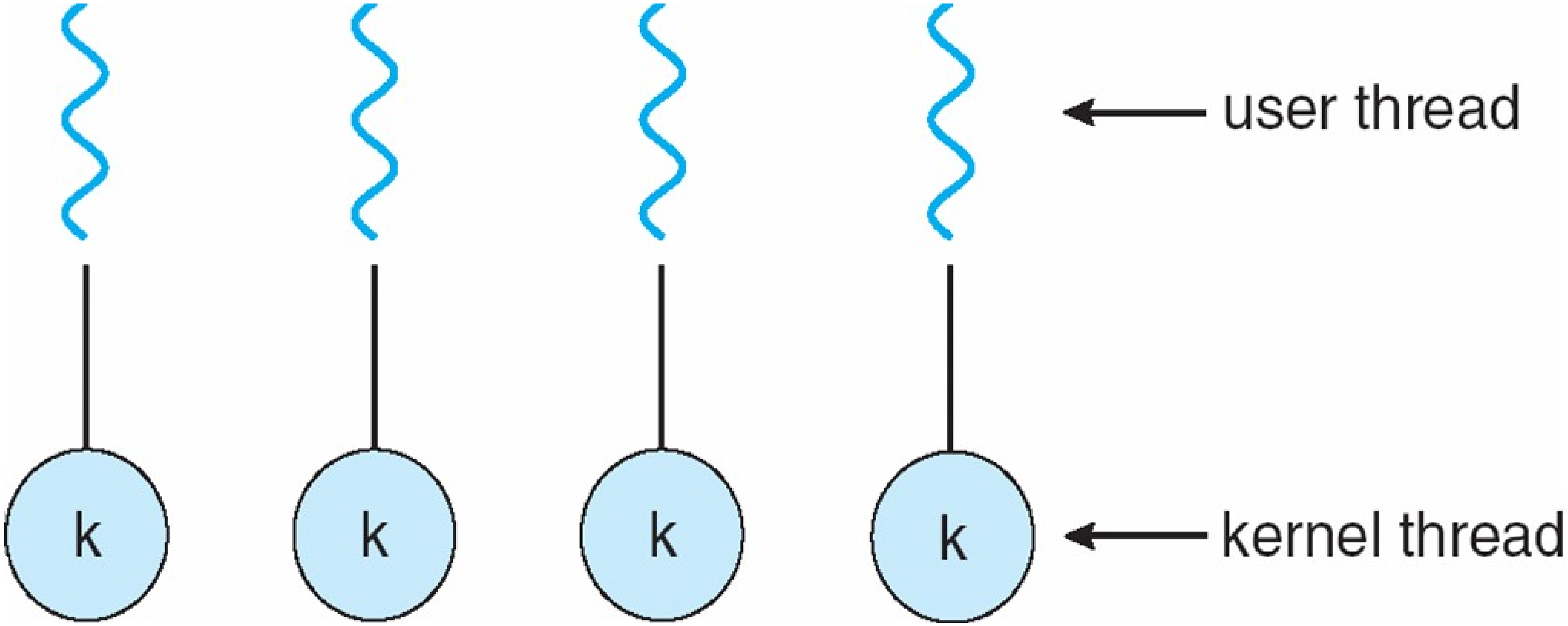


User Threads

- Kernel unaware of their existence
- *Cooperative* switching between threads
  - Threads must *yield* to allow others to execute
    - *Why are they cooperative?*
    - *What enables kernel threads to not need to be cooperative?*
- Context switches lightening fast!
  - Don't need to switch modes to kernel
- *What happens when one user thread requests blocking I/O?*
- *Support parallelism?*



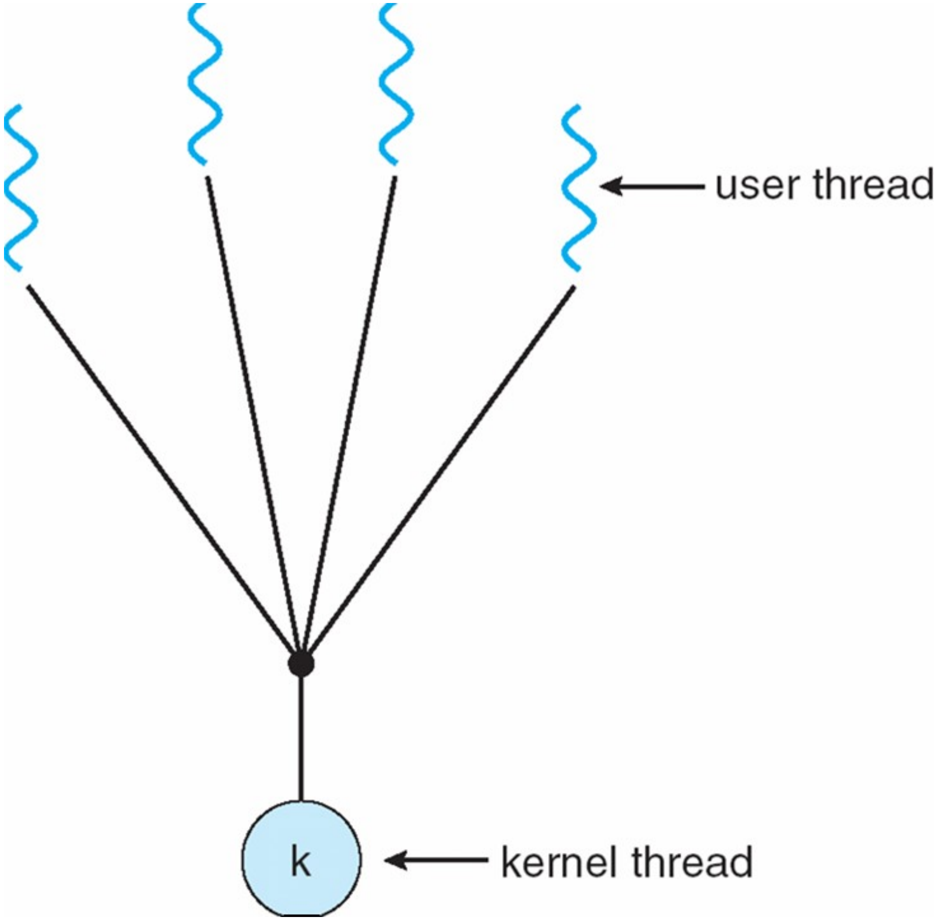
One to one



Method used by Java, Pthreads

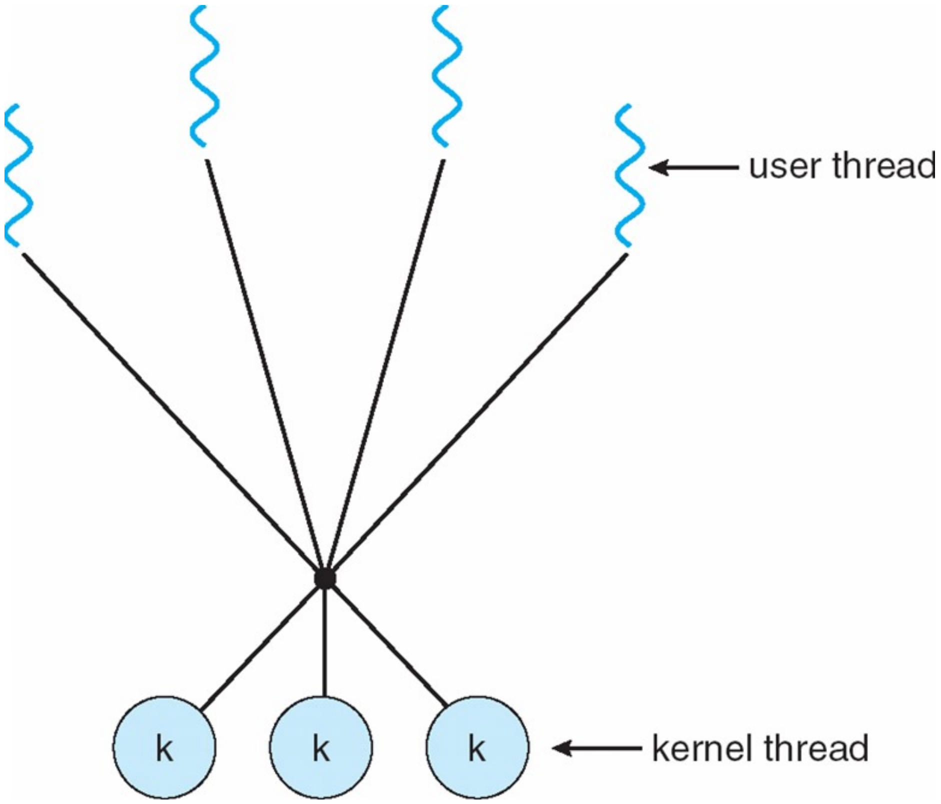
Many to one

- Method used by
  - ruby, ocaml, lua
- You can write your own threading library!



Many to many

- Kernel threads created on demand while there are runnable user threads
- I/O bound user threads tend to use a whole kernel thread
- CPU bound user threads share a single kernel thread



Design of a Facebook Webserver

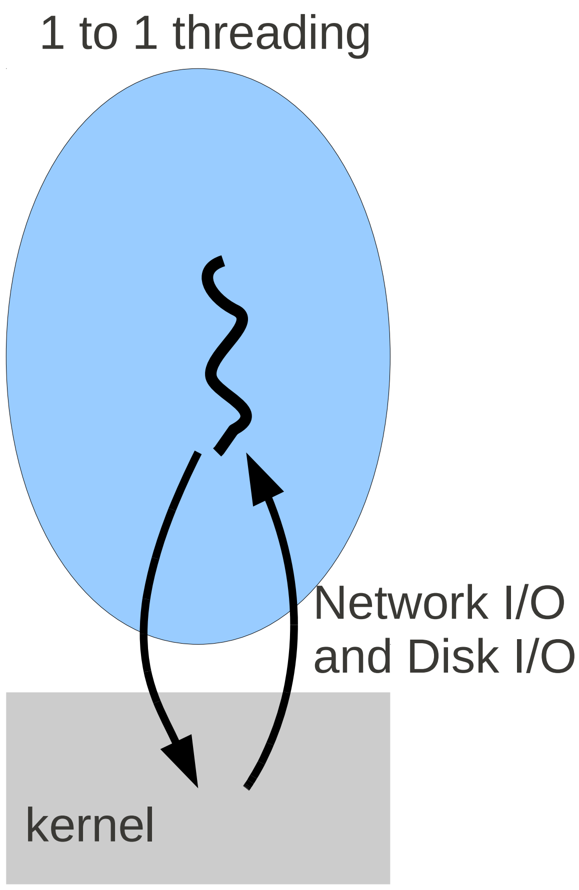
- A thread reads and writes from the network
  - Receives requests from clients for home/wall
  - Writes to the clients the response (i.e. home html)
- Question: how does the webserver retrieve and calculate what the response html should be?
  - Must perform blocking Disk I/O
  - Perform calculations to format the data
  - Given html to network thread to send back to client

Facebook Webserver: Goals

- Throughput: maximize number of clients served per second
  - Minimize cost of processing each content request

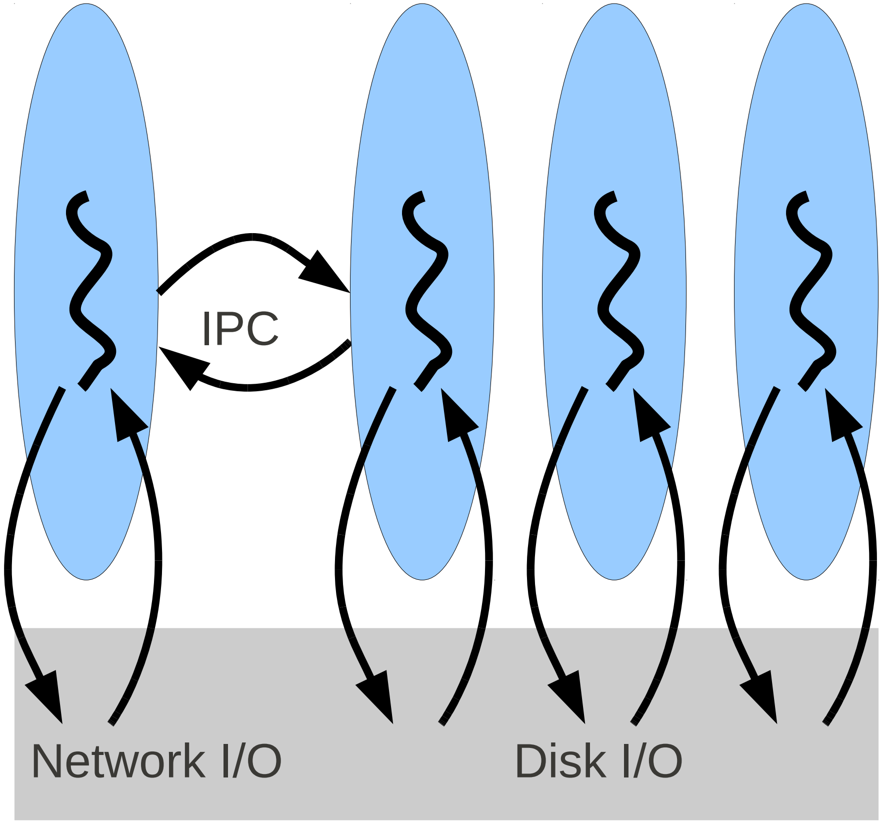
- Reliability: if one part of the system fails, will the rest fail?
  - Reliability: fault isolation

Facebook Webserver: naïve approach



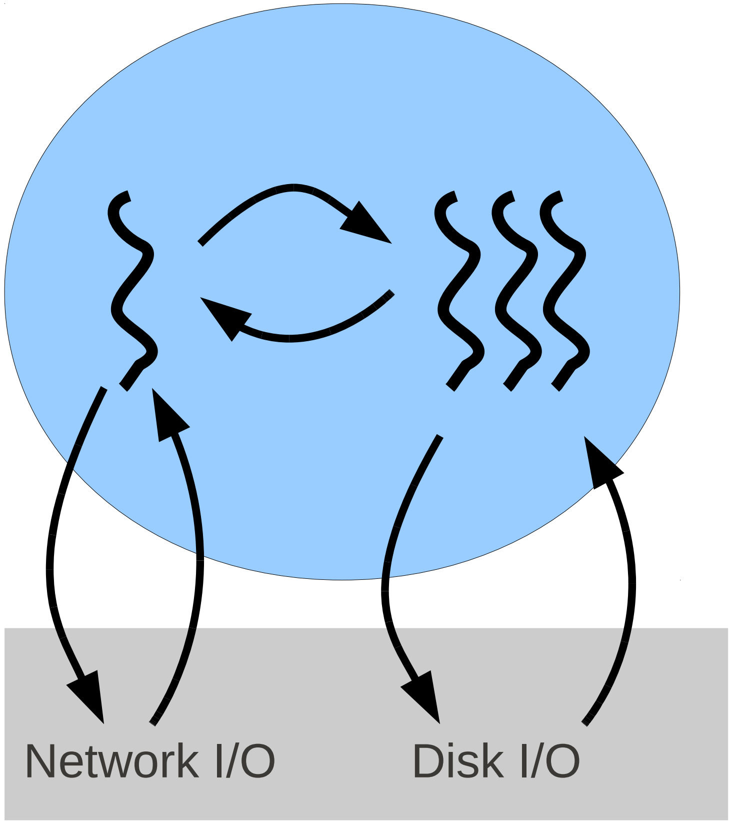
- Single thread
  - reads/writes to network
  - Reads from disk
  - Performs all calculations to format html
- Problems/Benefits?
  - Throughput?
  - Reliability?
  - Parallelism?

Facebook Webserver: Other Possible Approaches



- Multi-process server
  - Networking proc. Uses IPC to deliver requests to “worker” processes
  - Workers compute and do disk I/O
  - Return result to network process
  - blocking/nonblocking IPC?
- Problems/Benefits?
  - throughput/reliability/parallelism

Facebook Webserver: Other Possible Approaches



- Multi-threaded process
  - Network thd communicates with thds for computation and disk I/O
  - Thread type?
    - User threads
    - Kernel threads
- Problems/Benefits?
  - Throughput?
  - Reliability?
  - Parallelism?

Best Approach?

- So which approach is BEST?
  - You know the answer

- Best facebook web-server for what?
  - *Simplicity?*
  - *Reliability?*
  - *Throughput?*