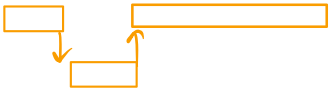
Topic 3: Introduction to Processes

Reading: Sections 2.1-2.4
Next reading: Section 4.1-4.6, 5.0

- With many things happening at once in a system, need some way of separating them all out cleanly. That's what *processes* are for.
  - What is a process? Running Program



    - "An execution stream in the context of a particular process state."
    - Process state includes code, stack, variables, open files, etc.
  - A more intuitive, but less precise, definition is just an executing program, along with all the things that the program can affect or be affected by.
  - Only one instruction at a time is executed within a process* (* indicates a white lie -- e.g. threads, pipelines).
- Is a process the same as an application?
  - No, it's both more and less.
  - More - an application is only part of the state; several processes may be created from the same executable.
  - Less - one application may use several processes, e.g. gcc uses multiple processes to do its job. use application can produce multiple processes

- *Independent process*: one that can't affect or be affected by the rest of the universe. only depend on input, pause / resume won't affect result
  - Its state isn't shared in any way with any other process.
  - *Deterministic*: input state alone determines results.
    - Reproducible.
    - Can stop and restart without effect.
- There are many different ways in which a collection of independent processes might be executed on a processor:
  - *Uniprogramming*: a single process is run to completion before the next process is started. Easier to write some parts of OS, but makes OS hard to use. 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.
  - *Multiprogramming (multitasking)*: share one processor among several processes. Easier to use than uniprogramming, but complicates OS. How to go back&forth seamlessly
  - *Multiprocessing*: if multiprogramming works, then it should also be ok to run processes in parallel on separate processors/cores.
    - A process runs on only one processor/core at a time.

        - A process may run on different processors/cores at different times (move state, assume processors/cores are identical).
        - Cannot distinguish multiprocessing from fine-grain multiprogramming.

- How often are processes independent? Not very, but that's another topic.
- If the OS supports multiprogramming then it needs to keep track of all processes. For each process, its *process control block* (PCB) holds:
    - Execution state (saved registers, etc.). Often called its *context*.
    - Scheduling information.
    - Accounting and other miscellaneous information.
- Each process has a unique name called its *process ID* (PID).
- Process table: an array of all PCBs. In Unix the process table traditionally is a fixed-size array, and a process's PID is derived from the index of its PCB in the process table.
- 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 (scheduling).

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

- *Dispatcher*: innermost portion of the OS that runs processes:
    - Run process for a while
    - Save state
    - Load state of another process
    - Run process for a while…
- How does dispatcher decide which process to run next?
    - Plan 0: search process table from beginning, 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 the queue. *Round-robin*
    - 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 mechanism and policy get confused. Priorities are decided by a separate *scheduler*. Will be discussed later.