## Processes and Process Management

Brian Bershad

## Processes are about virtualizing resources

- There are only a limited number of CPUs.
- You may want to run more than one program at a time.
  - » why?
- We need an abstraction that decouples the physical CPU from the computing resource presented to the program.
- This abstraction will trade *ease of use* for performance and control.



## Process

- **Basic idea is the *process*:**
  - process is the unit of of execution
  - the unit of scheduling and resource allocation
  - the dynamic (active) execution context (as opposed to a program, which is static)
- **A process is sometimes called a *job* or a *task* or a *sequential process*.**
- **A sequential process is a program in execution; it defines the sequential, instruction at a time, execution of a program.**



## Process Management

- **Process management deals with several issues:**
  - what are the units of execution?
  - how are those units of execution represented in the os?
  - how is work scheduled on the CPU?
  - what are possible execution states, and how does the system move from one to another?



## What's in a process?

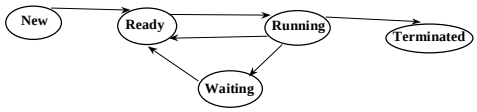
- **A process consists of at least:**
  - the code for the running program
  - the data for the running program
  - an execution stack showing the state of calls made
  - the Program Counter, indicating the next instruction
  - a set of general purpose registers with current values
  - a set of operating system resources (memory, open files, connections to other programs, etc.)
- **The process contains all the state for a program in execution.**



## Process State

- **There may be several processes running the same program (e.g., an editor), but each is a distinct process with its own representation.**
- **Each process has an *execution state* that indicates what it's currently doing, e.g.:**
  - *ready* : waiting for the CPU
  - *running* : executing instructions on the CPU
  - *waiting* : waiting for an event, e.g, I/O completion
- **As a program executes, it moves from state to state.**

## Process State Changing




**Processes move from state to state as a result of actions they perform (e.g., system calls), operating system actions (rescheduling), and external actions (interrupts).**



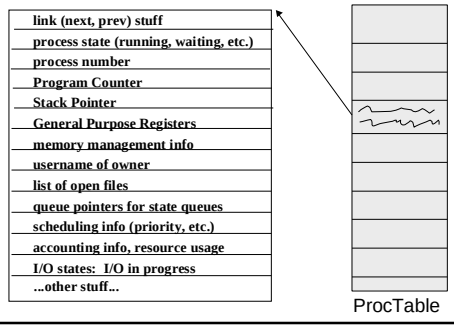
## Process Data Structures

- **At any time, there are many processes in the system, each in it's particular state.**
- **The OS must have data structures representing each process: this data structure is called the PCB:**
  - *Process Control Block*.
- **The PCB contains all of the info about a process.**
- **The PCB is where the OS keeps all of a process' execution state (e.g., PC, registers) when it is not running.**



## PCB

**PCB contains lots of information, e.g.:**






## PCBs and Hardware State

- **When a process is running, it's Program Counter, stack pointer, registers, etc., are loaded on the CPU (i.e., the hardware registers contain the current values).**
- **When the OS stops a running process, it saves the current values of those registers into the PCB for that process.**
  - » **save state**
- **When the OS is ready to start executing a waiting process, it loads the hardware registers from the values stored in that process' PCB.**
  - » **restore state**
- **The process of switching the CPU from one process to another is called a *context switch*. Timesharing systems may do 100s or 1000s of context switches a second.**



## Context Switch

- Two requirements when we switch contexts.
  - » save away the previous context
  - » restore the next context.
  - » eg, swtch_context(P1, P2) might be an OS call deep deep down.
    - – saves current context in P1's PCB.
    - – restores the processor to the context of P2's.
- How is context switching different from calling a procedure?



## Creating a Process

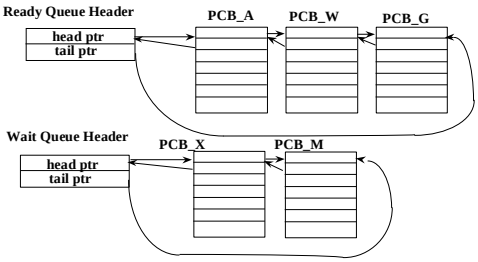
- **One process can create other processes to do work. These are *child* processes and the creator is the *parent*.**
- **In some systems, the parent defines (or donates) resources and privileges for its children.**
- **When a child is created, the parent may either wait for it to finish its task, or continue in parallel.**
- **In Unix, subprocesses are created by a call to** Fork**; the child is identical to the parent, except for a return code from Fork. The child often begins by executing a new (and different) program within itself, via a call to** Exec. **(**Fork **and** Exec **are Unix system calls.)**
  - » this can seem like total magic.
  - » but see last week's lecture for details

## State Queues

- The OS maintains a collection of queues that represent the state of all processes in the system.
- There is typically one queue for each state, e.g., ready, waiting for I/O, ...
- Each PCB is queued onto a state queue according to its current state.
- As a process changes state, its PCB is unlinked from one queue and linked onto another.



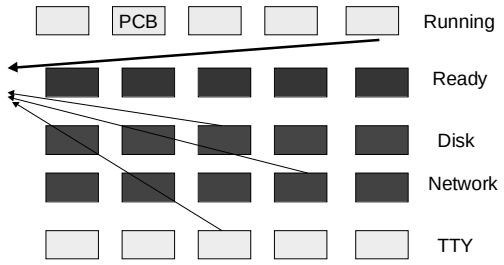
## State Queues




There may be many wait queues, one for each type of wait (wait for process message, wait for timer, wait for specific device...)



## Many Types of Queues






## PCBs and State Queues

- PCBs are data structures, dynamically allocated in OS memory.
- When a process is created, a PCB is allocated to it, initialized, and placed on the correct queue.
- As the process computes, its PCB moves from queue to queue.
- When the process is terminated, its PCB is deallocated.
- Exercise: Show the allocation and queuing that occurs during the UNIX fork() system call



## Cooperating Processes

- Processes can be independent or they can be cooperating to accomplish a single job.
- Cooperating processes can be used:
  - to gain speedup by overlapping activities or performing work in parallel
  - to better structure an application as a set of small cooperating processes
  - to share information between jobs
- Sometimes processes are structured as a pipeline where each produces work for the next stage that consumes it, and so on.
- To see how this works, we need to understand threads.
  - » that's the next lecture