“Process management 1” lecture review

- process is a program in execution - a unit of work for OS
- besides the code of the program a process has *state* (snapshot of process’ execution) which consists of - procedure stack, program’s static and dynamic data, etc.
- to keep track of process’ states OS maintains a structure called *process control block*
- time-sharing OSes interleave execution of processes giving users an illusion of simultaneous process execution
- OS may cycle through all processes giving each a chance to use CPU (two state model) - inefficient since process may be waiting on something and cannot use CPU
- five state model introduces *blocked* state where process is waiting on some event to occur

Lecture 5: Process Management 2

- Unix process management:
  - process creation
  - process scheduling
- scheduling queues
- types of schedulers
- context switch

Unix process creation

- One process can create another process, perhaps to do some work for it
  - The original process is called the *parent*
  - The new process is called the *child*
  - The child is an (almost) identical **copy** of parent (same code, same data, etc.)
  - The parent can either wait for the child to complete, or continue executing in parallel (*concurrently*) with the child
- In UNIX, a process creates a child process using the system call *fork( )*
  - In child process, fork( ) returns 0
  - In parent process, fork( ) returns process id of new child
- Child often uses *exec( )* to start another completely different program

Example of UNIX Process Creation

```
#include <sys/types.h>
#include <stdio.h>
int a = 6;          /* global (external) variable */
int main(void) {
  int b;            /* local variable */
  pid_t pid;        /* process id */
  b = 88;
  printf("..before fork\n");

  pid = fork();
  if (pid == 0) {   /* child */
    a++;  b++;
  } else            /* parent */
    wait(pid);

  printf("..after fork, a = %d, b = %d\n", a, b);
  exit(0);
}
```

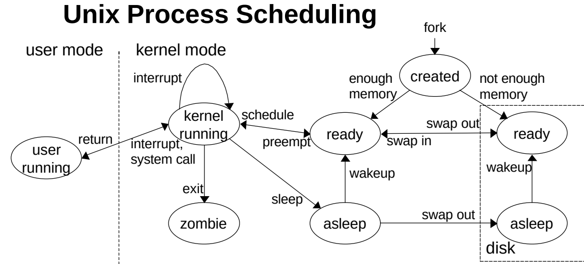
example execution

```
aegis% fork
..before fork
..after fork, a = 7, b = 89
..after fork, a = 6, b = 88
```


Unix process states

- CPU cycles can still be wasted in 5 state model: all processes in main memory can be blocked on I/O.
- solution: use virtual memory to admit more processes hoping that they will keep CPU loaded
- blocked and ready states has to be split depending on whether a process is swapped out on disk or in memory
- running state is also split depending on the mode: kernel or user
- Unix process states:
  - created - just created not yet ready to run
  - ready (memory) - ready as soon as kernel schedules it
  - ready (disk) - ready, but needs to be swapped to memory
  - asleep - blocked (memory) - waiting on event in memory
  - asleep - blocked (disk) - waiting on event on disk
  - running (kernel) - executing in kernel mode
  - running (user) - executing in user mode
  - zombie - process exited but left a record for parent to collect

Unix Process Scheduling

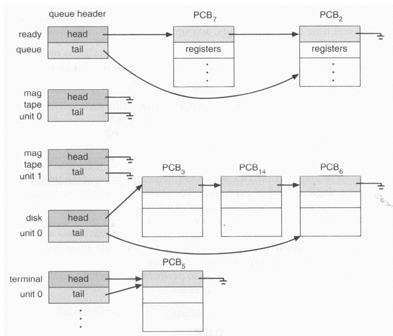


- process is running in user mode until an interrupt occurs or it executes a system call
- if time slice expires the process is *preempted* and another is *scheduled*
- a process goes to *sleep* if it needs to wait for some event to occur and is woken up when this event occurs
- when process is created decision is made whether to put it in memory or disk

Scheduling queues

OS organizes all waiting processes (their PCBs) into a number of queues:

- Queue for ready processes
- Queue for processes waiting on each device (e.g., mouse) or type of event (e.g., message)



Types of Schedulers

- Long-term scheduler (job scheduler)
  - Selects job from spooled jobs, and loads it into memory
  - Executes infrequently, maybe only when process leaves system
  - Controls degree of multiprogramming
    - Goal: good mix of CPU-bound and I/O-bound processes
  - -sharing systems
- Medium-term scheduler
  - On time-sharing systems, does some of what long-term scheduler used to do
  - May swap processes out of memory temporarily
  - May suspend and resume processes
  - Goal: balance load for better throughput

Types of schedulers (cont.)

- Short-term scheduler (CPU scheduler)
  - Executes frequently, about one hundred times per second (every 10 ms)
  - Runs whenever:
    - Process is created or terminated
    - Process switches from running to blocked
    - Interrupt occurs
  - Selects process from those that are ready to execute, allocates CPU to that process
  - Goals:
    - Minimize response time (e.g., program execution, character to screen)
    - Minimize variance of average response time — predictability may be important
    - Maximize throughput
      - Minimize overhead (OS overhead, context switching, etc.)
      - Efficient use of resources
    - Fairness — share CPU in an equitable fashion

Context switch

- Stopping one process and starting another is called a *context switch.* The state of the old process needs to be saved:
  - When the OS stops a process, it stores the hardware registers (PC, SP, etc.) and any other state information in that process' PCB
  - When OS is ready to execute a waiting process, it loads the hardware registers (PC, SP, etc.) with the values stored in the new process' PCB, and restores any other state information
  - Performing a context switch is a relatively expensive operation (1-1000 us - compare with 2-10 ns speed of the CPU – several thousand CPU cycles)
    - However, time-sharing systems may do 100–1000 context switches a second
    - Why so often?
    - Why not more often?