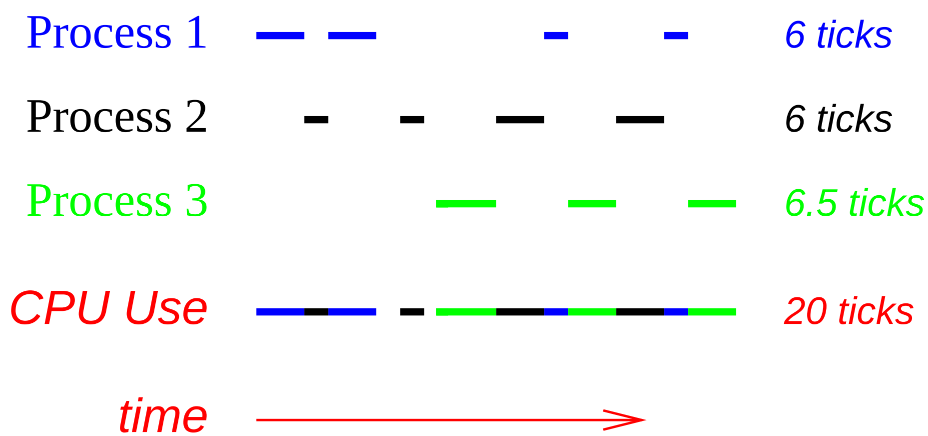
Multiprogramming

- Computers don't really run multiple programs simultaneously; it just appears that way
- Each *process* runs to completion, but intermixed with other processes



- The exact timing pattern varies for each process
- Note the idle times

Process Time

- What matters is that each process *eventually* finishes
- You start reading a book, put it down for a while, pick it back up and resume where you left off
- As long as you finish soon enough — whatever that means — the exact time doesn't matter

Real-Time Scheduling

- In some environments, processes need to run at the right time
- Think of process control computers — valves must open and close promptly
- *Not* suitable for this paradigm

What’s a Process?

- I keep talking about “processes”. What are they?
- No rigorous definition!
- Precise characteristics differ on different systems
- Common themes: separately scheduled; some measure of isolation

Separately Scheduled

- On all systems, processes can compete for the CPU
- One process *blocking* or being pre-empted lets another process run
- On some systems, a process can be composed of several *threads* that themselves can compete for the CPU
- On multiprocessor (multi-CPU) machines, different processes can execute truly simultaneously on different CPUs

Isolation

- Termination protection
- Address space
- Security context
- Other system-related state

Termination

- Processes are generally isolated from failures of other processes
- Termination of a process — normal or abnormal — does affect other processes
- Often a reason for process creation: let failures happen in an isolated setting, with minimal cleanup needed

Address Space

- Processes often have separate address spaces from each other
- Changes to memory in one process do not affect other processes
- May or may not use virtual memory to provide overlapped address space — on early PDP-11 Unix systems, all processes started at location 0

Exceptions. . .

- On OS/360, the *job* was the unit of memory protection; all “tasks” (the OS word for “process”) shared memory
- On some versions of MVS, all jobs have parts of kernel memory available at the same addresses
- On Unix systems, program instructions — but not data — are shared among different processes; this often includes shared libraries
- Unix processes can arrange to share certain memory areas
- Files can be mapped to memory areas; on different processes, the same data can thus appear at different addresses

Security Context

- Access credentials — UID on Unix — are process-specific
- SetUID applies to a process
- On some systems, process can share credentials

System State

- Open files
- Current working directory
- Trap-handling state
- Permissions for newly-created files
- Often much more

A Historical Note

- PDP-11s had a 16-bit address space: 65536 bytes
- The page size — the granularity of memory protection — was 4096 bytes
- Even with tiny programs, that meant at most 16 processes if they shared address space
- Separate address space per process was a *necessity*

Processes and System Calls

- Suppose a process issues a system call. What happens in the kernel?
- Interrupt hardware saves old PSW; loads new PSW
- Software interrupt handler saves registers; branches to system call dispatcher
- Dispatcher figures out which system call it is, and calls that subroutine
- That subroutine may call others
- We need a stack

The Kernel Stack

- As discussed previously, cannot trust user-level setup
- Must have a kernel stack
- Stack size is limited — watch for too-deep recursion!
- Where is this stack?

Per-Process Stacks

- Suppose this system call blocks waiting for I/O
- Another process can run; what if it issues a system call?
- It can't share the first process' stack, because that one may need to be used while this one is active
- We need a separate kernel stack per process!

Per-Process State

- Actually, we need a lot of state per process
- The basic per-process structure on Linux (**`task_struct`**) is 175 lines of C, and it points to other per-process structures
- What's in them?
- Two broad classes: fields needed when running and fields needed when deciding whether or not to run the process

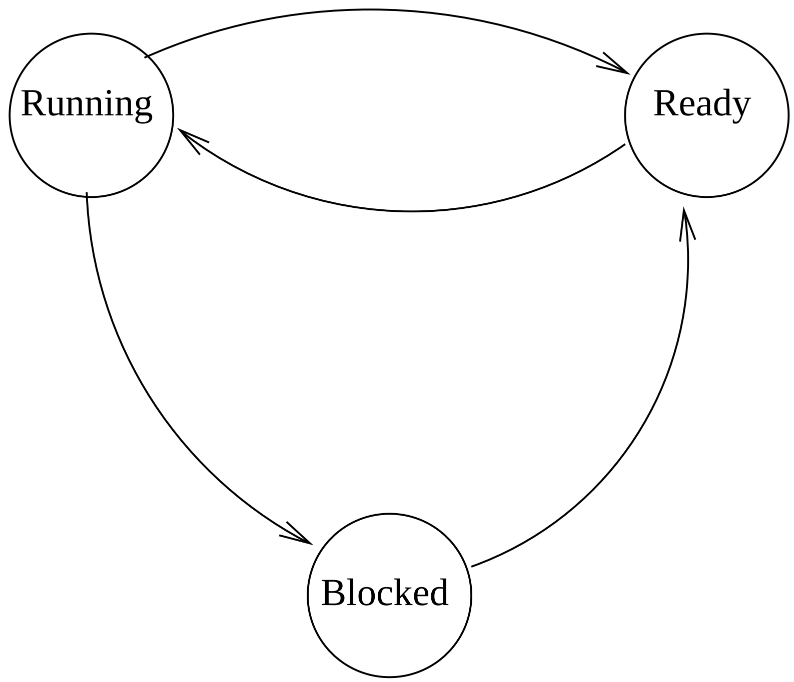
Per-Process State: Always Needed

- Process state: running, ready, blocked
- What it's blocked on
- resource (CPU, RAM, I/O, etc.) usage history
- Priority
- Signal status
- ProcessID, process group
- Pointers to other fields
- Memory allocations

Per-Process State: Needed When Running

- Kernel stack
- PSW, program counter
- Open file descriptors
- Some state information, such as current directory
- user credentials

Scheduling State Transition



Note that when making the transition from Blocked to Ready, it may be necessary to copy in some state data from disk. This, of course, is itself a blocking operation.

Process Creation

- The first process is created at boot time
- Any process, including that one, can create new processes
- Details differ widely between operating systems

Process Creation on Unix

- Basic operation: **`fork()`**
- Creates an exact copy of the process, code, data, and state
- Only difference: parent is passed back processID of child; child is passed back 0
- The child process typically manipulates some file descriptors, then **`exec()`**s some other program
- Note: virtually all Unix commands create separate processes. (That had been the intention for Multics, but it was too expensive there.)

Optimization?

- Copying all of that data is expensive
- Instead, use the same pages, marked read-only, and let the virtual memory system copy as needed
- Manipulating all of those page table entries is remarkably expensive, too — it saves less than you might hope

Inheritance

- Unix processes inherit copies of file descriptors
- If you’re not careful, output can be intermixed; input consumed by one is not available to the other
- Since an X11 window is an open file descriptor, windows are inherited as well
- All of this is very powerful, but easy to get wrong

Process Creation on Windows

- The **`CreateProcess`** call creates processes on Windows
- Executing a new program is part of the process creation mechanism
- 10 parameters control the program to be executed, window creation, priority, security attributes, file inheritance, and much more
- The Windows call does more for you, but is it simpler?

Process Relationships in Unix

- A newly-created process is a *child* of the parent process
- When a child process terminates, its resource consumption is passed up to the parent
- The parent process is notified when a child terminates, and needs to "reap" it (via the **`wait()`** system call)
- If not, the process remains a *zombie*
- Processes whose parent dies become children of process 1

Process Groups

- Related processes — say, all the elements of a pipeline — form a *process group*
- Certain *signals* — interrupts to a process — are sent to all members of a process group
- Thus, if you hit `^C`, all of the processes are killed

Windows Process Relationships

- All Windows processes are siblings; there are no other relationships
- When a process creates another process, it receives a *process handle* that can be used to control that process
- The process handle can be passed around to other processes

Process Termination

- When a process terminates, its resources must be freed
- Some of these resources including open files; closing a file can block
- Termination isn't easy, and may not terminate quickly...

Creating a Process — Overview

- Parent issues a system call
- Interrupt handler invokes the kernel
- Kernel creates the process
- At some point, it runs

Issuing a System Call

Parent Issues (machine-language) system call instruction

Hardware Old PSW and program counter are saved

Hardware New PSW and program counter are loaded

Assembler Registers are saved in current process' kernel data structure

Assember System call dispatcher is invoked

C Process creation routine invoked

Process Creation Routine

- Verify that resources are available
  - Process table entry
  - User's process quota
- Create new process table entry
- Copy inherited data to new process
- Make sure "saved" registers are correct, including return value to indicate it's a child process
- Return to system call dispatcher

Returning From the Kernel

- When the new process is created, the dispatcher invokes the *scheduler*
- The scheduler decides which process will run next — the parent, the child, or some other process entirely
- Assembler code restores registers *for whatever process is the next to run*
- The old PSW and program counter are reloaded by “return from interrupt” instruction

Note Well. . .

- How the new proces behaves is *completely* determined by what is put into the process structure
- "Registers" aren't a C concept, but the contents of the any process' registers are determined by what is put into this structure
- Many subtle details; see "You are not expected to understand this" at `http://cm.bell-labs.com/cm/cs/who/dmr/odd.html`

Summary

- Processes are fundamental to multiprogramming
- The details differ widely among different systems
- Process creation and interrupt-handling are closely linked