Fork in action

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

int main() {

   int child_pid = fork();

   if (child_pid == 0) {          // child process
        printf("I am process %d… I mean, process %d\n", childpid, getpid());
        return 0;
   } else {                       // parent process
        printf("I am %d the parent of process %d\n", getpid(), child_pid);
        return 0;
   }
}
```

Possible outputs?

Creating and managing processes

Syscall	Description
fork()	Create a child process as a clone of the current process. Return to both parent and child. Return child's pid to parent process; return 0 to child
exec (prog, args)	Run application prog in the current process with the specified args (replacing any code and data that was present in process)
wait (&status)	Pause until a child process has exited
exit (status)	Current process is complete and should be garbage collected.
kill (pid, type)	Send an signal (≈ interrupt) of a specified type to a process (a bit of an overdramatic misnomer...)

[Unix]

Signals

- Signals allow the kernel to inform processes of the occurrence of asynchronous events
- Just as the HW can generate an asynchronous interrupt, which is caught by a handler specified by the kernel...
- ...so the kernel can generate an asynchronous signal, which is caught by a handler specified by the user process

Signals: What purpose?

- Inform of the termination of a process
- Handle exceptions (e.g. attempting to access address outside of virtual address space)
- Handle unexpected error conditions during a sys call (e.g. passing a non-existent syscall no.)
- Asking to receive an alarm after a period of time
- Communicating with other processes via kill syscall
- Inform of a terminal interaction (e.g., ctrl-C)
- …

How does the Kernel send a signal?

- It sets a bit in the process' PCB
  - PCB includes a bit for every possible signal…
  - …but just one bit
    - can remember multiple types of signals
    - but not multiple instances of the same type
- Kernel checks for signals only when process returns from Kernel mode to User mode
  - thus, a user process that is not running is not notified right away

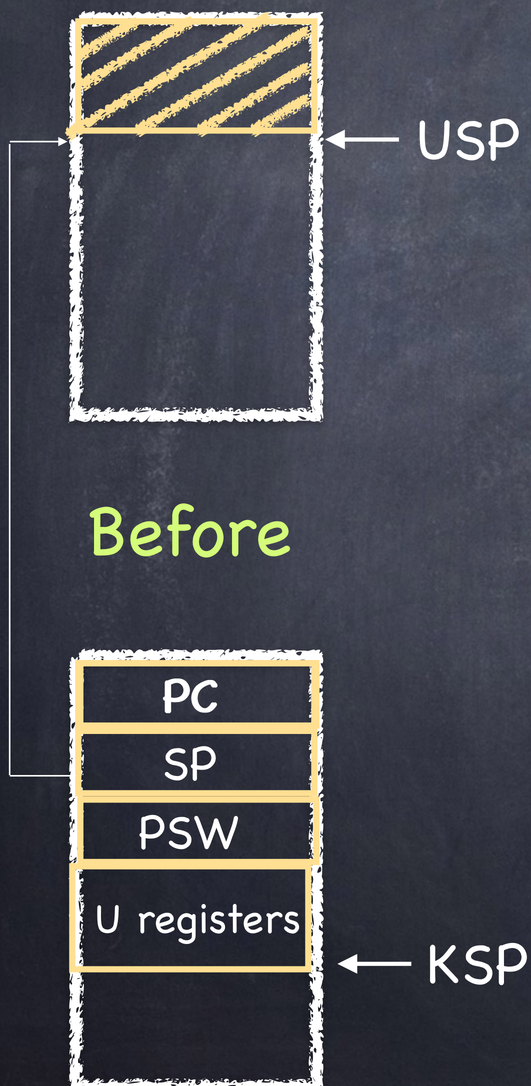
How is a signal handled?

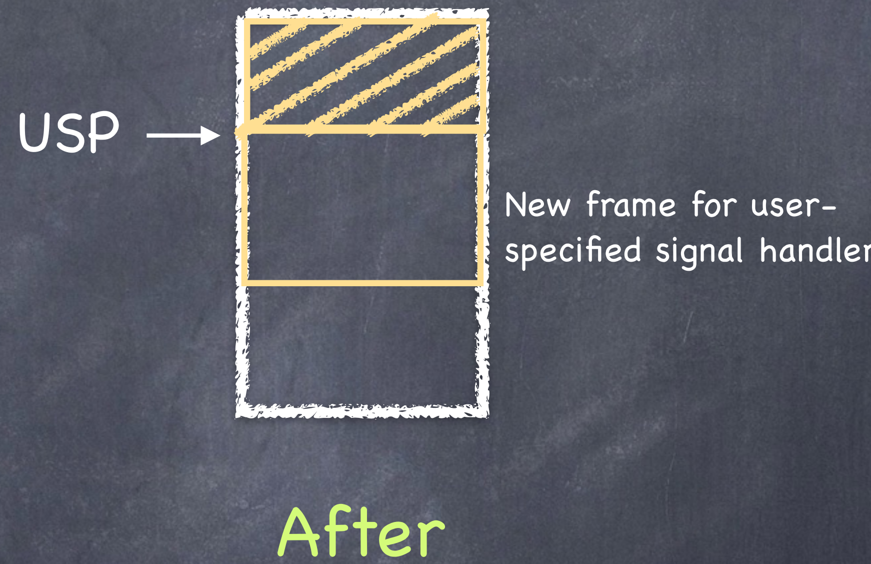
- Three cases
  1. Process exits (default)
  2. Process ignores the signal
  3. Process executes a specific user defined function
     - function specified with the signal system call
       - signal (signum, &function)
       - signal (signum, 0) ≡ exit
       - signal (signum, 1) ≡ ignore

Some POSIX Signals

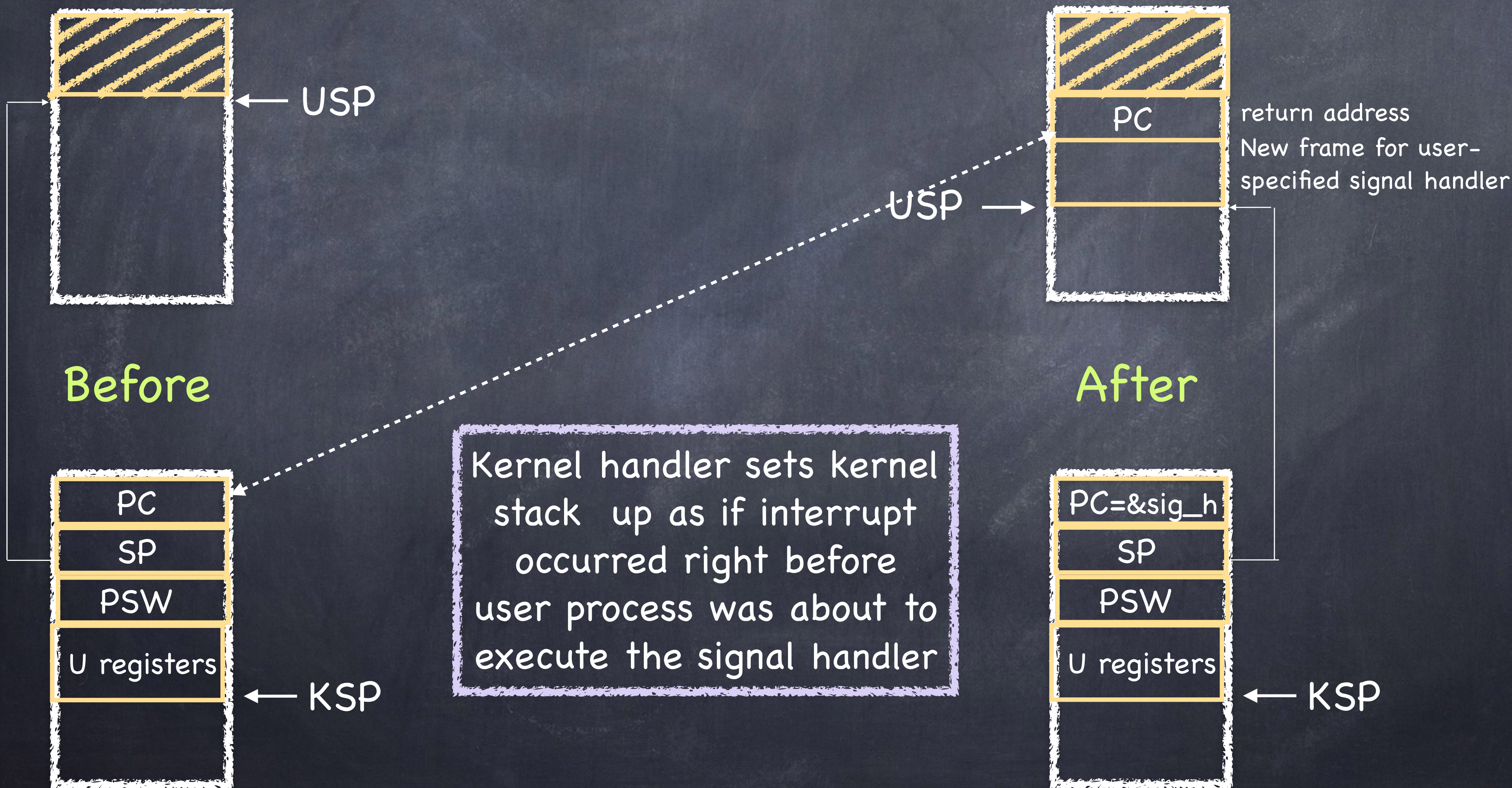
ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt (e.g., CTRL-C from keyboard)
3	SIGQUIT	Terminate (Core dump)	Terminal quit signal
8	SIGFPE	Terminate	Kill program
9	SIGKILL	Terminate	Kill program (cannot be caught or ignored)
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated
20	SIGSTP	Stop until SIGCONT	Stop signal from terminal (e.g., CTRL-Z from keyboard)

Signal Handling: The Mechanism





Signal Handling: The Mechanism



Handler Example

```c
void int_handler(int sig) {
    printf("Process %d received signal %d\n", getpid(), sig);
    exit(0);
}
int main() {
    pid_t pid[N];
    int i, child_status;
    signal(SIGINT, int_handler) // register handler for SIGINT
    for (i = 0; i < N; i++) // N forks
        if ((pid[i] = fork()) == 0) {
            while(1); // child infinite loop
        }
    /* Parent terminates the child processes */
    for (i = 0; i < N; i++) {      // parent continues executing
        printf("Killing proc. %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }
    /* Parent reaps terminated children */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status)) // parent checks for each child's exit
            printf("Child %d terminated w/exit status %d\n", wpid,
                    WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
    exit(0);
}
```

Header files

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

Review

- A process is an abstraction of a running program
- The process' context captures its running state:
  - registers (including PC, SP, PSW)
  - memory (including the code, heap, stack)
- The implementation uses two contexts:
  - user context
  - kernel (supervisor) context
- A Process Control Block (PCB) serves both contexts and has other information about the process

Review

- Processes can be in one of the following states:
  - Initializing
  - Running
  - Ready (aka "runnable" on the "ready" queue)
  - Waiting (aka Sleeping or Blocked)
  - Zombie

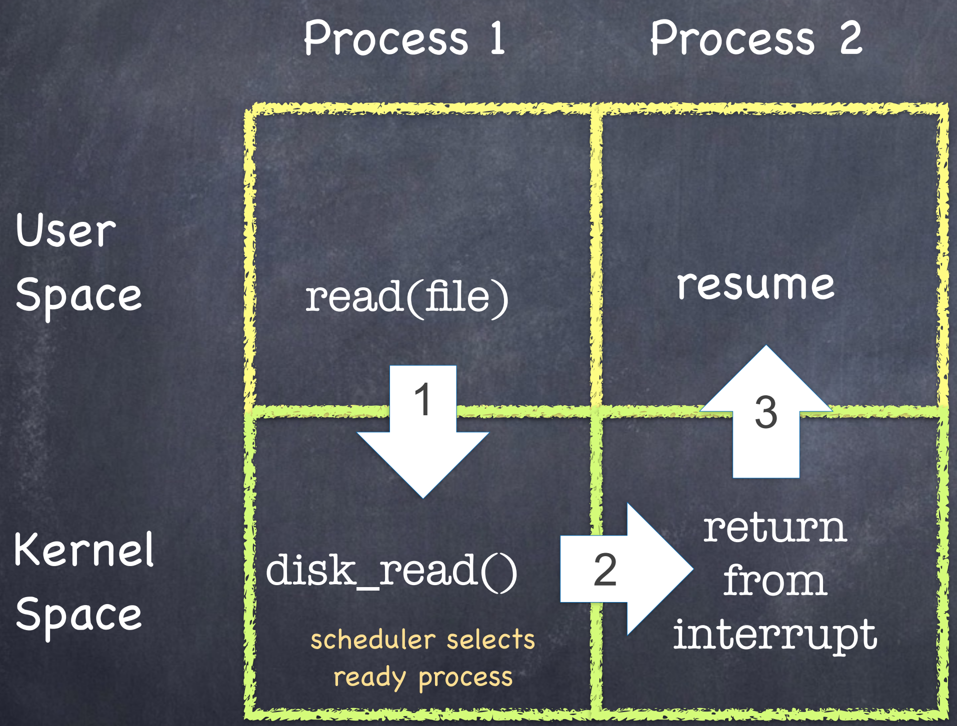
More Processes than Processors

- Solution: time multiplexing
  - Abstractly each processor runs:
    - for ever:
      - NextProcess = scheduler()
      - Copy NextProcess->registers to registers
      - Run for a while
      - Copy registers to NextProcess->registers
  - Scheduler selects some process on the ready queue

Three Flavors of Context Switching

- Interrupt: from user to kernel space
  - on system call, exception, or interrupt
  - Stack switch: P_x user stack $\rightarrow$ P_x interrupt stack
- Yield: between two processes, inside kernel
  - from one PCB/interrupt stack to another
  - Stack switch P_x interrupt stack $\rightarrow$ P_y interrupt stack
- Return from interrupt: from kernel to user space
  - with the homonymous instruction
  - Stack switch: P_x interrupt stack $\rightarrow$ P_x user stack

Switching between Processes



1. Save Process 1 user registers (including SP and PC)
2. Save Process 1 kernel registers; switch SP; restore Process 2 kernel registers
3. Restore Process 2 user registers

Threads

Our second major abstraction

(Chapters 25-27)

A new Abstraction

- The process abstraction gives each running program the illusion of running on a machine of their own
  - CPU & Memory
- Context switching allow to support multiple "virtual machines" on top of a single physical machine
- ...but a machine may have multiple CPUs...

Threads

- It is how the kernel virtualizes a CPU!
  - A thread's state consists of
    - registers (including PC and SP)
    - a stack
  - it lives inside some host address space (provided by the host process)
- Just as a single machine can have multiple CPUs, so a single process can host multiple threads
  - all sharing the same Virtual Address Space (the one of the host process)

The Power of Abstractions

Infinite machines! †

Infinite cores!†

†on a single CPU (?!?)

Processes and Threads

- The processes that we have described so far host one thread only
- Many OSs offers the ability to have multiple concurrent threads execute in a process
  - Multiple threads in a process allow multiple task to be performed concurrently, at the same time (at least, logically)
    - multiple processes too —-but they do not easily communicate. A process' threads instead share the same memory!
- A kernel that supports multi-threading manages hardware resources differently:
  - CPU state managed on a per-thread basis
  - All other resources on a per-process basis

Why Threads?

- To express a natural program structure
  - updating the screen, fetching new data, receiving user input — different tasks within the same address space
- To exploit multiple processors
  - different threads may be mapped to distinct processors
- To maintain responsiveness
  - slow, long running task performed by background threads
  - foreground threads respond immediately to user interactions
- Masking long I/O device latency in blocking syscalls
  - do useful work while waiting

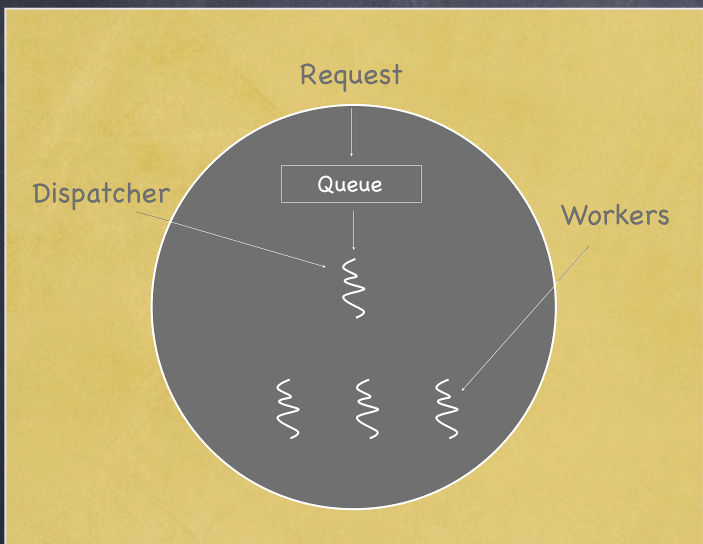
Multithreading: Responsiveness

- Common web browser pattern:
  - UI thread draws web page, handles mouse clicks
  - Pool of background threads downloads web pages from remote web servers
- Does this require multiple CPUs to yield a benefit?
  - NO!
  - BG threads will usually be blocked on I/O
  - Ditto for UI thread
- Even with a single processor, multithreading can greatly improve application responsiveness
  - especially when tasks are I/O bound

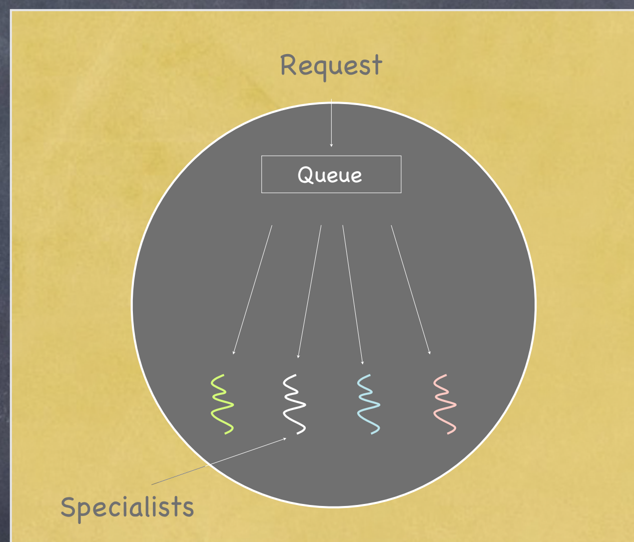
Multithreading: Scalability

- A large scientific/mathematical computation:
  - instead of using a single thread, split in multiple concurrently executing threads
- Does this require multiple CPUs to yield a benefit?
  - YES!
  - Threads will be mostly CPU bound, not I/O bound
  - With only one CPU, multithreading will actually likely slow execution, not speed it up!
    - (context switches, synchronization overheads, etc)
- On the other hand... A single-threaded process cannot take advantage of multiple CPUs
  - need either multiple processes, or one process with multiple threads

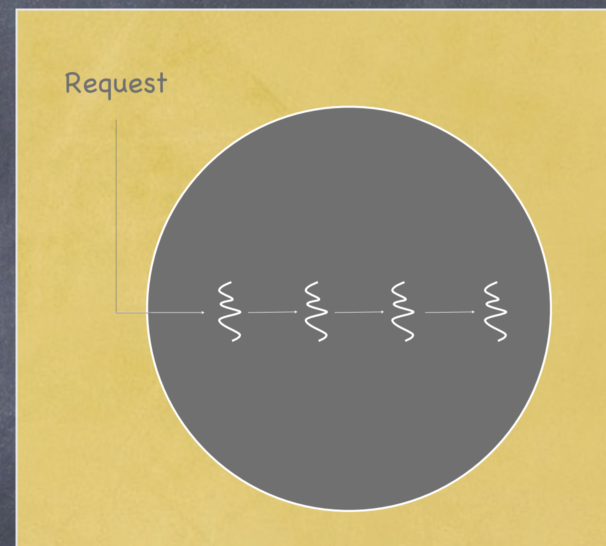
Multithreaded Processing Paradigms



Dispatcher/Workers



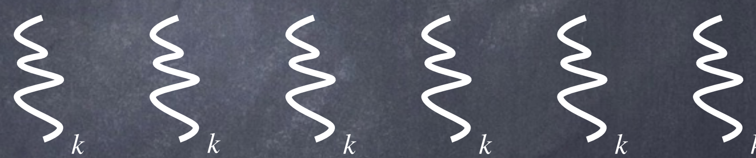
Specialists

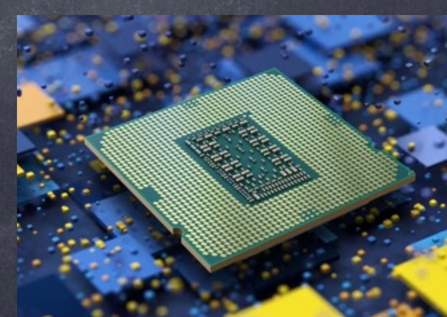


Pipeline

Where should threads be implemented?

- In the Kernel!
  - Kernel multiplexes each physical CPU across multiple threads
  - Kernel can assign one or more threads to a process
  - Scheduler schedules threads

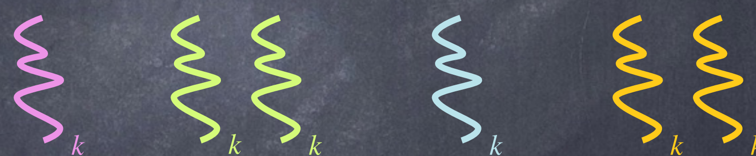




Hardware

Where should threads be implemented?

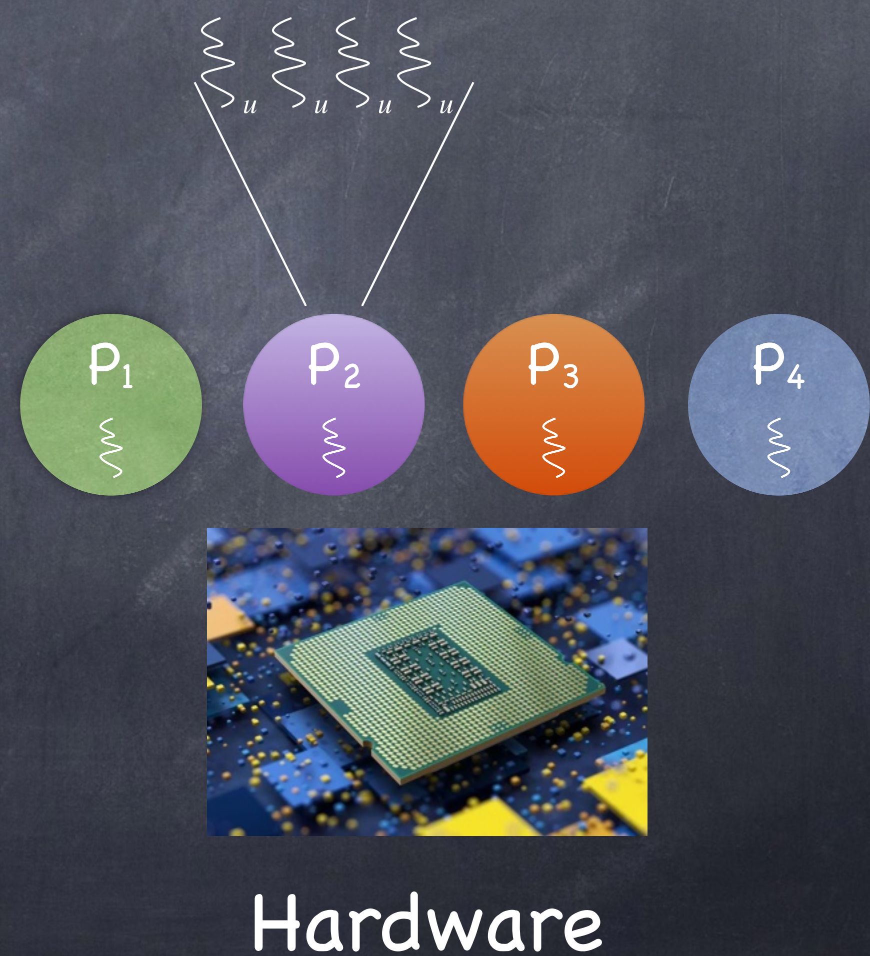
- In the Kernel!
  - Kernel multiplexes each physical CPU across multiple threads
  - Kernel can assign one or more threads to a process
  - Scheduler schedules threads



Hardware

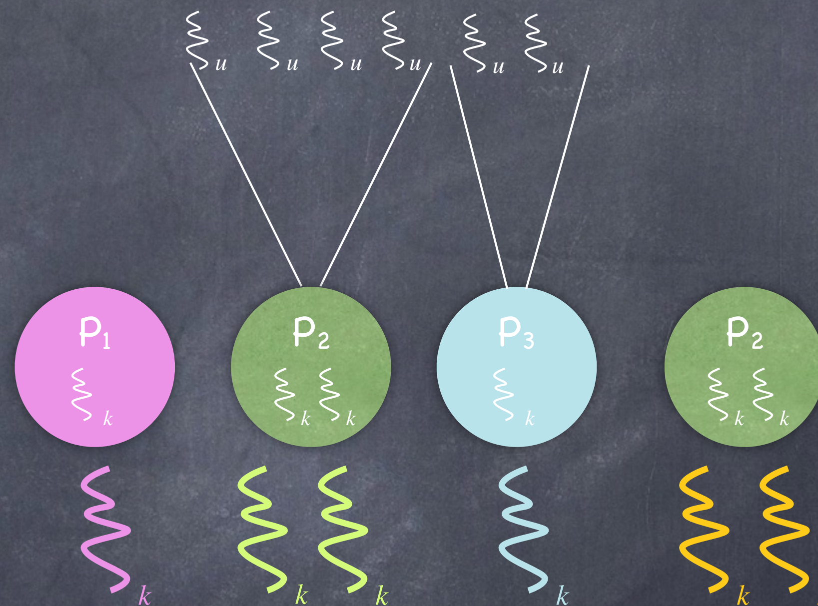
Where should threads be implemented?

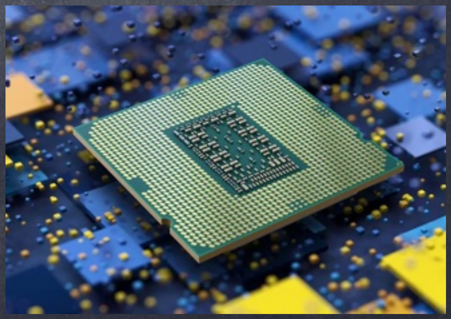
- In User space!
  - Kernel assigns one thread per process
  - Kernel multiplexes each physical CPU across multiple processes
  - Scheduler schedules processes
  - User level library multiplexes the process' single kernel thread across multiple user level threads



Where should threads be implemented?

- In both!
  - Kernel multiplexes each physical CPU across multiple threads
  - Kernel can assign one or more threads to a process
  - Scheduler schedules threads
  - User level library multiplexes the process' single kernel thread across multiple user level threads





Hardware