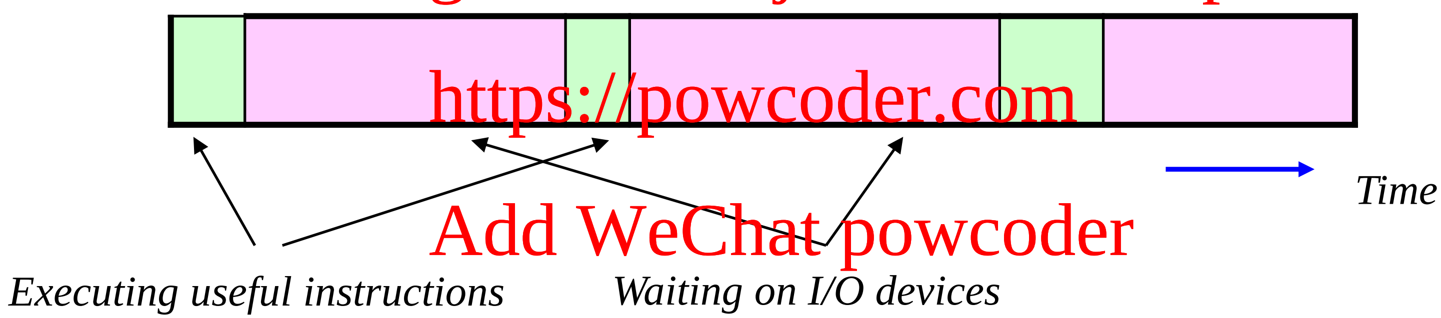
9. Operating Systems.

Operating Systems

- Purpose: to provide high level facilities beyond the capabilities of the hardware
- To provide security for data
- An operating system is really a collection of a large number of services.
  - Some are essential and are provided always;
  - Others are available to an application via explicit request (usually via an instruction like TRAP)
- Examples: Windows, Unix, mainframe systems, real-time systems.
- In general modern operating systems are big
  - E.g. Windows 50,000,000 source lines of code.

Historical Motivation: Buffered I/O

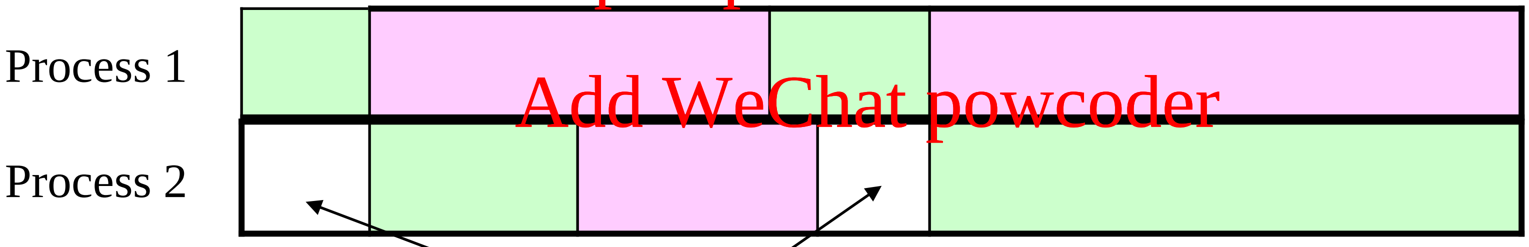
- The problem: calculations are fast, I/O is usually slow
  - E.g. reading a character of input is much slower than executing a statement like x = 2*3
  - Therefore, if we just have a bare machine running a program, most of the time will be spent waiting on I/O, with little useful work getting done



- An idle CPU is a wasted resource. Solution: find it something else to do!
- *Example*: when a computer prints a document, it doesn't just freeze up until the paper is sitting there with ink on it! It continues to perform other tasks.

A Solution: Concurrent Processes

- Introduce the idea of processes
- A process is a running program, with its evolving state and dynamic behaviour
- It seems like several processes can run at the same time on a single CPU. This is called multi-tasking.
  - In reality, the OS uses timer interrupts and time slices to implement multitasking.



Ready*—not actually executing, but ready to go when CPU becomes available*

Two activities can make progress; neither is frozen because of the presence of the other

The OS Controls Processes

- There may be many independent processes running concurrently. But:
  - When waiting for I/O how would an application know which other application to jump to?
  - How does an ordinary application even know what programs are running?
- If we want multitasking:
  - What about I/O devices which multiple processes may need (screen, keyboard, printer etc.)?
  - How do we stop two independent processes from over-writing each other's memory?
- Solution: the Operating System (OS)
  - Maintains a table of all running processes
  - Mediates every process break, deciding what to do next
  - Controls which process gets to access which I/O devices at any given time
  - Controls access to physical memory with the help of a hardware device capable of translating outgoing addresses (a memory management unit or MMU).
- Process control is the most fundamental facility provided by operating systems
- It's implemented by the most central code in the OS, called the kernel.

Multitasking on one CPU

- At any moment in time, the CPU is executing just one instruction!
  - This belongs either to one of the processes, or to the operating system
  - One process is running, the others are waiting
- CPU switches execution between the processes rapidly (~ 100 times per second)
- User's point of view:
  - Computer responds smoothly to keystrokes and mouse motions
  - In fact CPU is running other processes most of the time.
- Computer's point of view
  - Rapidly alternates attention among many processes
- The Operating System maintains a list of programs running at the same time
  - What does "at the same" time mean? Depends on time scale:
    - *At any given instant*, hardware is executing one instruction belonging to one program
    - *Over a longer period of time,* as interrupts keep transferring control between programs, all programs get some execution time.

Processes

- A ***program*** is a document: a text file containing a source program, or a binary file containing a machine language program.
  - A program just sits there, like a book.
- A ***process*** is a running program. May not be executing all the time but soon an interrupt will cause the computer to resume its instructions
- At any moment in time, each process is in one of the following states:
  - Running: at this very nanosecond, the digital circuitry in the computer is executing an instruction belonging to the process
  - Blocked: the process is waiting for I/O
  - Ready: the process is not waiting for I/O, so it could be running, but the CPU is doing something else

Concurrency and Parallelism

- Related but distinct concepts that are often confused
- Several processes run concurrently when all make progress on user's time scale
  – Can involve parallelism, but usually just the CPU switching rapidly among the processes.
- Processes run in parallel when they execute simultaneously.
  – Requires multiple processors, used for high performance applications
  – Many modern microprocessors have multiple cores: these are essentially multiple CPUs on one chip and can genuinely execute processes in parallel.
  – An individual core can itself run multiple processes concurrently (not truly in parallel).
- In order to get smooth concurrency on one CPU, must switch rapidly and regularly among the processes even if the running process has not requested I/O.
  – If we don't, computer won't respond smoothly to user requests.
  – Even with multiple CPUs, there are usually not enough for all processes so, each must also switch processes rapidly and regularly.

Co-operative & Pre-emptive Scheduling

- Simplest approach is co-operative scheduling:
  - All applications must be written so as to voluntarily give up control to the OS at regular periods.
  - The relinquish operation is called a process break and accesses via a TRAP instruction with appropriate parameter.
  - If an application fails to perform a process break, it gets all the machine's resources. All other processes are starved.
  - This is a poor solution but needs no extra hardware
  - A process break is meant to suspend a process temporarily not terminate it.
- Much better is pre-emptive scheduling.
  - Hardware and OS work together to enforce a policy of compulsory regular process breaks.
  - The hardware must have the ability to signal to the CPU at any given time that it must stop its current sequence of instruction cycles.
  - This is provided via special control inputs to the CPU called interrupt lines.
  - Needs extra hardware such as timer.

Physical I/O

- I/O devices have control/status registers as well as data registers.
  - These locations may appear in the normal physical address space and are accessed by usual read and write operations (memory mapped).
  - Alternatively they may be accessible by special I/O instructions.
  - Some systems use both approaches (e.g. a PC graphics card)
  - Storing data in a data register for an output device sends the word to the device
  - Reading data from the data register for an input device accepts data from the device.
- Some I/O devices can be shared between several applications at once; others can only service one at a time.
- Often I/O devices are slow, so process requesting I/O may have to wait.
- A process may wait by looping and polling the device (busy wait) or it may be notified when request completes. In latter case it may suspended by OS (synchronous I/O) or it may carry on running (asynchronous I/O).
- A suitable OS can relieve applications of responsibility for directly controlling I/O devices and can ensure that each device is shared appropriately between applications.
- When I/O is under OS control, application will normally access a device via high-level commands. These are translated into physical access by an OS device driver.

Interrupts

- A hardware interrupt is a feature that allows an external hardware device to indicate to a CPU that it should perform a process break.
- When an interrupt occurs the CPU is directed to an interrupt handler routine which has been prepared in advance and which it begins executing.
  - As with a subroutine, the return address must be saved so that the CPU can return to the interrupted process when ready.
  - Except in the very simplest systems, the interrupt handler routine is in the OS and the OS decides what to do next
  - An interrupt can occur at any time but the CPU *always finishes the current instruction before jumping to the handler*.
- Interrupts may be generated at arbitrary intervals by I/O devices that need service or regularly by a system timer (see next).

Interrupt Handling

- In a pre-emptively scheduled system a special hardware timer is used to generate interrupts at regular intervals (e.g. every 100ms).
  - This invokes a special interrupt routine inside the OS called the process scheduler.
  - The process scheduler uses a scheduling algorithm to choose which process goes next if more than one is ready to run (this may include some concept of process priorities).
  - No process gets more time than the period of the timer (the time slice).
- Interrupts are not initiated by a process itself and can occur at the end of any instruction.
- An interrupt is only supposed to suspend a process not terminate it.
- In order to resume a process from any arbitrary point, all registers in the programmer's model must be saved when an interrupt handler is invoked. Why just the programmer's model?