UNIVERSITY OF CALIFORNIA AT BERKELEY
College of Engineering
Department of Electrical Engineering and Computer Sciences
Computer Science Division

CS 162 John Ousterhout
Fall 1988

Topic 1: Introduction to Processes; Dispatching

- Readings for this topic:
  - Silberschatz/Peterson: Sections 2.3, 4.2.

- With many things happening at once in a system, need some way of separating them all out cleanly. That's a process.

- Important concept: decomposition. Given hard problem, chop it up into several simpler problems that can be solved separately.

- What is a process?
  - ''An execution stream in the context of a particular process state.''
  - *A more intuitive, but less precise, definition is just a running piece of code along with all the things that the code can affect or be affected by.*
  - Process state is everything that can affect, or be affected by, the process: includes code, particular data values, open files, etc.
  - Execution stream is a sequence of instructions performed in a process state.
  - Only one thing happens at a time within a process.

- Is a process the same as a program? *No, it's both more and less.*

(what is a program? the statements that a user writes, or a command he/she invokes)

- *More - a program is only part of the state; several processes may be derived from the same program. If I type ‘‘ls’’, something different happens than if you type it.*
- *Less - one program may use several processes, e.g. cc runs other things behind your back.*

- Some systems allow only one process (mostly personal computers). They are called *uniprogramming* systems (not uniprocessing; that means only one processor). Easier to write some parts of OS, but many other things are hard to do. *E.g. compile a program in background while you edit another file; answer your phone and take messages while you’re busy hacking. Very difficult to do anything network-related under uniprogramming.*

- Most systems allow more than one process. They are called *multiprogramming* systems.

- First, have to keep track of all the processes. For each process, *process control block* holds:
  - Execution state (saved registers, etc.)
  - Scheduling information
  - Accounting and other miscellaneous information.

  Process table: collection of all process control blocks for all processes. In Unix the process table is a fixed-size array.

- How can several processes share one CPU? OS must make sure that processes don’t interfere with each other. This means
  - Making sure each gets a chance to run (fair scheduling).

    - Making sure they don't modify each other's state (protection).

- *Dispatcher*: inner-most portion of the OS that runs processes:
    - Run process for a while
    - Save state
    - Load state of another process
    - Run it ...

- How does dispatcher decide which process to run next?
    - Plan 0: search process table from front, run first runnable process. *What does runnable mean?*
        - Might spend a lot of time searching.
        - Weird priorities.
    - Plan 1: link together the runnable processes into a queue. Dispatcher grabs first process from the queue. When processes become runnable, insert at back of queue.
    - Plan 2: give each process a priority, organize the queue according to priority. Or, perhaps have multiple queues, one for each priority class.
    - *Who decides priorities? Could do in dispatcher, but then the implementation and policies get confused. Priorities are decided by a separate scheduler. Will be discussed in a month.*

- CPU can only be doing one thing at a time: if user process is executing, dispatcher isn't: OS has lost control. How does OS regain control of processor? *One approach: sleeping beauty (hope process is Prince Charming and will wake you up). Another approach: alarm clock (make sure you get woken up). What are alarm clocks? Interrupts.*

- Internal events (things occurring within user process):
  - System call.
  - Error (illegal instruction, addressing violation, etc.).
  - Page fault.

  These are also called *traps*. They all cause a state switch into the OS.

- External events (things occurring outside the control of the user process):
  - Character typed at terminal.
  - Completion of disk operation (controller is ready for more work).
  - Timer: to make sure OS eventually gets control.

  External events are usually called *interrupts*. They all cause a state switch into the OS. *This means that user processes cannot take I/O interrupts*.

- *Lecture 3 ended here.*

- When process isn't running, its state must be saved in its process control block. What gets saved? Everything that next process could damage:
  - Program counter.
  - Processor status word (condition codes, etc.).
  - General purpose registers.
  - Floating-point registers.
  - All of memory? *Three possibilities: 1) trust next process; 2) move everything to disk (Alto); 3) memory protection (will be discussed in 6 weeks).*

- Saving state of user process: it's tricky because the the OS needs some state to execute the state saving and restoring code. All

machines provide some special hardware support for saving and restoring state:

- Motorola 68000: hardware doesn't know much about processes, it just moves PC and status word to/from the stack. OS then transfers to/from PCB, and handles rest of state itself (must be done carefully, using assembler code, to avoid overwriting state while saving it).
- Intel 432: hardware does all state saving and restoring into process control block, and even dispatching.

- Creating a process from scratch:
  - Load code and data into memory.
  - Create (empty) call stack.
  - Create and initialize process control block.
  - Make process known to dispatcher.

- Forking (like in TOY or UNIX): want to make a copy of existing process. *(warning: this may not be quite the same as fork from the book or fork from a programming language fork-join pair)* For example, the shell makes a copy of itself for each command. One waits around, the other goes off and executes the command.
  - Make sure process to be copied isn't running and has all state saved.
  - Make a copy of code, data, stack.
  - Copy PCB of source into new process.
  - Make process known to dispatcher.

- What's missing?

- *The two processes are exactly the same, so this isn't very interesting.*

Don’t usually want to do the same thing twice. Joke about a duplicated professor in class.

- *Explain difference between fork and exec, go through fork example.*
- *Lecture 4 ended here.*