University College Dublin
An Coláiste Ollscoile, Baile Átha Cliath

Operating Systems

Dr. Vivek Nallur (vivek.nallur@ucd.ie)

Processes

Processes

- Early computer systems executed one program at a time, but all modern operating systems execute many kinds of activities **concurrently**
  - User programs
  - Batch jobs and command scripts
  - System programs

Processes

- A **process** is a program in execution
- The operating system manages most things about processes
  - it creates, deletes, suspends and resumes processes
  - it schedules & manages processes

Process or Program

- A program is a **passive** entity stored on a disk
- A program becomes a process when it is **loaded into memory**
- One program can be **many** processes

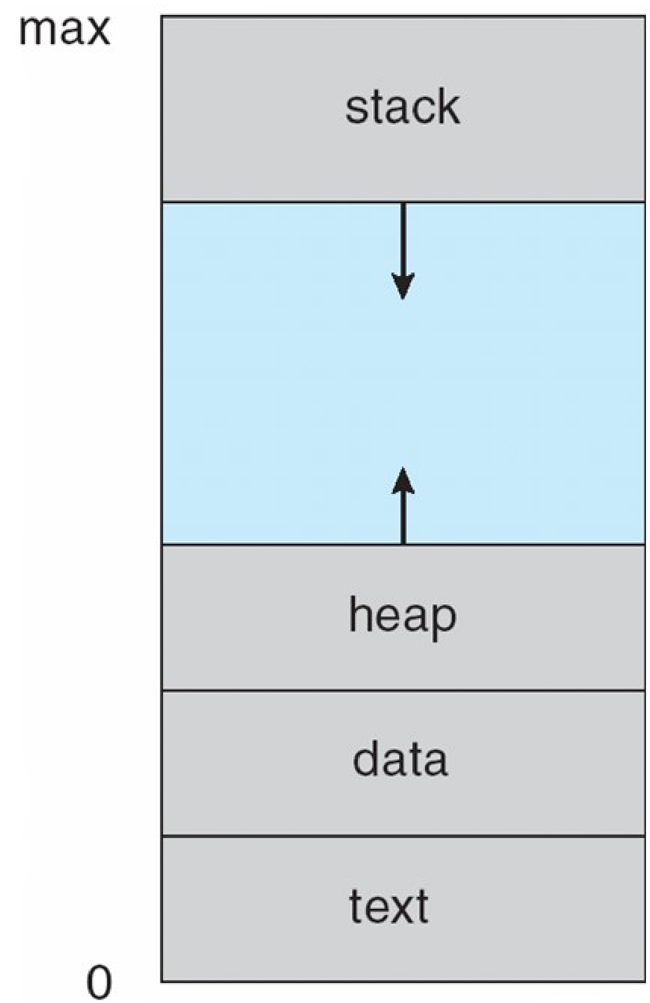
Multiple Processes

- Typical example: Chrome browser
- Each tab is a **separate** process
- They are all treated **independently** by the operating system

What is in a Process?

- The program code, called **text**
- Current activity including **program counter**, processor registers
- Stack containing temporary data
- Data section containing global variables
- Heap containing memory dynamically allocated during run time

Process in Memory



Process in Operating System

- Each process runs in its **own address space** (very important)

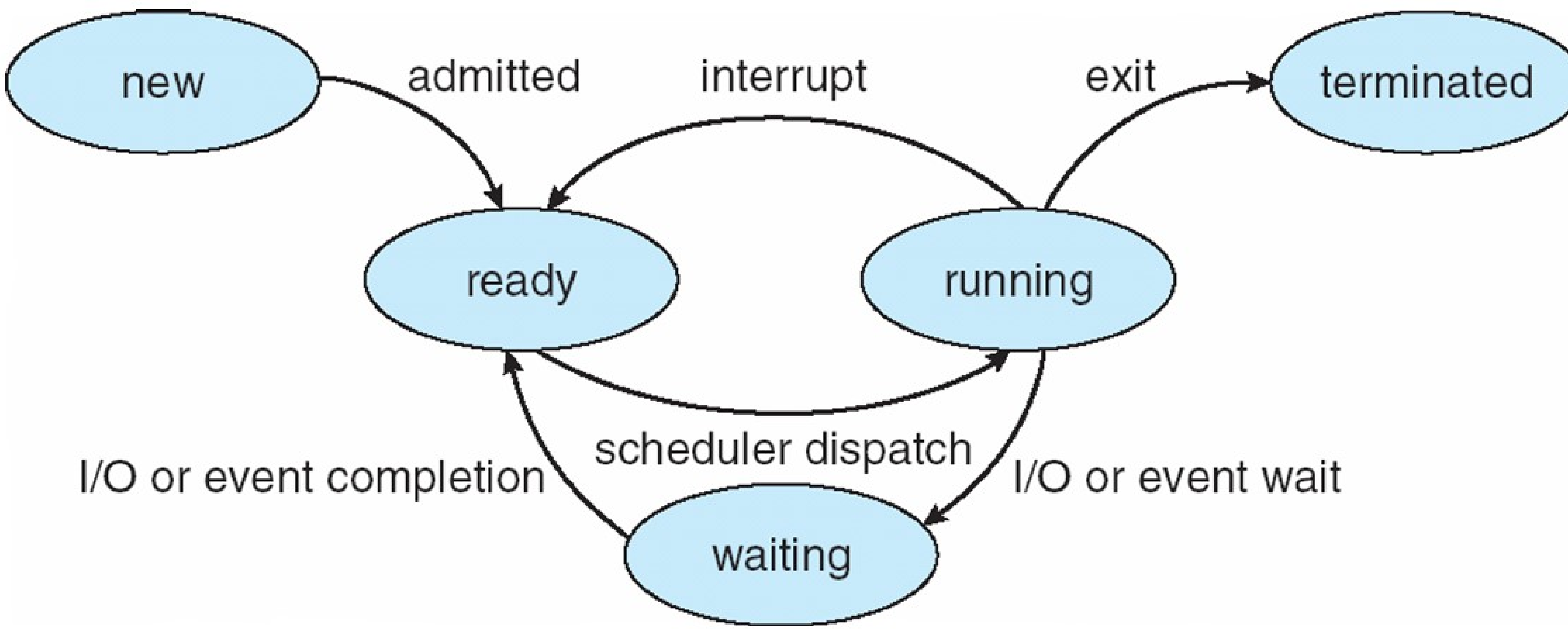
- The **same** address in two different processes will be stored in two **different** locations in memory

Process States

- As a process executes, it can be in a number of different **states**

- **New:** The process is being created
- **Running:** Instructions are being executed
- **Waiting:** The process is waiting for some event to occur
- **Ready:** The process is waiting to be assigned to a processor
- **Terminated:** The process has finished

Process States



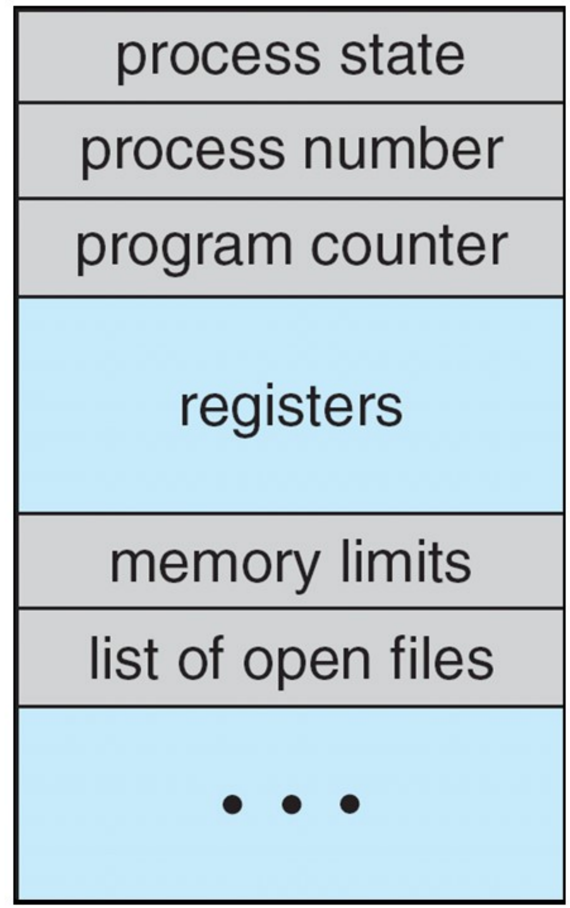
Process Control Block (PCB)

- PCB contains information associated with each process (also called task control block)

- The OS keeps a either a system-wide or a per-user process table
  - each entry contains a PID and a pointer giving the address of that process's PCB in memory

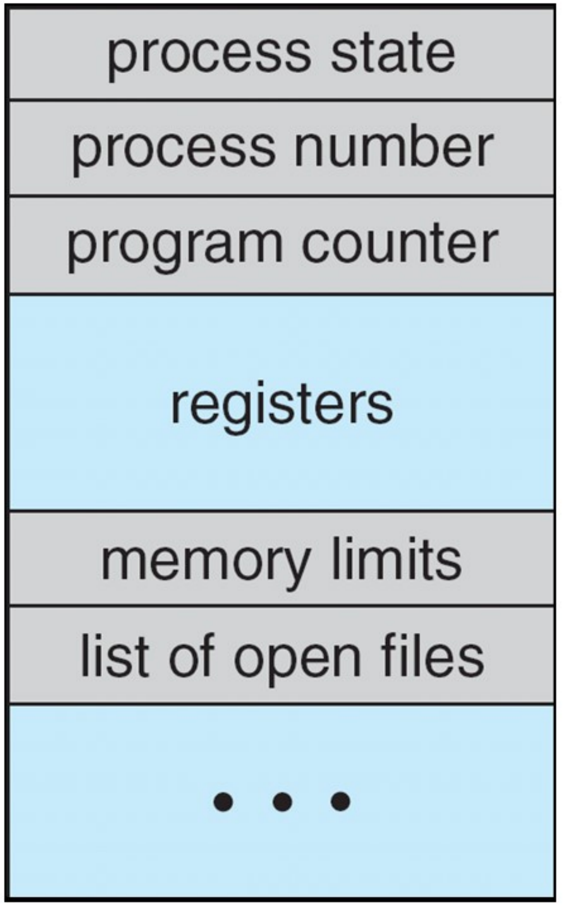
Process Control Block (PCB)

- Process state: running, waiting etc.
- Program Counter (PC): location of the next instruction to execute
- CPU registers – contents of all registers for this process



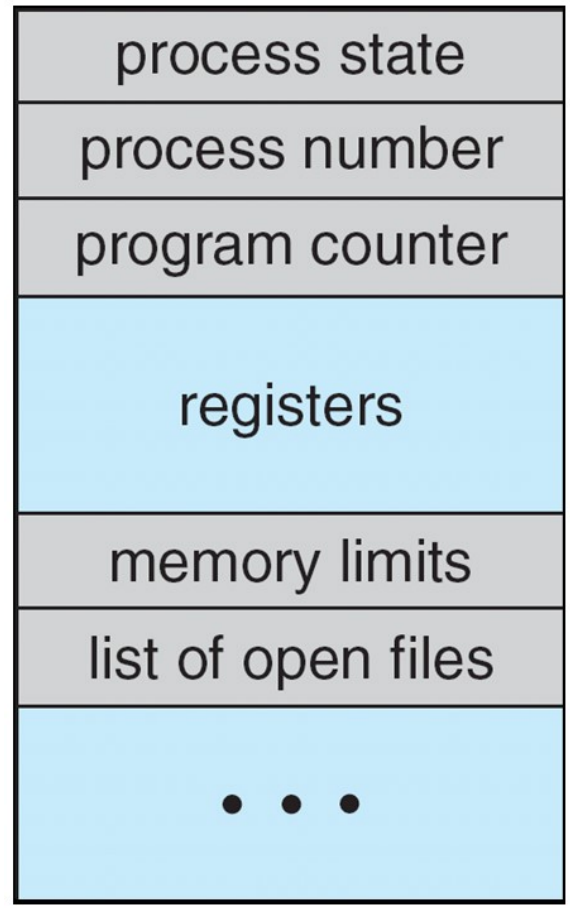
Process Control Block (PCB)

- CPU scheduling information- priorities, scheduling queue pointers
- Process number: also called PID
- Memory-management information – memory allocated to the process



Process Control Block (PCB)

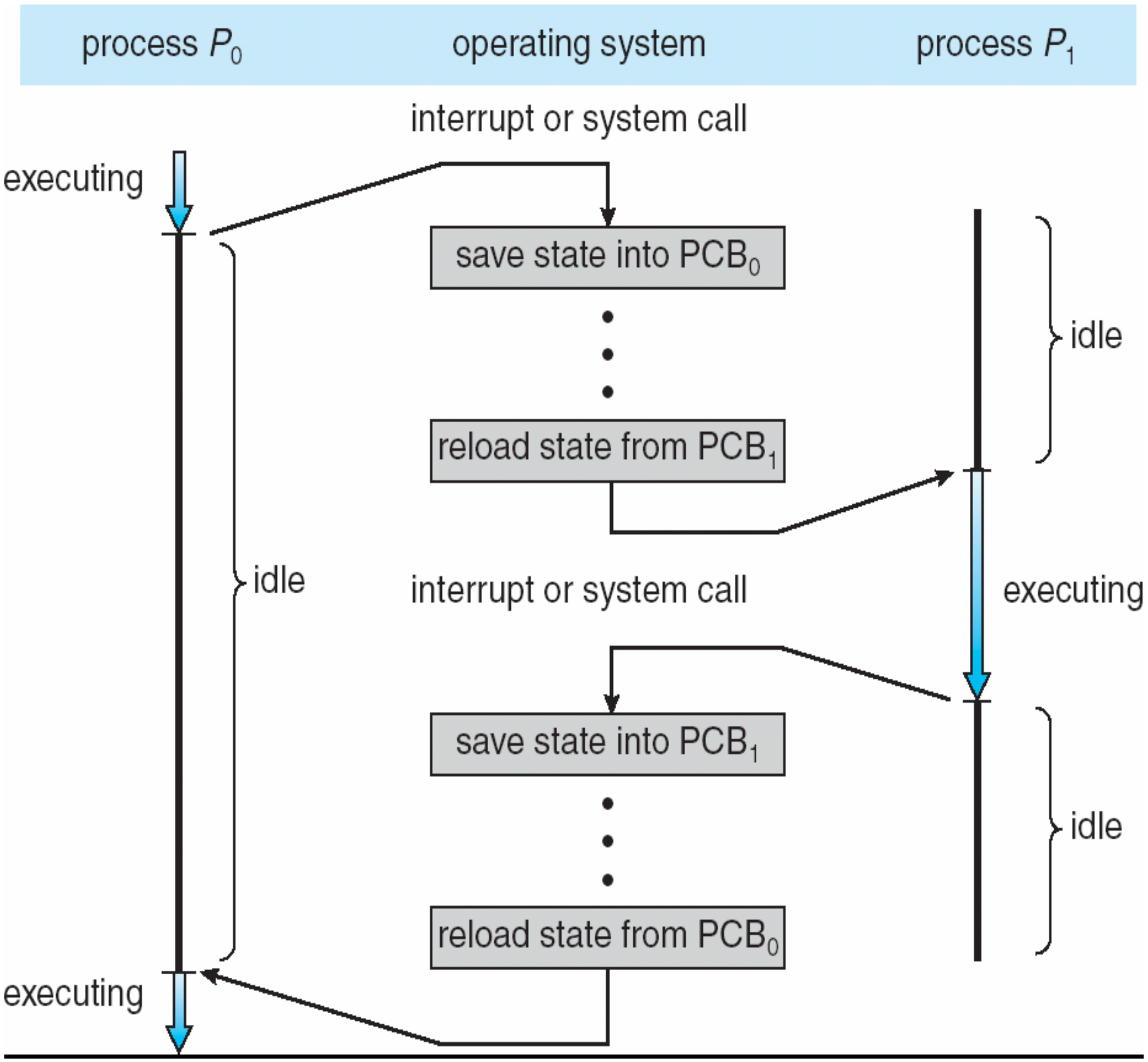
- Accounting information: CPU used, clock time elapsed since start, time limits
- I/O status information: I/O devices allocated to process, list of open files



Process Switching

- Using the information stored in the PCB the OS can easily save and load the state of a process

- This allows processes to be easily switched; this is called a **context switch**


process P_0
operating system
process P_1
interrupt or system call
executing
save state into PCB_0
idle
reload state from PCB_1
idle
interrupt or system call
executing
save state into PCB_1
idle
reload state from PCB_0
executing

Overhead in Context Switch

- While a context switch is happening, the system is not doing any work

- The time taken is considered the **overhead** of the operation

Overhead in Context Switch

- To minimise the amount of time wasted context switches must be fast (hardware dependent)
- Also we don't want to switch too **much** (too much overhead) or too **little** (not interactive enough)

Process Creation

- Processes are created by two main events:
  - System boot
  - Execution of process creation system call by another process

Process Termination

- Processes are terminated in different conditions:
  - **Voluntary**: normal exit, error exit
  - **Involuntary**: fatal error, killed by another process

Process Termination

- Voluntary termination can **only** happen from the running state

- Involuntary termination can happen from **any** state

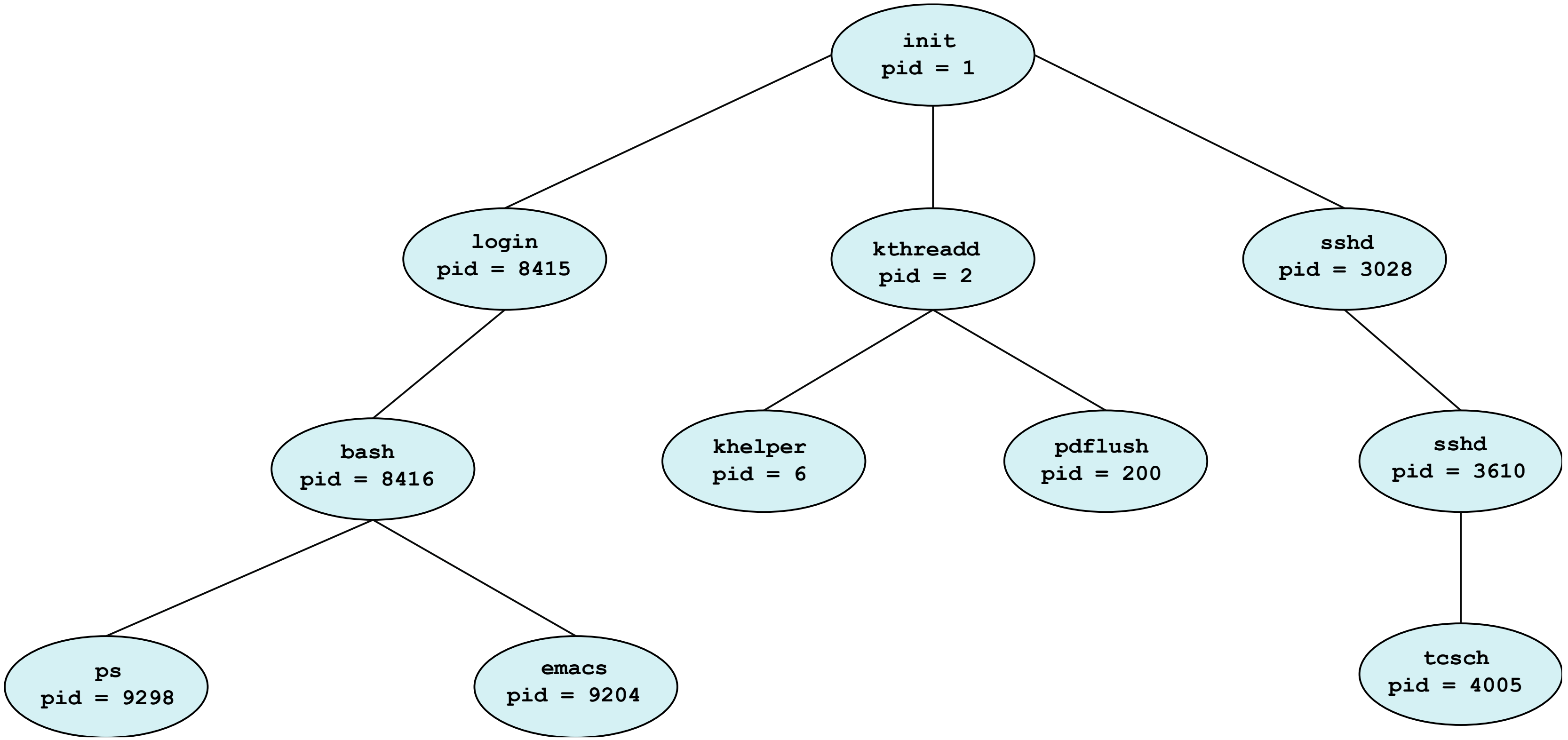
Child Processes

- A process can spawn (create) new processes
  - The creator process is called the **parent**
  - The created process is called the **child**

Child Processes

- A hierarchical process structure is created in this way (any child process only has one parent), called **process tree**
- In some OSs, stopping the parent stops all spawned children
- An **orphan process** is a computer **process** whose parent has stopped but the process remains running.
- Normally these processes are adopted by another parent process
- If this does not happen , they become a **zombie** process
- They should then be removed or **reaped** from the system
- A failure to reap a zombie process is normally due to a bug in the operating system.

Process Tree



Example (Unix)

- A child process is created using the **fork**() system call
- The child receives almost everything from its parent
- In essence the parent address space & PCB are **copied**
- One attack on Unix is to create a fork bomb where a process infinitely attempts to create children who then in turn try and create more children until all system resources are used up.

Example (Unix)

- The child process must have a new **PID**, and will have different pointers for its parent/child processes
- Because the PCB is copied the child process begins execution **after** the fork() instruction

Example (Unix)

- The fork system call returns an integer value
- To the parent it will return the **PID** of the new child process
- To the child it will return **zero**

Threads

Interprocess Communication (IPC)

- Processes within a system may be **independent** or **cooperating**
- Cooperating process can affect or be affected by other processes, including sharing data

Independent Processes

- Independent processes are those that can neither affect nor be affected by the rest of the system
- Two independent processes cannot share system state or data
- Example: processes running on different non-networked computers

Properties of Independent Processes

- **Deterministic** behavior: only the input state determines the results
  - This means the results are **reproducible**
- Can be stopped and restarted without causing problems

Cooperative Processes

- Cooperative processes are those that share something (not necessarily for a purpose)

- Two processes are cooperative if the execution of one of them may affect the execution of the other

- Example: processes that share a single file system

Properties of Cooperative Processes

- **Nondeterministic** behavior: many factors may determine the result

- This means results may be difficult to reproduce

- This makes testing and debugging very difficult

Properties of Cooperative Processes

- Cooperative processes are subject to **race conditions**

- This means that the result of the process may depend on the sequence or timing of events in other processes

Why Allow Processes to Cooperate?

- Resources and Information sharing
  - Multiple processes can share a single resource and communicate
- Convenience
  - We can do things like editing a file at the same time it is being printed

Why Allow Processes to Cooperate?

- System speed-up, by introducing concurrency into program execution
- We can overlap I/O with computations
- The multiplication of two $n \times n$ matrices can be divided into n^2 independent subtasks

Processes and Cooperation

- Multiple concurrent cooperative activities necessarily happen in an OS
- Why not define each and every one of these concurrent activities within a different process?
- Unfortunately processes are **not ideal for cooperation**

Issues with Processes

- Processes are not very efficient:
  - creation of a new process is costly
  - all the process structures must be allocated upon creation

Issues with Processes

- Processes don’t (directly) share memory
  - Each process runs in its own address space
  - But parallel and concurrent processes often want to manipulate the same data
  - Most communications go through the OS: slow

Motivation for Threads

- Consider a process that is running a file server
- Occasionally it will have to wait for the hard disk to respond
- During this time the process will be blocked and unable to respond to new requests

Motivation for Threads

- In order to speed up future operations, the process will keep a **cache** of recent files in its memory

- A good idea would be to run a second concurrent file server to work while the other waits

Motivation for Threads

- However it is not possible to efficiently achieve concurrency by creating two independent processes

- They would have to run in **the same address space** to efficiently share a common cache

Solution: Threads

- The idea is that there is more than one **active entity** (thread of control) within a single process

File Server Example

- Considering our file server earlier
  - The probability of a file being in cache is .6 and the operation takes 15ms
  - The probability of a file being on the disk is .4 and the operation takes a further 75ms
- What is the maximum number of requests that can be handled per second?

File Server Example

- Single Threaded: Average Time
  - .6 x 15 + .4 x (15 + 75) = 45ms (22.22 requests/s)
- Multi Threaded: additional threads can be started when files are in disk without having to wait for the I/O operation to complete
  - The average time to handle a request is 15 ms (66.67 requests/s)

Threads

- Modern OSs support both entities (process & thread): **multi-threaded OS**
    - Process: defines the address space and general process attributes
    - Thread: defines a single sequential execution stream within a process

Concurrency in some existing OS

- MS-DOS: one address space, one thread
- Unix (originally): multiple address spaces, one thread per address space
- OSX, Solaris, Windows 10: multiple address spaces, multiple threads per address space (multi-threading)

Threads & Processes

- All threads belonging to a process share almost everything in the process:
  - Address space (code and data)
  - Global variables
  - Privileges
  - Open files
  - Timers
  - Signals
  - Semaphores
  - Accounting information

Threads & Processes

- Threads however **do not share**:
  - Register set, in particular:
    - Program counter (PC)
    - Stack pointer (SP)
    - Interrupt vectors
  - Stack
  - State
  - Child threads

Threads & Processes

- Threads do not exist on their own, they belong to processes
  - There must always be at least one thread

- Threads are cheap to create (no need to allocate PCB, new address space)

Threads & Processes

- Threads can communicate with each other efficiently through the process global variables or through common memory, using simple primitives
- Threads facilitate concurrency, and therefore are useful even on uniprocessor systems

Threads & Processes

- Threads can be created statically or dynamically (by a process or by another thread)
- If a thread needs a service provided by the OS (system call) it acts on behalf of the process it belongs to

Thread Implementations

- Threads can be implemented in two basic ways
  - Threads in user space (many-to-one model)
  - Threads in kernel space (kernel threads, one-to-one model)
  - Additionally it is possible to combine the two approaches in the Hybrid (many-to-many) model

Threads in user space

- The kernel schedules **processes** (does not implement or know about threads)

- **per process thread table**: process decides which of its threads to run when it is running

Advantages of Threads in user space

- a single-threaded OS can emulate multi-threading
- thread scheduling controlled by run-time library, no system call overheads
- portability: a OS independent user-space threads library is possible

Disadvantage of Threads in user space

- If a thread blocks, all other threads belonging to the same process are blocked too

- This is because the OS only schedules processes

Threads in kernel space

- The kernel schedules threads
- In this case threads (and not processes) are the smallest units of scheduling
- **System-wide thread table** (similar to a process table)

Advantages of Threads in kernel space

- Individual management of threads

- Better interactivity

Disadvantages of Threads in kernel space

- Scheduling and synchronisation operations always invoke the kernel, which increases overheads

- Less portable

That's all, folks!

- Questions?