## Lecture Overview

- Quick review of topics covered so far
  - Computer hardware
    - CPU, instructions, registers
    - Memory
    - I/O devices
  - Operating systems
    - Processes
    - Concurrency
    - Threads and lightweight processes



## Review of Lectures So Far

- Computer hardware
  - In general, we can think of the CPU as a small, self-contained computer
    - It has instructions for performing mathematical operations
    - It has a small amount of storage space (its registers)
    - We can feed instructions to the CPU one at a time and use it to perform complex calculations
      - This is the ultimate in "interactive" operation; the user does everything
      - It would be better if there was some way to give the CPU a lot of instructions all at once, rather than one at a time

## Review of Lectures So Far

- Computer hardware (con’t)
  - We need to combine the CPU with RAM and a memory bus
    - The bus connects the CPU to the RAM and allows the CPU to access address location contents
    - Since we are going to load many instructions (i.e., a program) into memory, the CPU must have a special register to keep track of the current instruction, the *program counter*
      - The program counter is incremented after each instruction
      - Some instructions directly set the value of the program counter, like `JUMP` or `GOTO` instruction

## Review of Lectures So Far

- Computer hardware (con’t)
  - We need to combine the CPU with RAM and a memory bus (con’t)
    - By adding memory we must extend the operations that the CPU needs to perform, it needs instructions to read/write to/from memory
    - We can use memory for two purposes now
      - Storing instructions (the program code)
      - Storing data
    - This doesn’t allow us to interact with the program and memory is still pretty expensive for its size

## Review of Lectures So Far

- Computer hardware (con’t)
  - Now we add I/O devices to the communication bus
    - The CPU communicates with I/O devices via the bus
    - This allows user interaction with the program (e.g., via a terminal)
    - This also allows more data and bigger programs (e.g., stored on a disk)

## Review of Lectures So Far

- Computer hardware (con’t)
  - Up until this point we have described what amounts to a simple, but reasonable computer system
    - This system stores programs and data on disks
    - It executes a one program at a time by loading a program’s instructions into memory and sets the program counter to the first instruction of the program
    - A program runs until completion and has complete access to the hardware and I/O devices
    - *There really isn’t much of an operating system and no such thing as a process*
  - This is good, but a lot of the time the CPU is just sitting around with nothing to do because the program is waiting for I/O

## Review of Lectures So Far

- Computer hardware (con't)
  - Since the CPU is much faster than the I/O devices it has three options when performing I/O
    - It can simply wait (not very efficient)
    - It can poll the device and try to do other work at the same time (complicated to implement and not necessarily timely)
    - It can allow the I/O devices to notify it when they are done via interrupts (still a bit complicated, but efficient and timely)
    - *The last two options require a sophisticated OS, we will focus on the last option*

## Review of Lectures So Far

- Providing an Operating System
  - An OS could better utilize our CPU if we could run more than one program at once
    - *Multiprogramming* - executing another program when the current program blocks
    - *Time-sharing/multitasking* - executing one program for a short period of time and then switching quickly to another and so on
  - This introduces the notion of a *process* (i.e., an executing program)

## Review of Lectures So Far

- Providing an Operating System
  - An OS must define some way to stop running the current process and start running another, there are two options
    - Implement all I/O calls to give up CPU when they might block and provide functions to yield the CPU voluntarily; this is *cooperative multitasking*
    - Add a *hardware timer interrupt* to our CPU so that we can automatically interrupt processes after some amount of time; this is called *preemptive multitasking*
  - Now that we have multiple processes running, we need some way to protect the OS from them and them from each other
    - Hardware support in the form of *dual-mode* CPU operation
      - This means that some instructions can only be executed by the OS and not by processes

## Review of Lectures So Far

- Providing an Operating System (con't)
  - On a uniprocessor computer, a process can only make progress when it has the CPU and only one process can have the CPU at a time
    - This means that only one process is actually executing at a time on a uniprocessor computer
  - The OS must share the CPU among all processes so that all process can get a chance to execute
  - How does the OS share the CPU among multiple processes?
    - It *preempts* the current process (or the current process cooperatively blocks) and the OS chooses another process for the CPU

## Review of Lectures So Far

- Providing an Operating System (con't)
  - What does the OS do when it preempts a process?
    - Saves the CPU registers for the current process since they contain unfinished work; the CPU registers are saved in the *process descriptor* in OS's process table
      - The process descriptor keeps track of all process information for a specific process
    - Saves the program counter in the process descriptor so it knows where to resume the current process later
  - What does the OS do when it gives the CPU to a process?
    - Restores the process' CPU registers from the saved values in the process descriptor for that process
    - Restores the program counter to the next instruction for the new process

## Review of Lectures So Far

- Providing an Operating System (con't)
  - The OS must also share other resources, such as
    - Memory
      - Make sure that each process has its own address space
        - » This is not a physical address space, but a logical one
      - Uses the process descriptor to keep track of the memory that a process is using
    - I/O devices
      - Uses wait queues to allow access to devices
      - Uses the process descriptor to keep track of various I/O resources, like file descriptors

## Review of Lectures So Far

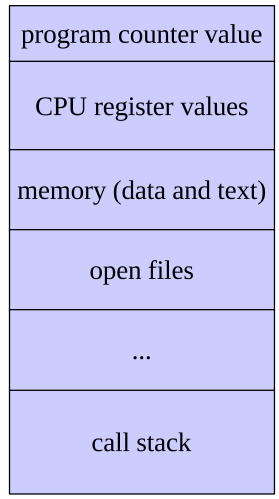
- Providing an Operating System (con't)
  - We now have created a multitasking OS
  - Is it a *concurrent* system? *Yes, in computer science terms.*
    - English definition of "concurrent"
      - Happening at the same time as something else
    - Computer science definition of "concurrent"
      - Non-sequential execution (non-deterministic)
    - Definition of "parallel"
      - Happening at the same time as something else
      - This is the same as the English meaning of "concurrent"
    - In computer science something that is parallel is also concurrent (i.e., non-sequential), but something that is concurrent is not necessarily parallel

## Review of Lectures So Far

- Defining a process
  - An executing program
  - This means that the process must contain
    - A program counter value
      - This keeps track of the next instruction to execute and must be saved in the process descriptor when the process loses the CPU
    - All CPU register contents
    - Call stack
    - Open files
    - Memory (including actual program text/code)
    - Any other resources owned by the process
  - All of this stuff is in the process descriptor

## Review of Lectures So Far

- Defining a process
  - A process is a resource container with a single execution flow

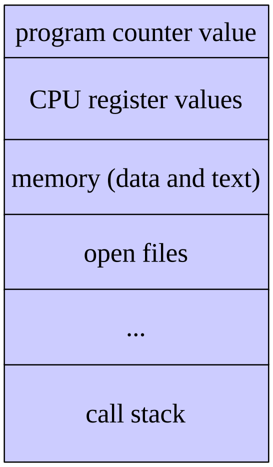



## Review of Lectures So Far

- Defining a thread
  - It is possible to conceptually break a process into two distinct, but separate notions
    - A resource container
    - An execution flow
  - After making this conceptual division, we call the resource container a *process* and the execution flow a *thread*
  - A thread cannot exist without a process, thus processes are then a “container” for threads
  - It is possible for multiple threads to exist in the same process
  - A process with a single thread is the equivalent of our original definition of a process

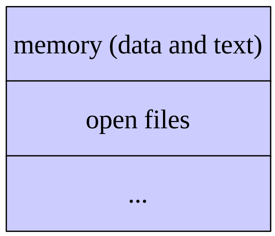
# Review of Lectures So Far

- Defining a thread

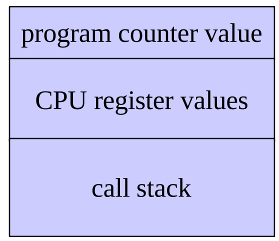


Original process

  - Since multiple threads can exist in the same process, this means that they can share lots of stuff, but not everything



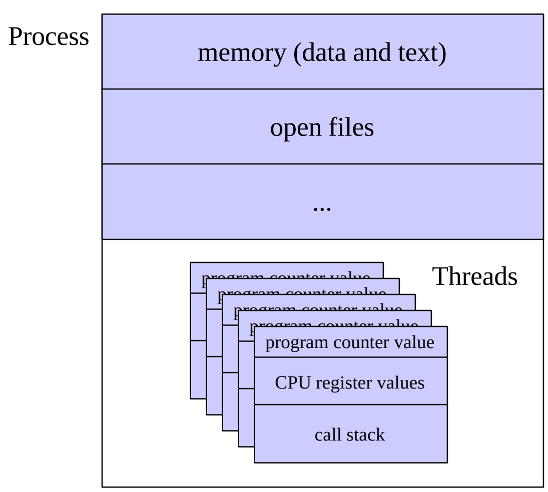
New process



Thread

# Review of Lectures So Far

- Defining a thread



## Review of Lectures So Far

- Defining a lightweight process
  - Sometimes the dividing line between a process and a thread is very thin
  - A lightweight process is pretty much the same as a normal process except that it may share some resources with other lightweight processes
    - In this regard a lightweight process is very much like a thread and can be used to implement threads
  - Linux uses lightweight processes to implement threads, which is why you can see the threads as processes when you list process with the `ps` command
  - Not every lightweight process is a thread