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

CS 162
Fall 1988

John Ousterhout

Topic 2: Independent and Cooperating Processes

- Readings for this topic:
  - Silberschatz/Peterson: Section 3.1.

- Independent process: one that can't affect or be affected by the rest of the universe.
  - Its state isn't shared in any way by any other process.
  - Deterministic: input state alone determines results.
  - Reproducible.
  - Can stop and restart with no bad effects (only time varies). Example: program that sums the integers from 1 to 100.

- 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 anything else can be run on the processor.
  - Multiprogramming: share one processor among several processes. If no shared state, then order of dispatching is irrelevant.
  - Multiprocessing: if multiprogramming works, then it should also be ok to run processes in parallel on separate processors.
    - A given process runs on only one processor at a time.
    - A process may run on different processors at different times

(move state, assume processors are identical).
    - Cannot distinguish multiprocessing from multiprogramming on a very fine grain.

- How often are processes independent?

- Cooperating processes:
    - Machine must model the social structures of the people that use it. People cooperate, so machine must support that cooperation. Cooperation means shared state, e.g. a single file system.
    - Cooperating processes are those that share state. *May or may not actually be ‘‘cooperating’’.*
    - Behavior is *nondeterministic*: depends on relative execution sequence and cannot be predicted a priori.
    - Behavior may be *irreproducible*.
    - Example: one process writes ‘‘ABC’’ 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 current one.
    - Divide job into sub-jobs, execute in parallel.
  - Want to construct systems in modular fashion. (e.g. tbl | eqn | troff)

- *Lecture 5 ended here.*

- Basic assumption for cooperating process systems is that the order of some operations is irrelevant; certain operations are independent of certain other operations. Only a few things matter:
  - Example: A = 1; B = 2; has 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 mattered, then there's no point in having multiple processes: just put everything in one process.

- *Atomic operations*: Before we can say ANYTHING about parallel processes, we must know that some operation is *atomic*, i.e. that it either happens in its entirety without interruption, or not at all. Cannot be interrupted in the middle. E.g. suppose that printf is atomic -- what happens in 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, records, or even floating-point numbers).

- In uniprocessor systems, anything between interrupts is atomic.
- If you don't have an atomic operation, you can't make one. Fortunately, the hardware guys give us atomic ops.
- *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.