

CS 414 : Operating Systems

UNIVERSITY OF VIRGINIA
Department of Computer Science

Spring 2008

Topic 2: Processes

- Context switching: saving the state (to where?) and restoring it

What gets saved? Everything that next process could damage:

- program counter, processor status word, registers.
 - all of memory? all disks? all the stuff on tape?
 - options for memory contents: a) trust next process b) move everything to disk (Alto system), c) memory protection
-
- Why tricky?
-
- All machines provide some special hardware support.
 - Motorola 68000: hardware just moves PC and status word to the stack. OS then handles rest of state itself. Must be done carefully. Why?
 - Intel 432: hardware does all state saving and restoring into process control block, and even dispatching. Hardware dispatcher. Desirable?
-
- Ugly issue of performance: sometimes making dirty shortcuts.
-
- Birth of a process: A) Creating it from scratch:
 - Allocate memory
 - Load code and data into memory.
 - Create (empty) call stack.
 - Create and initialize process control block.
 - Make process known to dispatcher.

- B) Forking: copying existing process
 - Make sure process to be copied isn't running and has all state saved.
 - Make a copy of code, data, stack.
 - Copy PCB into new PCB.
 - Make process known to dispatcher.

What's missing?

- Unix FORK: when user typed "ls"

```
pid = fork();
if (pid < 0)
    printf(stderr, "Fork Failed");
    exit(-1);
else if (pid == 0)
    execlp("/bin/ls", "ls", NULL);
else
    wait(NULL);
    printf("Child Complete");
    exit(0);
```

- Independent process: neither affect nor affected by the rest of the universe.
 - its state is not shared in any way with other processes
 - deterministic
 - reproducible
 - scheduling does not affect the result
- There are many different ways in which a collection of independent processes might be executed on a processor.
- How often are processes independent?
- Cooperating processes:
 - cooperating processes share state. *May or may not actually be “cooperating”.*
 - *nondeterministic*: depends on relative execution sequence.
 - *irreproducible*.
 - example: one process writes “ABC” to the monitor, another writes “CBA”.
- Why allow processes to cooperate?
 - want to share resources:
 - one computer, many users.
 - one file of checking account records, many tellers.
 - want to do things faster:
 - read next block while processing current one.
 - divide job into sub-jobs, execute in parallel.
- 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*.

- *Atomic operations:*
 - If printf is atomic -- what happens in `printf("ABC"); printf("CBA")` example?
 - References and assignments are atomic in almost all systems.
 - 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.
 - If you have any atomic operation, you can use it to generate higher-level constructs and make parallel programs work correctly.