2 Introduction to Processes

Reading: Anderson/Dahlin Chapter 4; Silberschatz/Galvin/Gagne: Chapter 3–4

With many things happening at once in a system, need some clean way of separating them all out cleanly. ⇒ sequential process, or just *process*.

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

What is a process?

- An executing program along with all the things that the program can affect or be affected by.
- The dynamic execution context ("active spirit") of a program (as opposed to the program, which is static.)
- Only one thing happens at a time within a process
- The unit of execution and scheduling

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 ; executing a program (e.g., ls) multiple times results in separate processes, one for each execution of the 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.

Early personal computer operating systems allowed only one process. They were 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.

What's in a process? A process contains all the state of a program in execution:

- the code for the running programs
- the data for the running program
- the execution stack showing the state of all calls in progress
- the program counter, indicating the next instruction
- the set of CPU registers with current values
- the set of OS resources held by the program (references to open files, network connections)

Process State
Each process has an execution state that indicates what it is currently doing

- ready—waiting for the CPU
- running—executing instructions on the CPU
- waiting—waiting for an event, e.g., I/O completion

Only one process can be *running* on any processor at any instance. Many processes can be *waiting* and *ready.*

OS has to keep track of all the processes. Each process is represented in the OS by a data structure called *process control block* (PCB):

queue pointers
process state
process number
program counter (PC)
stack pointer (SP)
general purpose register contents
floating point register contents
memory state
I/O state
scheduling information
accounting information
⋮

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 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 later.

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? Naive approach: hope process is well-behaved and will eventually return control to OS. Safe 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 also cause a state switch into the OS. This means that user processes cannot take I/O interrupts.

How do interrupts differ from function calls?

1. the target address is determined by a table controlled by the OS kernel (int type $->$ handler address)

2. an interrupt switches the CPU to *kernel mode*; returning from the interrupt handler switches the CPU back to *user mode*.

When the CPU is in user mode, certain *priviledged instructions* are not available and cause a trap if executed. E.g., turning off interrupts is a priviledged instruction. As a result, a user process cannot escape preemption by timer interrupts, and the OS kernel re-gains control periodically.

PCBs and hardware state

- When a process is running, its PC, SP, register contents are loaded in the CPU, and the PCB is not up-to-date.
- When the OS preempts a running process, it stores the values of these registers into the PCB for that process.
- When the OS dispatches a running process, it loads the CPU registers with the values stored in that process'es PCB.
- This process of switching the CPU from one process to another is called a *context switch*.
- Context switch code tricky, since it needs some state of it own to do the state saving/restoring. Written in assembly code.

State Queues

- The OS maintains a set of queues that maintain the state of all processes in the system.
- There's typically one queue per process state.
- Each PCB is queued on a state queue corresponding to its current state.

PCBs and State Queues

- PCBs are dynamically allocated in OS memory.
- When a process is created, a PCB is allocated, initialized (how?), and placed on the appropriate queue.
- As the process changes state, its PCB moves from queue to queue.
- When the process terminates, its PCB is deallocated.

3 Independent and Cooperating Processes

Independent processes cannot affect or be affected by the rest of the universe:

- state isn't shared in any way by another process
- deterministic: input state alone determines results
- reproducible
- can stop and start 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)

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.