

Administrivia

- Remember to check the "Lecture topics and assignments" page for a link to Homework 1.

Slide 1

Process Abstraction, Review

- We want o/s to manage "things happening at the same time" — applications, hidden tasks such as managing a device, etc.
- Key abstraction for this — "process" — program plus associated data, including program counter.
- True concurrency ("at the same time") requires more than one CPU. We can get apparent concurrency via interleaving (as discussed last time) — model one virtual CPU per process and have the real CPU switch back and forth among them.
- Switching from one process to another — "context switch": Goal is to suspend work on a process such that we can later pick up where we left off — so we have to save CPU state (program counter, registers, etc.) for the current process and replace it with previously-saved state for the next process.

Slide 3

A Bit More about Virtual Machine Operating Systems

- An update to last time's discussion of VM/370: Its successors (VM/ESA, z/VM) are still in use, and now sometimes used to run hundreds of Linux servers on an IBM mainframe system (!).
- An interesting open-source virtual machine project is Mac-on-Linux.

Slide 2

Process Abstraction, Continued

- Can also associate with process an "address space" — range of addresses the program can use. Simplifying a little, this is "virtual memory" (like the virtual CPU) that only this process can use.

Slide 4

Process Creation and Termination

- When are processes created?
 - At system startup.
 - When another process makes a "create process" system call — e.g., to start a new application.
- When are processes destroyed?
 - At program exit.
 - After some kinds of errors.
 - When another process makes a "kill process" system call.

Slide 5

Process States, Continued

- Notice that moving to and from "blocked" state doesn't involve decision-making, but ready/running transitions do.
- The decision-maker — "scheduler" (to be discussed later). Often "running to ready" is triggered by an interrupt (I/O, timer, etc.), and "ready to running" involves this scheduler.

Slide 7

Process States

- Can think of processes as being in one of three states:
 - "Running" — being executed by the CPU.
 - "Blocked" — waiting for something to happen (I/O to complete, another process to do something, etc.) and can't do anything useful until it does.
 - "Ready" — not blocked, but waiting because another process is currently running on the CPU.
- Possible transitions:
 - Running to blocked — happens when, e.g., a process makes an I/O request and can't continue until it's complete.
 - Blocked to ready — happens when the event the blocked process is waiting for occurs.
 - Running to ready, ready to running — needed if we want some sort of time-sharing (give all non-blocked processes "a turn" frequently).

Slide 6

Implementing Processes

- Think about how you would implement this abstraction ...
- First, you'd want a data structure to represent each process, to include:
 - Process ID.
 - Process state (running / ready / blocked).
 - Information needed for context switch — a place to save program counter, registers, etc.
 - Other stuff as needed — a list of open files, e.g.
- Then you'd collect these into a table or something — "process control table", and individual data structures are "entries in the process control table" or "process control blocks".

Slide 8

Interrupt Handling, Revisited, Part 1

- When an interrupt occurs, the hardware:
 - Saves a little about the current process (program counter at least) in an agreed-upon location, e.g., on stack.
 - Transfers control to fixed location — could be always the same location, or one of several depending on the type of interrupt.

Slide 9

Processes Versus Threads

- So far I've used "process" in an abstract/general way.
- In typical implementations, though, "process" is more specific — something that has its own address space, list of open files, etc. (I'll call these "heavyweight processes".)
 - Advantages — such processes don't interfere with each other.
 - Disadvantages — they can't share data, switching between them is expensive ("a lot of state" to save/restore).
- For some applications, might be nice to have something that implements the abstract process idea but allows sharing data and faster context switching — "threads".

Slide 11

Interrupt Handling, Revisited, Part 2

- Code at fixed location is an "interrupt handler", which:
 - Saves enough of the CPU's current state to enable later restart, usually in current process's process control block.
 - "Handles" the interrupt, but minimally — saves data that could be lost (e.g., in device's input buffer), marks blocked processes ready if appropriate.
 - Invokes scheduler to decide which process to run next.
 - Restores saved CPU state for this next process (from its process control block), causing it to resume execution.
- Other interrupts may be "disabled" during this processing.

Slide 10

Threads

- So, threads are another way to implement the process abstraction.
- Typically, a thread is "owned" by a (heavyweight) process, and all threads owned by a process share some of its state — address space, list of open files.
- However, each thread has a "virtual CPU" (a distinct copy of registers, including program counter).
- Advantages? threads can share data (same address space), switching from thread to thread is fairly fast.
- Disadvantages? sharing data has its hazards (more about this later).
- Implementation involves data structures similar to process table.

Slide 12

Implementing Threads

- Two basic approaches — “in user space” and “in kernel space” (next two slides).
- Various hybrid schemes also possible.

Slide 13

Implementing Threads “In Kernel Space”

- Basic idea — operating system is involved in managing threads, the work of managing multiple threads happens via system calls (rather than user-level library calls).
- Advantages? avoids the difficulties of implementing in user space (previous slide).
- Disadvantages? probably less efficient.

Slide 15

Implementing Threads “In User Space”

- Basic idea — operating system thinks it's managing single-threaded processes, all the work of managing multiple threads happens via library calls within each process.
- Advantages? fewer system calls, hence probably more efficient.
- Disadvantages?
 - If a thread blocks, it may do so in a way that blocks the whole process.
 - Preemptive multitasking is difficult/impossible.
 - Using multiple CPUs is difficult/impossible.

Slide 14

Example Implementations

- Unix systems vary as to which they use (see chapter 10 for more info). Until recently Linux did kernel-space threading, but allegedly with some tweaks to make it more efficient. There have been some changes in the latest version ...
- Windows NT/2000 apparently is such that *all* processes have at least one thread, and the basic scheme is either kernel-space or a hybrid (see chapter 11 for more info).

Slide 16

Minute Essay

- In a system with 8 CPUs and 100 processes, what's the maximum number of processes that can be running? ready? blocked?
- Is there anything today that was totally unclear?