Topic 4: Cooperating Processes

- Reading: 4.1-4.6, 5.0
  Next reading: 5.1

- Independent processes are a nice abstraction, but it is often useful to have processes cooperate to achieve a common goal. E.g. `grep foo /usr/dict/words | wc.`
  start simulteneously, one feeds another, sys pace both.

- Cooperating processes are those that share state. May or may not actually be "cooperating".

  - Could be memory (assumes processes can share memory, more details on how to do this later).

  - Could be pipes -- one process writes, another process reads.

- Behavior is *nondeterministic*: depends on relative execution sequence and cannot be predicted a priori.

- Behavior may be *irreproducible*.

- Example: one process writes "ABC" one character at a time to the terminal, another writes "CBA". Are these cooperating processes? How? Can get different outputs, cannot tell what comes from which. E.g. which process output first "C" in "ABCCBA"? Note the subtle sharing that occurs here via the terminal. Not just anything can happen, though. For example, "AABBCC" can't occur.

- When discussing concurrent processes, multiprogramming is as dangerous as multiprocessing unless you have tight control over the multiprogramming. Also, smart I/O devices are as bad as cooperating processes (they share the memory).

- Why permit processes to cooperate?
  - Want to share resources:
  - One computer, many users.
  - One file of checking account records, many tellers. What would happen if there were a separate account for each teller? Could withdraw same money many times.
  - Want to do things faster:
  - Read next block while processing the current one.
  - Divide job into sub-jobs, execute in parallel.
  - Want to construct systems in modular fashion. (e.g. `ls | grep foo | wc`)
- Basic assumption for systems of cooperating process is that the order of some operations is irrelevant; some operations are independent of other operations. Only a few things matter:
  - Example: A = 1; B = 2; has the same result as B = 2; A = 1;
  - Another example: A = B+1; B = 2*B can't be re-ordered.
  - Another example: suppose A = 1 and A = 2 are executed in parallel: *race condition*. Don't know what will happen; depends on which one goes fastest. What if they happen at EXACTLY the same time?   Can't tell anything without more information. Could end up with A=3!
  - If the exact order of everything matters, then there's no point in having multiple processes: just put everything in one process.
- *Atomic operations*: Before we can say ANYTHING about cooperating processes, we must know that some operation is

atomic, i.e. that it either happens in its entirety or not at all. An atomic operation cannot be observed in the middle. E.g. suppose that printf is atomic -- what happens in the printf("ABC"); printf("CBA") example?

- References and assignments are atomic in almost all systems. A=B will always get a good value for B, will always set a good value for A (but not necessarily true for arrays, structures, or even floating- point numbers -- anything that requires multiple memory accesses).
- In uniprocessor systems, anything between interrupts is atomic, provided the process doesn't invoke the dispatcher.
- It follows that anything with interrupts off is atomic (with the same caveat).
- If you don't have an atomic operation, you can't make one. Fortunately, the hardware folks give us atomic operations.
- In fact, if there is true concurrency, it's very hard to make a perfect atomic operation; most of the time we settle for things that only work "most" of the time.
- If you have any atomic operation, you can use it to generate higher- level constructs and make parallel programs work correctly. This is the approach we'll take in this class.