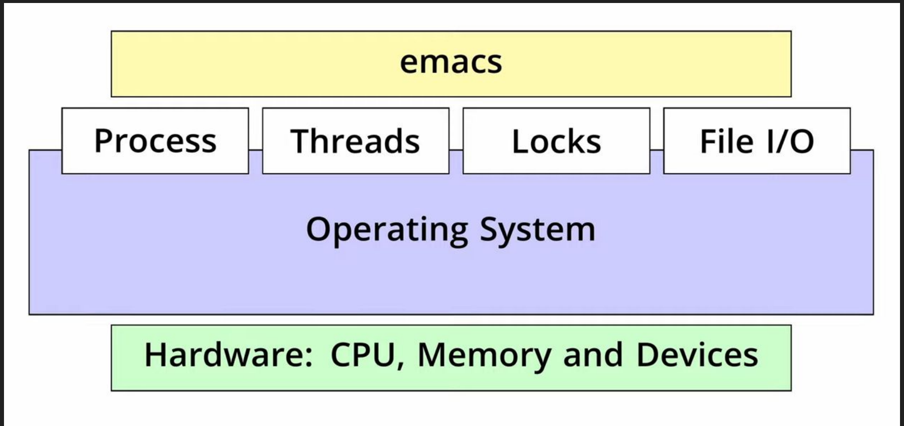
04 - Processes and Threads

CEG 4350/5350 Operating Systems Internals and Design
Max Gilson

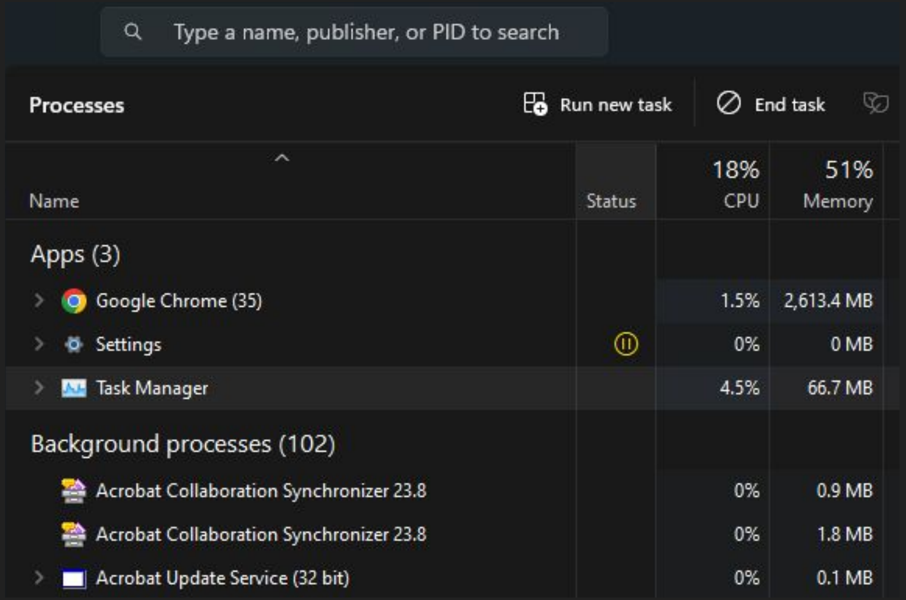
The Process Abstraction

- The OS provides abstractions for our programs
  - Abstraction is breaking down complex problems into smaller, manageable parts
  - An abstraction is basically a simplified interface
- The OS provides a process abstraction for our programs to use to get access to CPU time, memory, or other resources
- Emacs is a text editor available on Linux



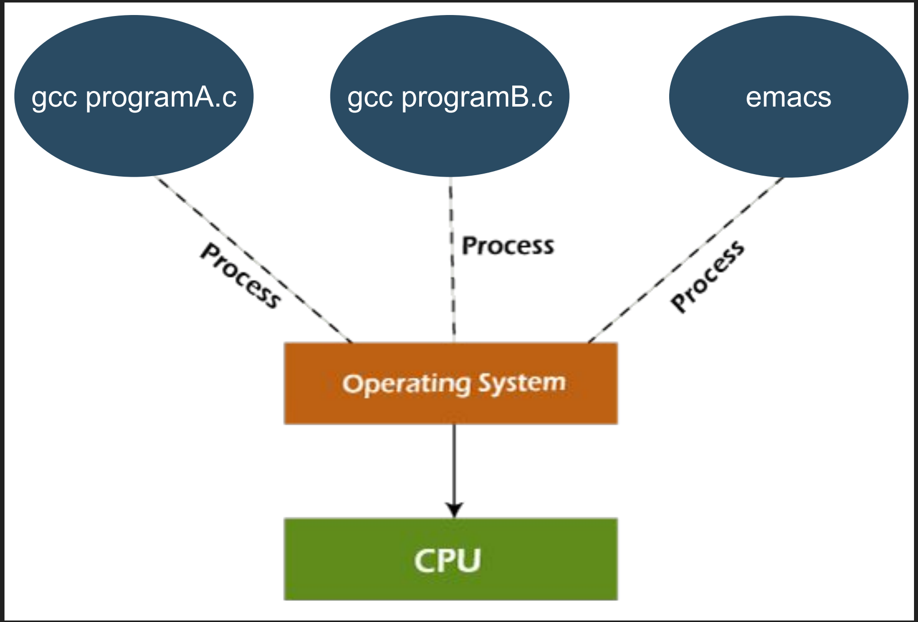
The Process Abstraction (cont.)

- A process (sometimes called a job or task) is an instance of a program running: Chrome, Fortnite, Notepad++, etc.
- Typically, multiple open windows of a program are still one process
  - Exception: Chrome creates a process for each site visited or tab opened
- Most OS can run multiple processes simultaneously
  - Notice the 35 Chrome processes ->



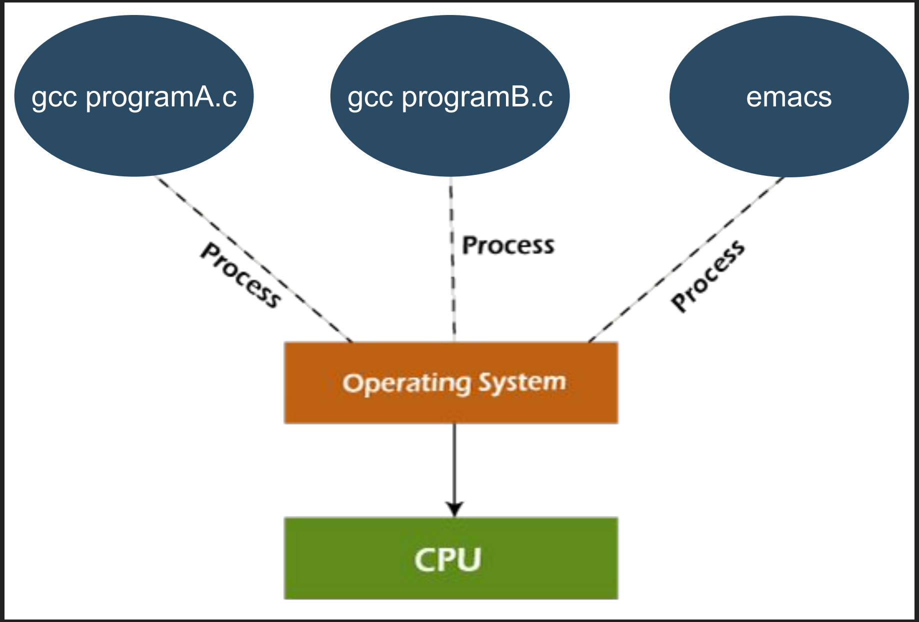
The Process Abstraction (cont.)

- The process abstraction ensures that data inside each process is unique to the process itself
  - Example: Starting an instance of gcc to compile programA.c and simultaneously starting another instance of gcc to compile programB.c
  - The two processes won't accidentally combine programA and programB together or get their data mixed up while compiling
  - The processes are kept separate



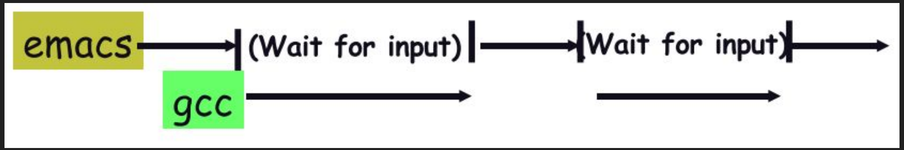
The Process Abstraction (cont.)

- This separation of processes states and data keeps our jobs as programmers "simple"
  - When gcc is compiling programA it doesn't have to worry about any other instance of gcc that might be currently running
  - When new processes are ran the states and data are basically reset
- Running multiple processes allow us to better utilize our CPU's computing power
  - Imagine only being able to run 1 process that spends 1 clock cycles running and 1,000,000 clock cycles waiting!



Multiple Processes Can Improve Performance

- Emacs (a text editor) spends most of its time waiting for you to type
  - So why not give up that CPU time spent waiting to a program that needs it (a compiler)

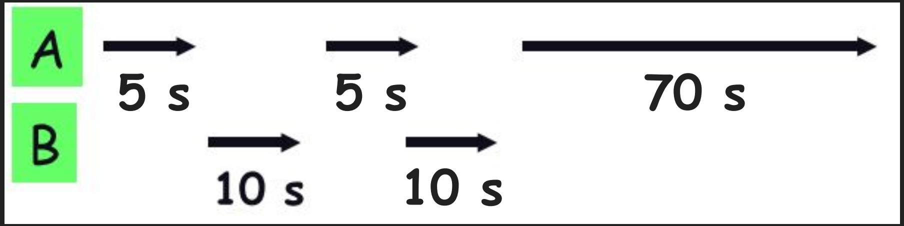


Multiple Processes Can Improve Performance (cont.)

- Old school computers could only run one process at a time
  - One right after the other
- Modern OSs allow us to switch between processes
  - From the user's perspective, the computer is running much faster!
- Note: Context switching (switching between processes) takes slightly more time than if ran one right after the other

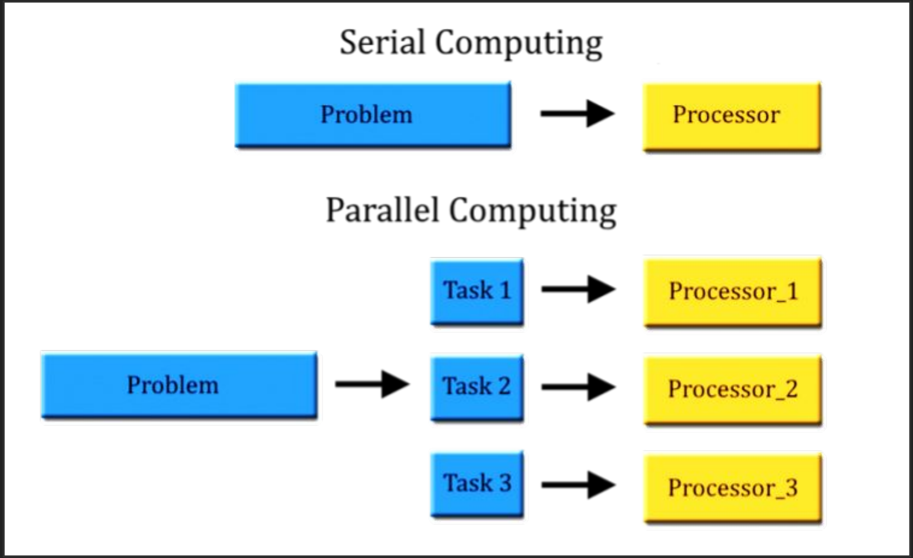
Old way:

Modern way:



Parallelism

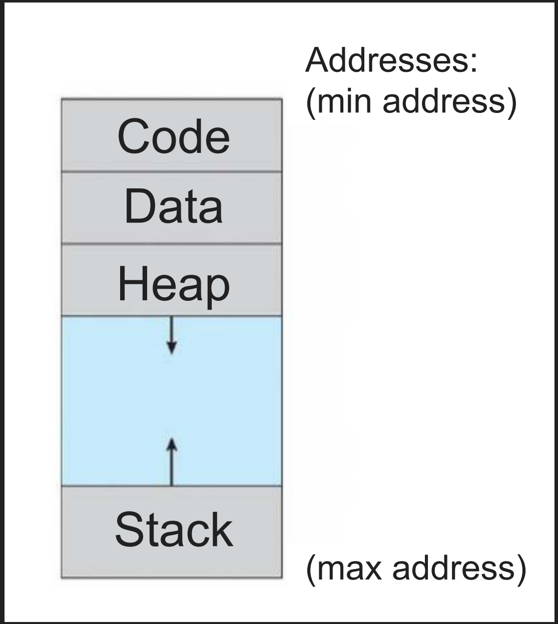
- Parallelism is the simultaneous execution of multiple tasks or processes in order to increase efficiency and speed
  - Imagine it takes a factory worker 1 day to make a product
  - If you want to make 100 products it will take this worker 100 days
  - Hire 100 workers it will take 1 day to make 100 products if they work perfectly in parallel
- Multi-core CPUs are how real parallelism is possible
- A 4 core computer can run 4 processes in parallel which yields 4x throughput



From the Process's Perspective

- Each process has its own unique view of the machine it's running on
  - Address space for code and data
  - Opened files (in memory)
  - "Virtual" CPU
    - The OS can take away CPU access whenever it wants even though the process does not know this
- One process of gcc is isolated from another process of gcc in memory
- To keep track of multiple processes information the OS uses a Process Control Block (PCB)
  - We'll take about this later

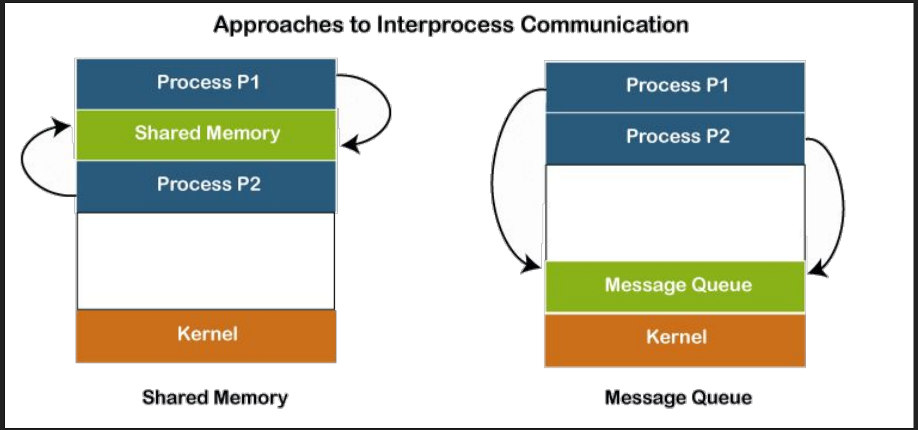
Process Memory (Process's View)



PCB (OS's View)

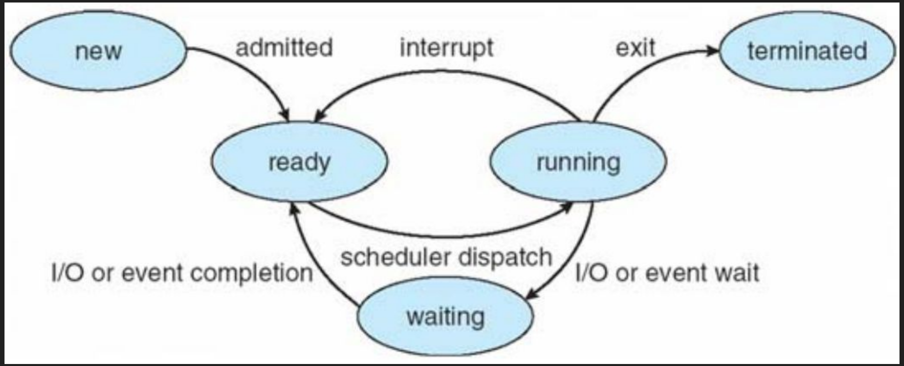
Processes Communicating

- Sometimes, we want processes to communicate with each other
- This is called Inter-Process Communication (IPC)
  - Shared files
    - i.e. edit a file with a emacs, save it, then compile that file with gcc
  - Shared memory
  - Message passing



Processes States

- New
  - A new process wants to run
  - The OS *admits* the process to the pool of other processes that want to run
- Ready
  - The process is now ready to run but must wait for the OS to dispatch it
- Running
  - The process is now executing on the CPU but it can be interrupted or forced to wait for I/O (i.e. keyboard or disk)
- Waiting
  - The process cannot run because it is waiting for something to happen
- Terminated
  - The process has finished or was forcefully terminated (i.e. closing out a window)



Creating a New Process in Linux

- Linux system calls make it easy to create new processes of the current program in C using the `fork()` function
- `int fork(void);`
    - Creates a process that is an exact copy of the current one that starts immediately after the fork() call
    - Returns process ID of new process to “parent”
    - Returns 0 to “child”
    - Returns negative if error

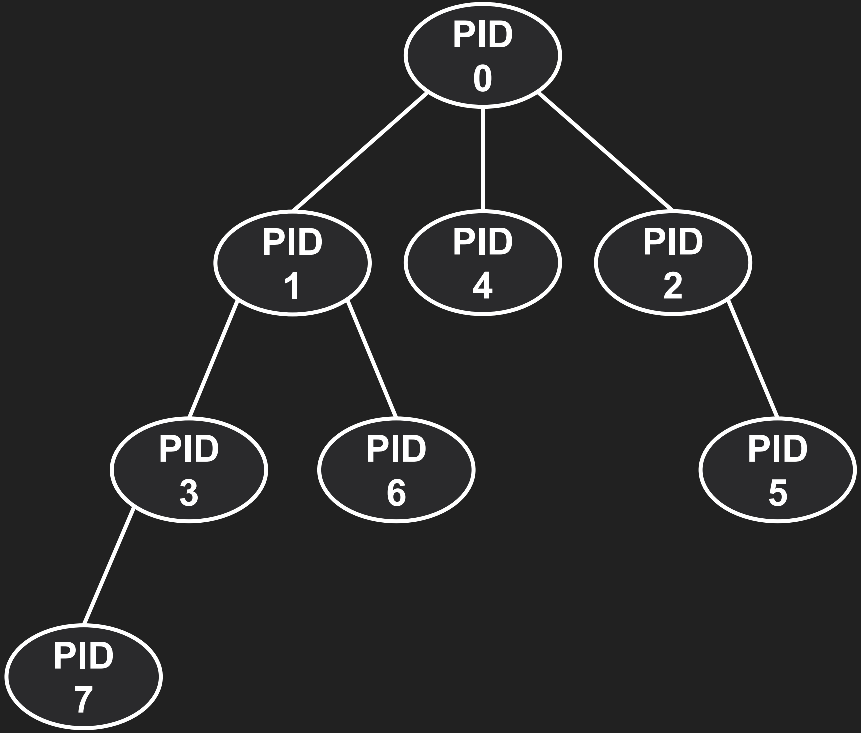
```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    fork();
    fork();
    fork();
    printf("Hello World!\n");
    return 0;
}

// "Hello World!" will get printed
8 times, why?
```

Creating a New Process in Linux (cont.)

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    fork();
    fork();
    fork();
    printf("Hello World!\n");
    return 0;
}

// "Hello World!" will get
printed 8 times, why?
```



Creating a New Process in Linux (cont.)

- Your program can also wait for your child processes to finish before continuing using the `waitpid()` function
- `int waitpid(int pid, int *stat, int opt)`
  - `pid` - the process id to wait for, or -1 for any
  - `stat` - contains exit value of finished process
  - `opt` - custom options
  - Returns process ID or -1 on error

Creating a New Process in Linux (cont.)

- Sometimes you might want to execute a different program instead of creating a child process
- For this you can use `execve()`, `execvp()`, or `execlp()`
- `int execve(char *prog, char **argv, char **envp)`
  - `prog` - full path of program to run
  - `argv` - arguments the program should run with
  - `envp` - environment variables such as `PATH`, and `HOME`
  - Returns -1 if an error has occurred calling the program, i.e. program could not be found
  - The current process is now overtaken by the program we have specified
- `int execvp(char *prog, char **argv)`
  - Search `PATH` for `prog` using the current environment
- `int execlp(char *prog, char *argv, ...)`
  - List arguments one at a time, finishing with `NULL`

Terminating a Process in Linux

- You can forcefully terminate the current process (or another process) using `exit(status)` or `kill(pid, SIGTERM)`
- `void exit (int status)`
  - `status` - status code returned to `waitpid`
  - Terminates the current process
  - By convention, `status` of 0 is success, non-zero is error
- `int kill(int pid, int sig)`
  - `pid` - the process id you want to terminate/kill
  - `sig` - either `SIGTERM` (15) or `SIGKILL` (9)
  - `SIGTERM` - safely terminate the process but allow the process to do some clean up before it is forced to terminate
  - `SIGKILL` - immediately terminate the process and give no warning or time for process to clean up

Creating a New Process in Windows

- Creating a new process in Windows is not so straightforward and requires many arguments using the Windows API, `WINAPI`
- There are even multiple functions that can be called to create a process, which makes it more confusing:
  - `CreateProcess()`
  - `CreateProcessAsUser()`
  - `CreateProcessWithLogonW()`
  - `CreateProcessWithTokenW()`
  - `...`

```
BOOL WINAPI CreateProcess(
  _In_opt_    LPCTSTR lpApplicationName,
  _Inout_opt_ LPTSTR lpCommandLine,
  _In_opt_    LPSECURITY_ATTRIBUTES lpProcessAttributes,
  _In_opt_    LPSECURITY_ATTRIBUTES lpThreadAttributes,
  _In_        BOOL bInheritHandles,
  _In_        DWORD dwCreationFlags,
  _In_opt_    LPVOID lpEnvironment,
  _In_opt_    LPCTSTR lpCurrentDirectory,
  _In_        LPSTARTUPINFO lpStartupInfo,
  _Out_       LPPROCESS_INFORMATION lpProcessInformation
);
```

From the OS's Perspective

- The OS maintains a data structure for each process called a Process Control Block (PCB)
  - Sometimes called Task Control Block (TCB)
- Tracks state of process
  - Running, ready, waiting, etc.
- Includes necessary information for running
  - Current registers being used, virtual memory mappings (what's in memory?), etc.
  - Open files
- Includes other details
  - User credentials, priority, etc.

PCB
(OS's View)

Scheduling Processes

- If more than 1 process needs to run, the OS must schedule them to run individually
  - If the machine has multiple cores and is capable of parallelism, multiple processes can run simultaneously
  - Without parallelism, your computer just appears to be doing many things at the same time because it’s just switching between them very quickly
- The OS will look at its list of PCBs, find all that are “Ready”, and decide which one gets to run
- How should the OS decide which one gets to run if there are multiple ready processes?
  - FIFO - First process that was ready is the first to run
  - Round Robin - Arrival time, burst time, and quantum
  - Priority - Highest priority runs first

Preemptive Multitasking

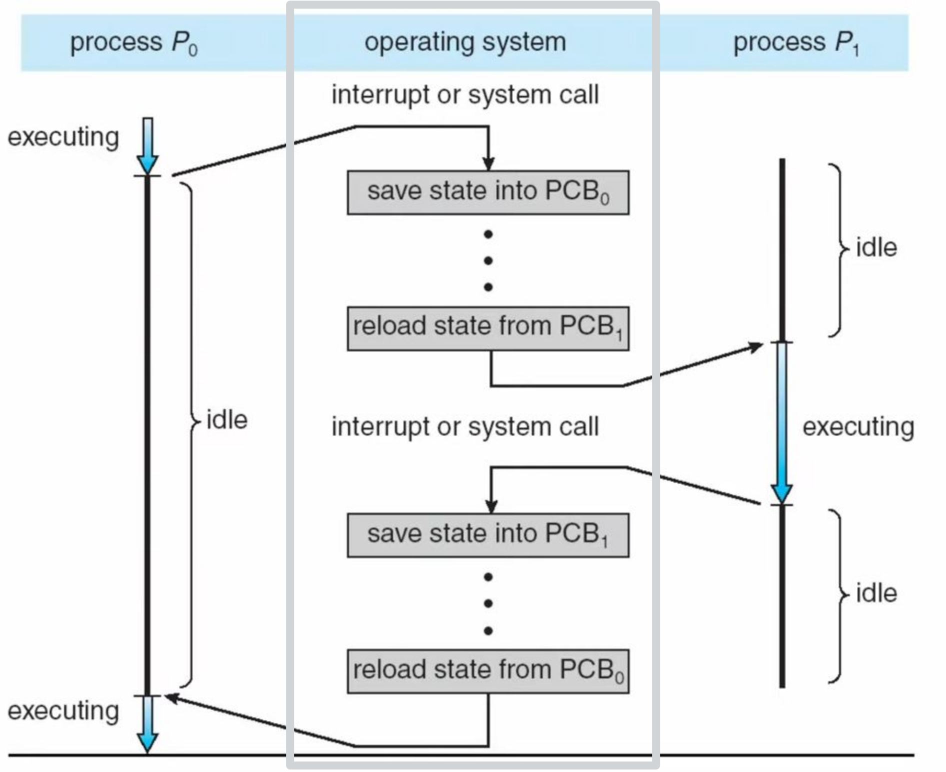
- A process can be preempted by the OS
  - Preempted means, pause for a short time, and allow something else to run, then resume later
  - Without preemption, 1 process could hog the CPU forever and your computer would appear frozen!
- The process may also preempt itself:
  - Makes a system call, waits to read disk, makes another process runnable (i.e. `fork()`), etc.
- Periodic timers interrupt the process
  - If the OS gives a fixed maximum amount of time the process can run, and the process exceeds this time, preempt the process so another process gets a chance to run
- Device interrupts
  - If process A is stuck waiting for our keyboard while process B is running, and we finally press a key, preempt process B and let the process A get a chance to capture the keyboard input
- The changeover between running processes is called *context switching*

Cooperative Multitasking

- A process must willingly yield or give up control for another task to run
  - Cooperative means every process must cooperate
  - At some point a process must give other tasks/processes a chance to run
  - The downside is if you have a process that does not play fair and hogs the CPU!
- This type of multitasking is much easier to implement because it does not involve any interrupts (timers)

Context Switch

- Switching between running processes is called context switching
- Process P_0 is currently executing but we want to run P_1:
    1. Save P_0's PCB data
    2. Reload P_1's PCB data
    3. Run P_1
- When the OS wants to switch back to P_0 the steps are the same
- If a process is not executing, it is not doing anything useful and is just waiting for the OS to give it a chance to run



Context Switch (cont.)

- Context switching is not free, it costs CPU time and memory (to store and access PCB data)
- Context switching is very hardware dependent
  - Which registers should get saved in the PCB and restored when we run the process?
    - What about floating point or special registers?
  - Are there flags that must be maintained?
  - Save/restore memory translations

Implementing Basic Multiprocessing in C

- To allow your OS to run multiple processes, we need to implement multiprocessing
  - Also known as multitasking
- Our OS will switch between multiple processes and execute each of them individually
- How does our system know when it needs to switch between processes?
  - Preemptive - hard to implement but better solution
  - Cooperative - easy to implement but worse solution

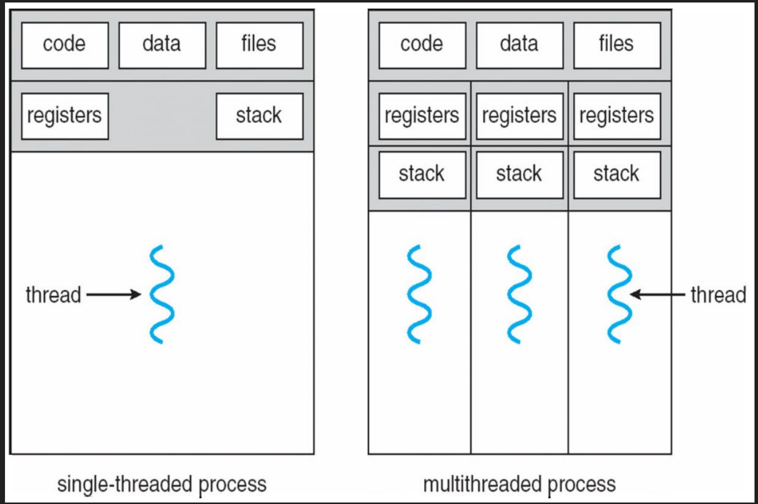
Implementing Basic Multiprocessing in C (cont.)

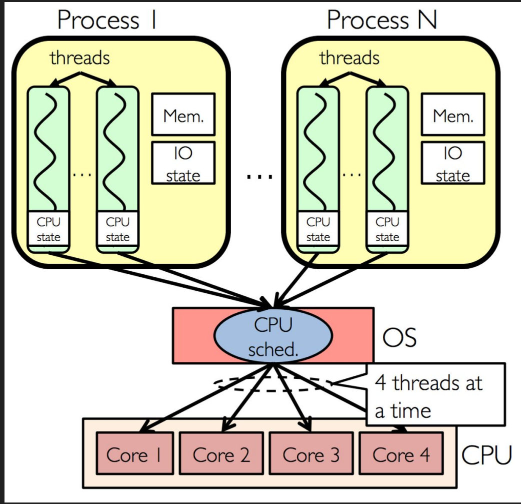
- To implement cooperative multitasking we must:
    1. Write a set of tasks (functions) that we want to execute
    2. Write a scheduler that can schedule tasks from a maintained list of PCBs
    3. Pass the tasks to a scheduler that will maintain PCBs for each task passed to it and construct a sequence of these tasks to execute them in order
    4. Write a `yield()` function that will save the state of the current task in the PCB in the scheduler and resume the state or start the next task in the scheduler
        a. Make sure to put the `yield()` function in each task (function) so it can give time to another task at some point
        b. Make sure to write an `exit()` function in each task so it can communicate to the scheduler when it should be removed from the task list
    5. The scheduler should loop through the list of PCBs until they have all exited

The remaining slides focus on ***threads*** not ***processes***

Threads

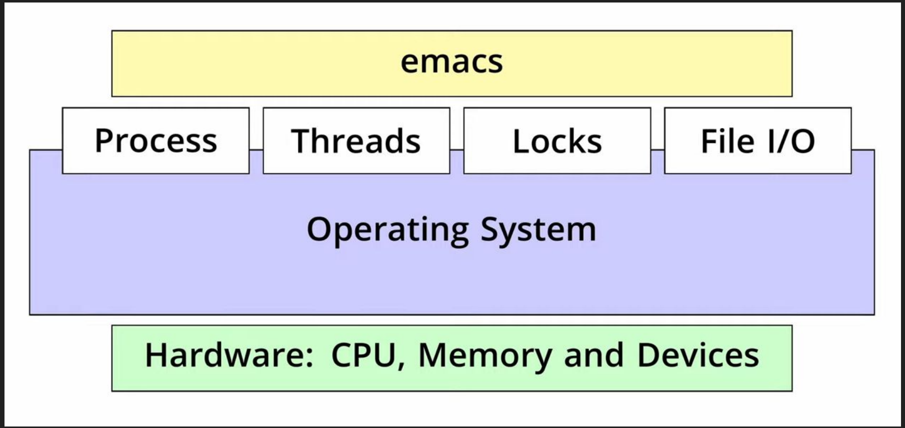
- A thread is a schedulable execution context
  - An execution context is all the information a CPU needs to execute a stream of instructions
- A process may have 1 or more threads
- Multithreaded processes share the same address space
  - All the threads share code, data, and any open files
  - Each thread has its own registers and stack
- Threads can be executed simultaneously by using CPU *cores*
  - A core is the CPU hardware
  - A thread is the instructions/data provided to the core





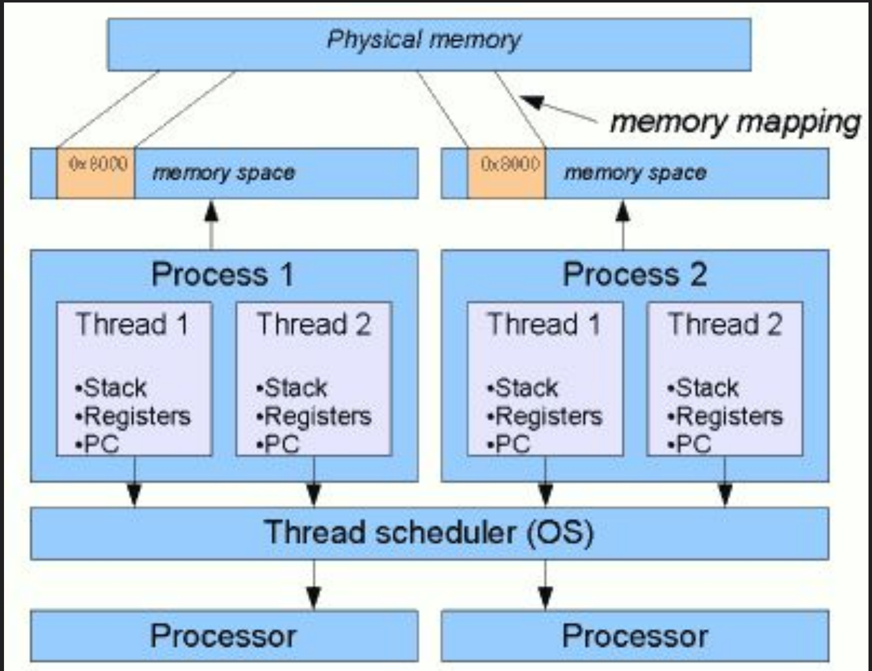
The Thread Abstraction

- Why separate the thread abstraction from the process abstraction?
  - What if you have 4 processes that want 4 threads?
  - What if you have 1 process that wants 4 threads?
- Keeping the threads as a separate system from processes allows for more flexibility
- The kernel usually has its own thread internally for every user mode thread or process
  - This internal thread keeps an eye on the user's processes/threads
  - Also has threads for every user currently logged into the system
- Just like processes, threads must be scheduled by the OS *or* by the process



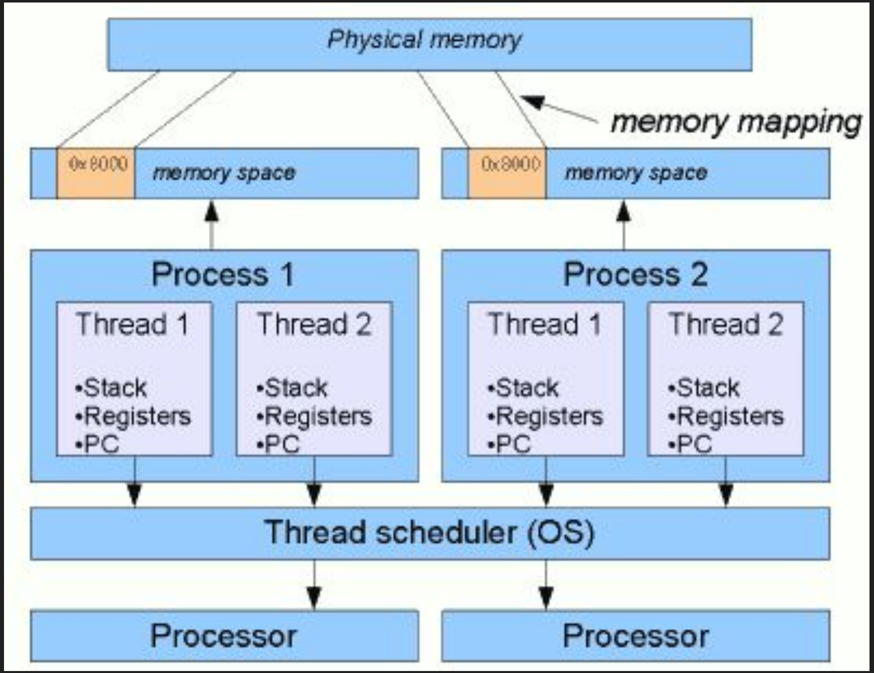
The Thread Abstraction (cont.)

- Threads are great for concurrency
  - Concurrency is executing multiple computations at the same time
  - This requires multiple physical CPU cores
- Threads are lighter-weight than processes
  - Threads share memory, files, etc.
  - Threads are easier to assign rather than spinning up a new process for each execution context
- Threads allow a process to do many things simultaneously
  - Execute code WHILE reading from a file, keyboard, etc.
  - I/O is really slow so this is very beneficial
  - Execute multiple pieces of code for faster results (physics simulations, games, etc.)
- The OS or user processes can create as much threads as they want and are not bound by how many physical cores exist



Multiprocess ≠ Multithreaded

- Threads and processes are two different things!
  - An OS can be multiprocess without being multithreaded and vice versa
- A multiprocess OS can run or switch between multiple processes
- A multithreaded OS can allow processes to use multiple threads
  - As long as there are CPU cores to use



Portable Operating System Interface

- The Portable Operating System Interface (POSIX) is a set of standards used to maintain compatibility between OSs
- POSIX standards allow for many OSs to use standardized methods for threads, I/O, file operations, and other OS functions
- The goal of POSIX is to define both the kernel and user-level application programming interfaces (APIs), along with command line shells and utility interfaces
    - This allows for software to be ported easier to other variants of Unix and other operating systems

Threads in POSIX

- `int pthread_create(pthread_t *thr, pthread_attr_t *attr, void *(*fn)(void *), void *arg);`
  - Create a new thread `thr`, with attributes `attr`, to run a function `fn`, using arguments `arg`
  - If the syntax for `fn` looks confusing, just know it is a pointer to a function, that accepts a pointer as its arguments, and a pointer as its return value, each are void to allow the developer flexibility to choose any type they want to return or pass as arguments
- `void pthread_exit(void *return_value);`
  - Exit or terminate the current thread and return a value `return_value`
- `int pthread_join(pthread_t thread, void **return_value);`
  - Wait for thread `thread`, to exit, and capture its return value `return_value`
- `void pthread_yield();`
  - Tell the OS to allow other threads to execute and pause this thread
  - Helpful when this thread needs to wait for something (like I/O)
- There are more APIs that allow for thread synchronization and other benefits but this is all we care about for now

Implementing POSIX Compliant Threads

- The kernel can implement thread creation using a system call and maintains user threads with kernel threads
  - Every process that wants a thread must use this system call
  - The OS has the final say on whether or not a process gets a thread and controls access to threads
- Implement `pthread_create`, and other thread APIs as a system call
- Create the process abstraction in kernel
- Allow processes to utilize `pthread_create`
- When a process calls `pthread_create`, create a new thread that uses the same address space, file table, code, as the process
- Assigns one kernel thread to this user thread using one-to-one thread model

Kernel Threads vs User Threads

- There are two levels of threads that we use in an OS
- Kernel Threads
  - Managed and scheduled within the kernel
  - Has direct access to the CPU cores
- User Threads
  - Managed and scheduled within the user program
  - Allows multiple parts of the user program to execute simultaneously
- In order for a process to execute, there must exist a relationship between user threads and kernel threads

Contention Scope

- Contention scope is the level at which contention for resources occurs between threads
  - This can be in user space and/or kernel space
- There are two methods for scheduling threads:
  - Process-Contention Scope (PCS)
  - System-Contention Scope (SCS)

Process-Contention Scope

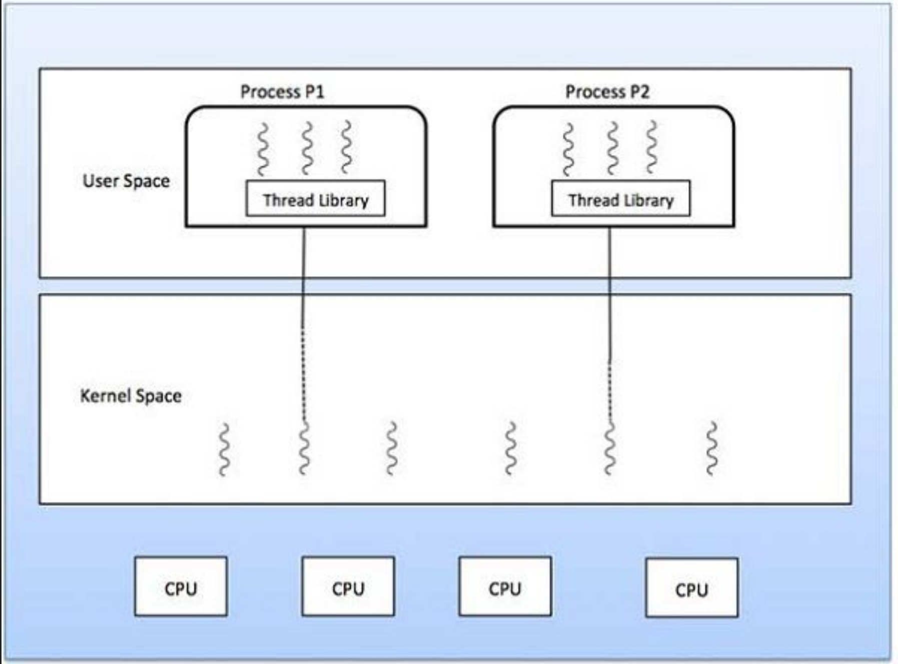
- In PCS, threads within the process compete with each other
- The scheduling mechanism for the thread is local to the process
  - The thread's library has full control over which thread will be scheduled
  - This is typically handled by the programmer assigning priorities to the threads when writing the multithreaded program

System-Contention Scope

- In SCS, threads within the kernel compete with each other
- The scheduling mechanism for the thread is within the OS
  - The OS determines, out of all kernel threads, which get to execute on the CPU(s)
- Many modern OSs only allow for SCS scheduling and the one-to-one thread model

Many-To-One Thread Model

- Each user-level thread is mapped to a single kernel-level thread per process
  - The operating system handles multiple threads as a single task
- This is implemented using a user level library rather than a system call
- If one user thread blocks, the entire kernel thread will become blocked
  - This prevents the other threads from getting a chance to execute
- Lacks true parallelism, the single kernel thread can only execute the user threads on 1 CPU



Implementing User Level Threads with Many-To-One

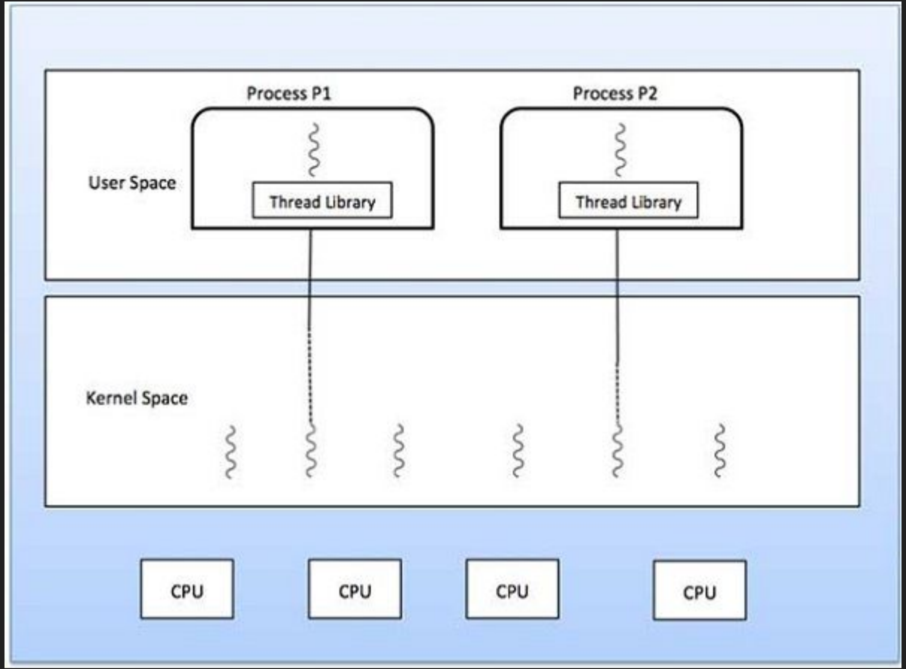
- In Many-To-One the user level threads are required to do a few things:
  - Allocate a new stack for each thread
  - Maintain a queue of *runnable* threads
    - Threads not waiting for I/O or other system calls
- A thread scheduler (running in user space) selects the next runnable thread from the queue and gives it to the kernel thread to execute
- When we want to execute a function using these threads we need to:
  - Write non-blocking versions of any blocking system calls so the thread can yield temporarily, a different thread can run, then resume this thread when done waiting (for I/O, network, storage, etc.)

Problems With User Level Threads in Many-to-One

- Our process only has access to 1 CPU core through the kernel thread
  - The only way to use multiple cores is to have multiple processes running
- Often, it is impossible to write a non-blocking disk read function (OSs don’t often give you the tools to do so)
  - This blocks the thread, waiting for the disk
  - Once the thread is blocked it can’t yield to other threads so the whole process is waiting for that 1 thread!
  - This gets even worse when working with virtual memory
- If one thread blocks another thread, the whole process may get stuck in deadlock

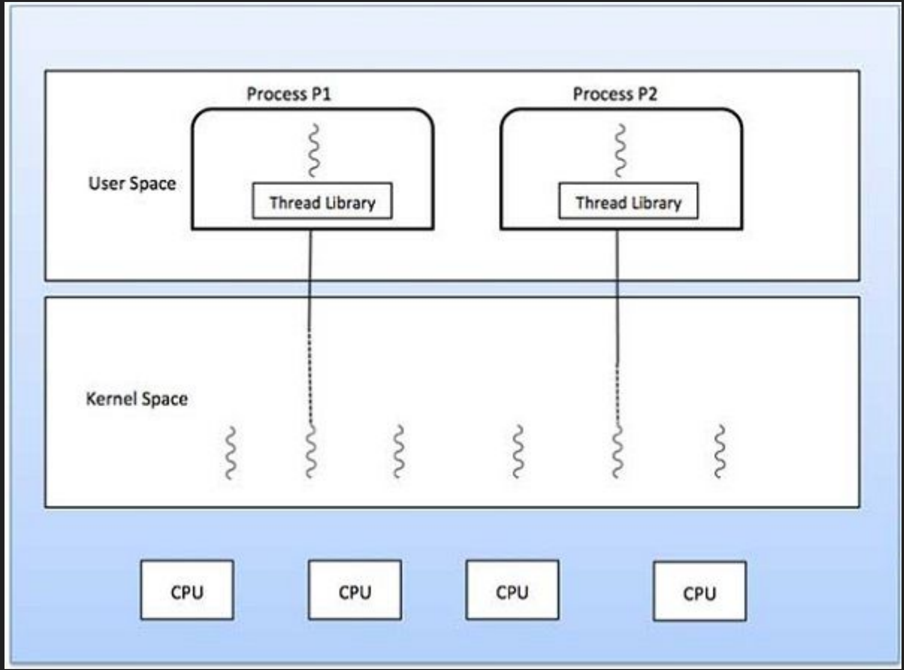
One-To-One Thread Model

- Each user-level thread corresponds to exactly one kernel-level thread
- Creates a direct association with a kernel-level thread managed by the operating system
- Threads can execute simultaneously on multiple processors
- When creating a user thread a kernel thread must also be created
    - This typically results in a lot of memory and processing overhead, creating kernel threads is very slow



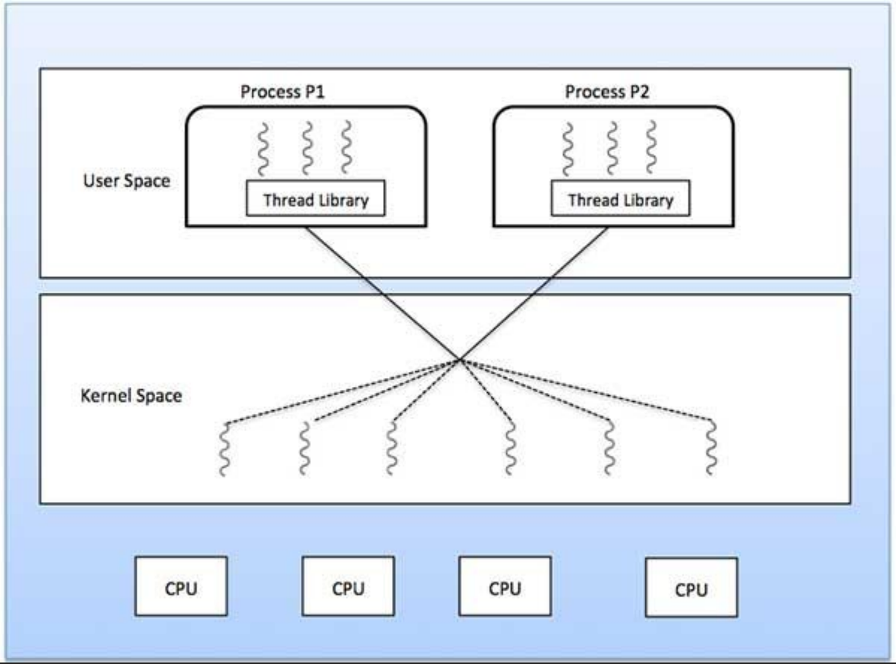
One-To-One Thread Model (cont.)

- What if a process wants 8 user threads on a 4 CPU system?
    - Only 4 can run in parallel
- What if 2 processes want 4 user threads each?
    - Only 1 process can execute its threads in parallel
- The application is required to limit its maximum user threads to the number of cores on the system



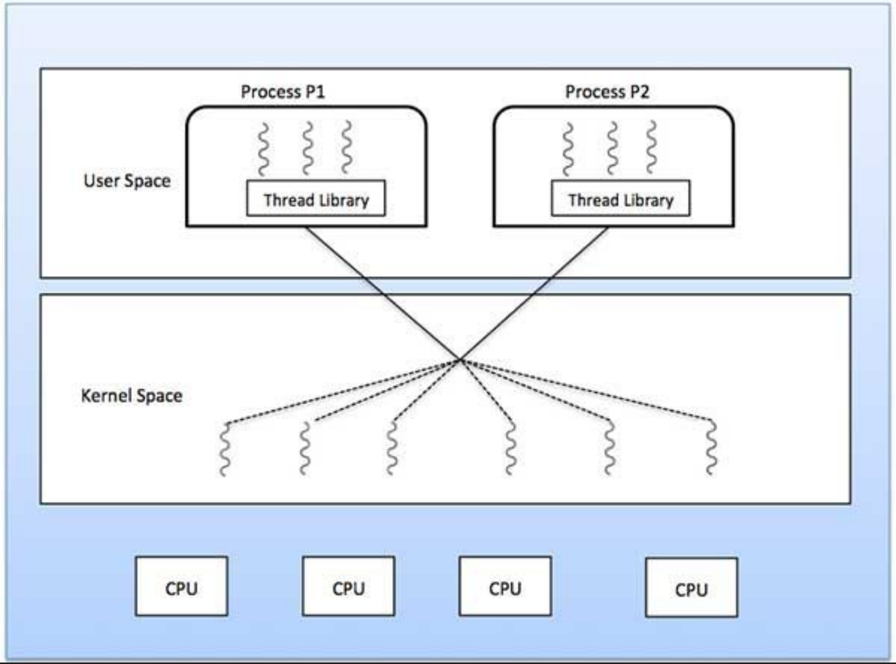
Many-To-Many Thread Model (aka n:m Threading)

- (n) number user-level threads are mapped to an equal or smaller number (m) of kernel-level threads
- Provides a balance between parallelism and efficiency, achieves better performance and flexibility
- User-level threads can run in parallel on multiple processors, and the operating system can manage and schedule a smaller number of kernel-level threads
    - Typically the kernel threads are between 1 thread to the number of cores on the CPU



Many-To-Many Thread Model (aka n:m Threading) (cont.)

- The OS determines how many kernel threads to assign
  - This may be specific to the application or to the hardware
- If a user thread blocks, the other user threads can continue to execute on separate kernel threads
- Many-to-many is not as commonly used due to its complexity



Problems With n:m Threading

- Our processes don’t really know what is happening with the actual CPU cores
  - Only the kernel knows how many CPU cores are available
  - The user space thread scheduler might constantly schedule a user thread that has a blocked *kernel* thread
- The kernel doesn’t know if one user thread is more important than another
  - If one user thread holds an important resource the kernel thread will eventually preempt and lock that thread for some time

Process Scheduling vs Thread Scheduling

- When using a multithreaded OS, threads are typically the main schedulable entity
- Each thread can, generally, be thought of as its own process
- On Linux and POSIX:
  - Threads of a multithreaded process are scheduled independently, rather than the whole process itself
  - Some threads may be allowed to be grouped, or be assigned priority, which may allow all the threads of one process to run simultaneously
  - For single threaded processes, only the single thread is scheduled amongst all the other threads in the system