Work.

15-440
Carnegie Mellon University
Distributed Systems

Project Notes

- What algorithm should I use for <foo>?
- A: We don't really care. Just use a reasonable one.
- A': Butler Lampson's "Hints for computer system design" (great paper) has a design hint:
  "When in doubt, use brute force."
  Think about d(technology)/dt. Simple, well understood algorithms are a good way to start. You can always optimize later if time doesn't save you.
- And we really, really don't want to see code-level micro-optimization that impairs readability.

- You have some computational "work" to do.
- How do you think about this work;

  what are the units in which it runs?

  the abstractions you use in a computer or distributed system to represent it?
- How do I divide up work in a distributed (parallel) system?

Today

- May seem pretty "definitional"
  (Other lectures will have a lot more "how to do cool stuff")
- No apologies: We need clear, precise definitions to understand, communicate, and build systems.
- Analogy: A computer will do exactly what you tell it. But you have to know exactly how to express what you want...
  - So first, you have to *know* exactly what you want
- This happens everywhere - ugrad, grad school, and beyond. Clear definitions are necessary for clear thought.
- A challenge: "Systems contain subsystems that are themselves systems" (S&K) -- aka, system decomposition is recursive.

Jobs: Chunks of work

- A *Job* (n): A task that is performed as if it was a single logical unit

  Remember, our definitions have to operate at multiple levels of abstraction

  Example:
  From the perspective of your password cracker server, cracking one password is a job. (Batch processing has similar views)

  From the client application's perspective, cracking a range is a job.

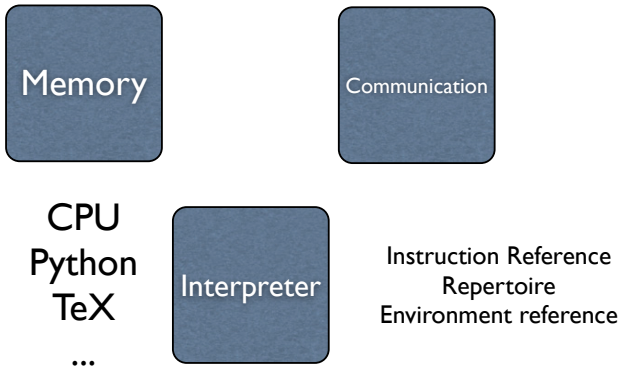
  From the OS kernel's perspective, a job is the granularity at which threads are scheduled (a burst of activity)

Let's examine representations of these from bottom-up

Tasks

- Set of instructions
- Usually, a "task" is more generic than a "process" or a "thread" (which have specific extra stuff with them)

Machine organization



The CPU (etc)

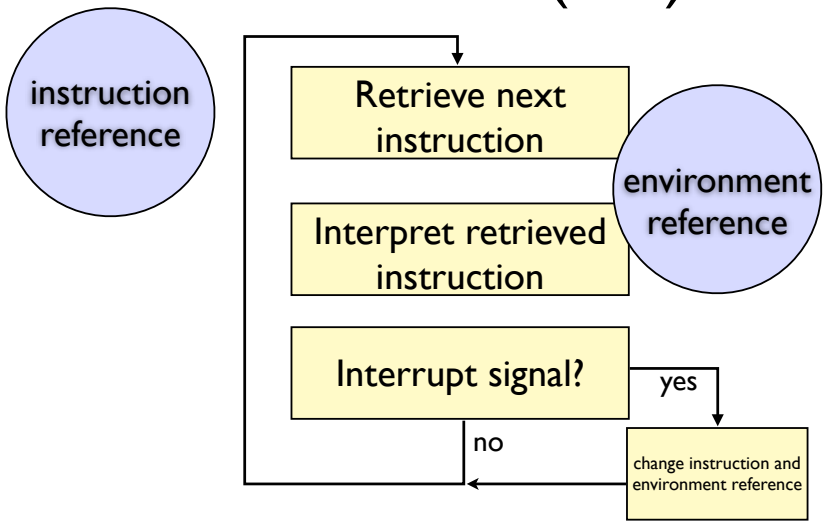
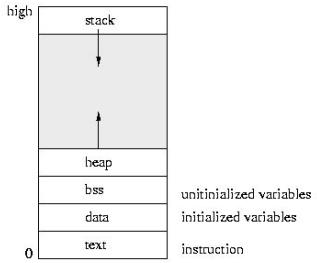


Diagram adapted from Saltzer & Kaashoek

At the hardware level

- The *program counter* is the instruction reference
  - A *program* is the instructions
  - contains address of memory location that stores next instruction
- Memory: Storage, the *stack*, the *page* table (virtual memory bindings), the registers



At the OS level

- A *process* is an instance of a program (code) in execution. In other words, it's ... the same stuff on the previous page, but applied to a single instance of a particular chunk of code running.
  - (You could have multiple processes running from the same program)
- Plus various operating system abstractions: open files, open sockets, etc.

Resource accounting and isolation again!

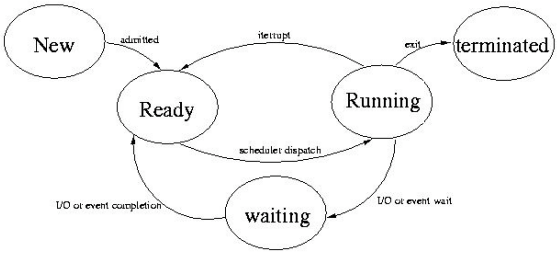
Practical Stuff: Using processes

- Creating a new *process* (has its own memory): fork()
  - Makes an almost exact copy of calling process (PID changes, etc.)
  - How to tell difference? Return value is 0 in child, child PID in parent. How?
  - Stack copied, but different value placed on top of each
- Executing a different *program*: exec()
  - Basically entirely replaces the process with a new one running the new program. But some things, maybe some file descriptors, *are* preserved.

Cool internals: copy-on-write

- CoW is a useful, general technique that shows up all over in systems.
  - Mark parents' memory read-only
  - Have child *share* parents memory instead of copying
  - If either one writes -- hey, it was read only! (CPU will raise an exception)
    - *Now* give the child its own copy of the page of memory someone was writing

Tasks & Scheduling



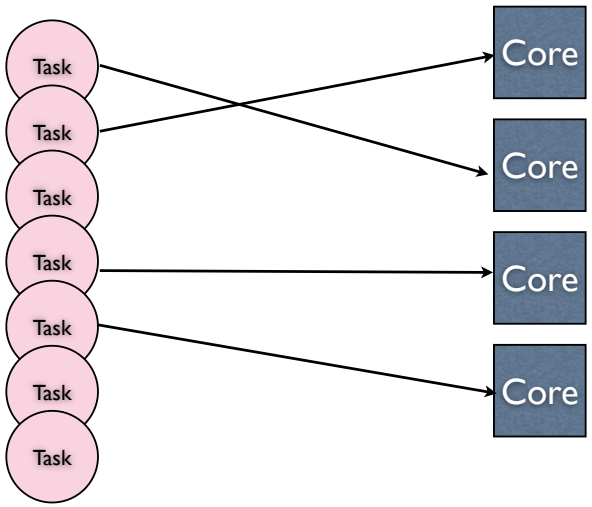
Remember last time that a process "blocked" if it tried to send too much data to a TCP socket?
"waiting"

Resource sharing again!

Types of scheduling

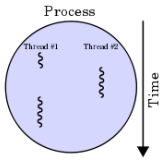
- Cooperative: one task explicitly *yields* to another
- Preemptive: some underlying management thingy (e.g., the OS) can forcibly switch which task is running. Prevents hogging, out of control tasks, etc.
- Food for thought: Why would you ever want cooperative, then? (We'll come back to this)

OS Scheduling



Threads

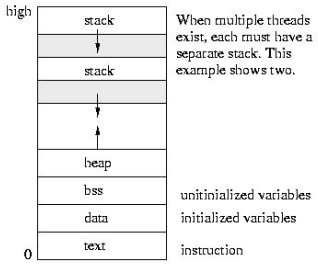
- A *thread* is, roughly, a task within a process
- There can be multiple threads within a single process
- A *process* is the unit of resource allocation (Threads don't have memory)



- Threads do have execution state, their own stack, etc.

figure from wikipedia

Threads in memory



Threads

- Threads share memory
- Handy! They can .. share stuff.
- Dangerous! They can .. muck stuff.

Why Threads?

- Switching between threads faster than switching between processes (don't have to change as much stuff around -- such as invalidating the page table cache)
- Creating and destroying is much cheaper than fork
- Provides convenient abstraction for chunking up work
  - Example: Assign a thread to handling an incoming request in a Web server
  - This use matches well to blocking semantics of posix
    - If we can't write to the socket, thread blocks, some other thread keeps running. That's cool - our thread doesn't *need* to do anything if it can't send to the client...
  - Though it isn't always the best way to do things, in practice

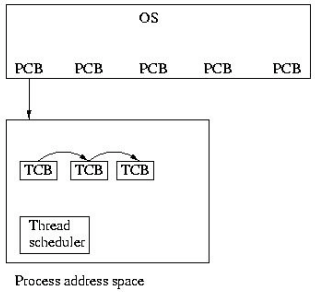
<break>

How threads?

- Well, that depends -- what do you want to accomplish?
- Early days: uniprocessor systems
  - Threads as a programming abstraction (as in previous slide)
  - This was source of many rollicking debates in system community about what abstraction was better. It got ridiculous and religious.
  - But no need for multicore foo like today, so...

User Threads

- Implemented via *thread libraries*.
- These give illusion of independent threads by masking actions that would block, and calling into the thread scheduler instead.
- Typically cooperatively scheduled, but tricky (done automatically on system calls, for ex.)



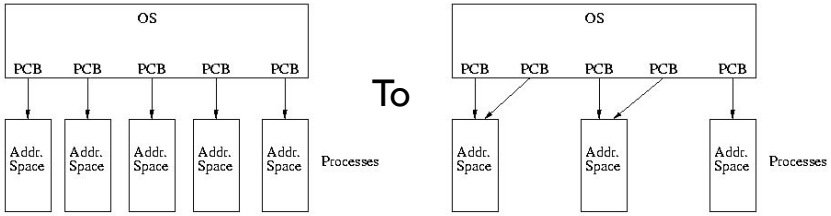
Advantages of User Threads

- Ridiculously fast thread switching.
  - Never even need to enter the kernel to switch threads.
- Provides a good abstraction.
- But...

- On an MP system, only one thread within a process can execute at the same time.
  - Even if the kernel can schedule multiple processes to run concurrently
- If any thread in a process makes a blocking system call, *all* threads will be blocked
  - Common example: Some DNS functions were not re-entrant, and some thread libs failed to mask them. A long DNS delay could hang process. Oops.
  - Not all system calls can be "checked" for blocking using select. Opening a file, e.g.
    - Why can this block???
    - Think about NFS...

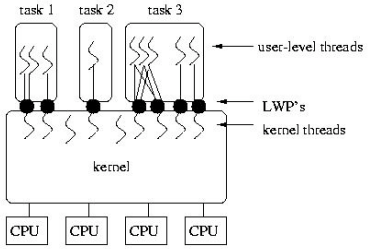
Kernel supported user threads

- Sun called these "Lightweight Processes"



And hybrids...

- If you can tolerate some complexity, you can hybridize these.
  - LWPs are fairly heavyweight (thread switches require going to kernel, etc.)
- Communication between LWPs requires kernel
- LWPs consume more resources than user threads
- More expensive to create and destroy



That said

- For simplicity, and as CPU gets cheaper and cheaper and the importance of maximally exploiting parallelism grows...
- more and more we're seeing just kernel supported user threads
- But at other levels...
  - Many interpreted languages (ruby, python) provide user threads
  - Keeps things simpler (don't have to write a parallel interpreter)
  - but you have to go multiple-process to use multicore.
  - (Sun wrote java and they've always liked SMP. Java uses native OS threads, so on sun, can use LWPs or user threads...)

Tasks in dist. systems

- May hear other terms:
  - "Workers" (*clients* that are assigned *jobs* by a *scheduler* of some sort)
  - "Master" (the node/task that hands out work)
- Mid-90s research looked at "remote fork" and similar primitives -- spawn a new task on some other computer. We'll look more at these abstractions later.