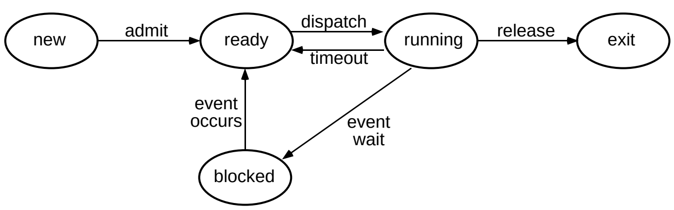
A Five-State Process Model (Review)

- The *not-running* state in the two-state model has now been split into a *ready* state and a *blocked* state
  - *Running* — currently being executed
  - *Ready* — prepared to execute
  - *Blocked* — waiting for some event to occur (for an I/O operation to complete, or a resource to become available, etc.)
  - *New* — just been created
  - *Exit* — just been terminated
- State transition diagram:



UNIX Process Model

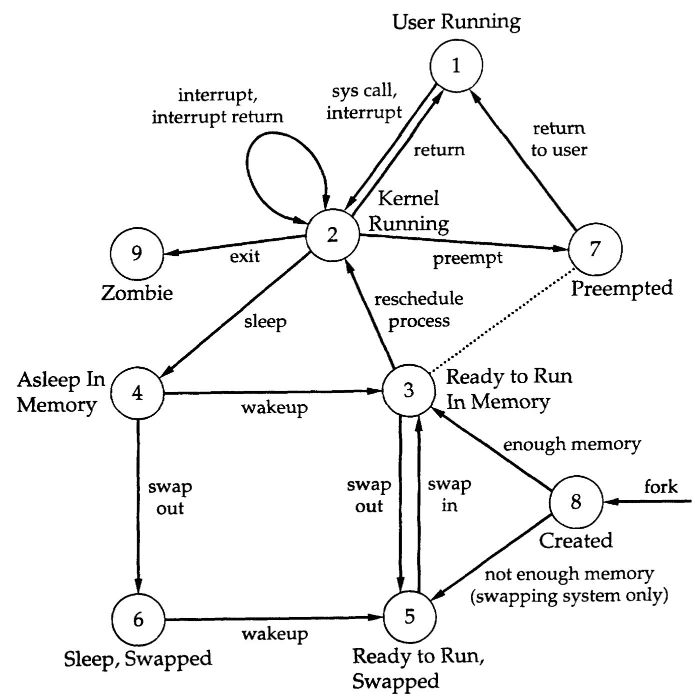


FIGURE 3.16 UNIX process state transition diagram [BACH86]

Figure from *Operating Systems*, 2nd edition, Stallings, Prentice Hall, 1995

Original diagram from *The Design of the UNIX Operating System*, M. Bach, Prentice Hall, 1986

UNIX Process Model (cont.)

- Start in **Created**, go to either:
  - **Ready to Run, in Memory**
  - or **Ready to Run, Swapped** (Out) if there isn't room in memory for the new process
  - **Ready to Run, in Memory** is basically same state as **Preempted** (dotted line)
    - **Preempted** means process was returning to user mode, but the kernel switched to another process instead
- When scheduled, go to either:
  - **User Running** (if in user mode)
  - or **Kernel Running** (if in kernel mode)
  - Go from **U.R.** to **K.R.** via system call
- Go to **Asleep in Memory** when waiting for some event, to **RtRiM** when it occurs
- Go to **Sleep, Swapped** if swapped out

Process Creation in UNIX

- 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);
}

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


Context Switching

- Stopping one process and starting another is called a *context switch*
  - 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
    - However, time-sharing systems may do 100–1000 context switches a second
    - Why so often?
    - Why not more often?

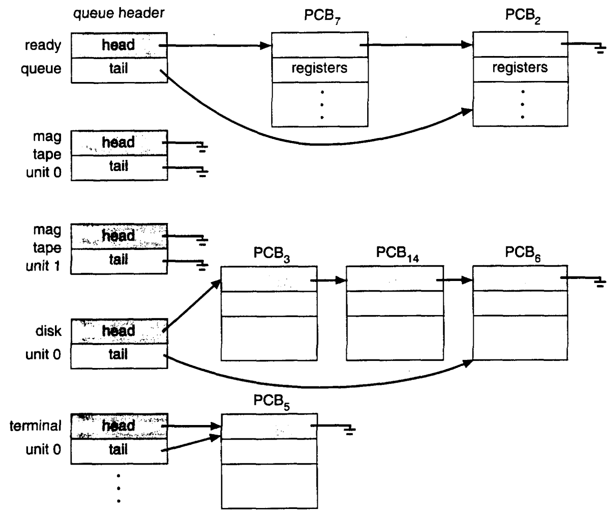
Schedulers

- 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
  - Doesn't really exist on most modern time-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

Schedulers (cont.)

- Short-term scheduler (CPU scheduler)
  - Executes frequently, about one hundred times per second (every 10ms)
  - 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

Ready Queue and Various I/O Device Queues



From *Operating System Concepts*, Silberschatz & Galvin., Addison-Wesley, 1994

- OS organizes all waiting processes (their PCBs, actually) 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)