Process

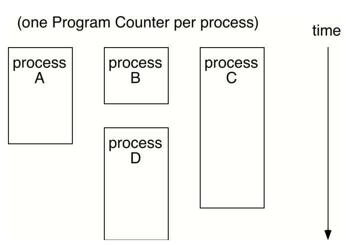
- A *process* (sometimes called a *task,* or a *job*) is, informally, a program in execution
- "Process" is not the same as "program"
  - We distinguish between a passive program stored on disk, and an actively executing process
    - Multiple people can run the same program; each running copy corresponds to a distinct process
  - The program is only part of a process; the process also contains the execution state
- List processes (HP UNIX):
  - ps — my processes, little detail
  - ps -fl — my processes, more detail
  - ps -efl — all processes, more detail
- Note user processes and OS processes

Process Creation / Termination

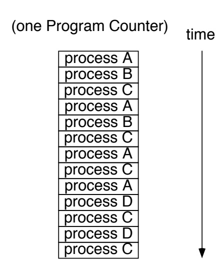
- Reasons for process creation
  - User logs on
  - User starts a program
  - OS creates process to provide a service (e.g., printer daemon to manage printer)
  - Program starts another process (e.g., netscape calls xv to display a picture)
- Reasons for process termination
  - Normal completion
  - Arithmetic error, or data misuse (e.g., wrong type)
  - Invalid instruction execution
  - Insufficient memory available, or memory bounds violation
  - Resource protection error
  - I/O failure

Process Execution

- Conceptual model of 4 processes executing:

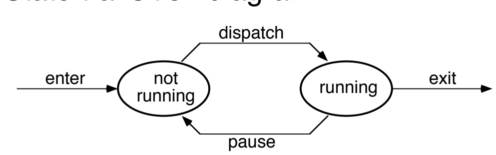


- Actual interleaved execution of the 4 processes:

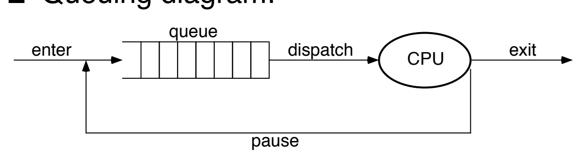


A Two-State Process Model

- This process model says that either a process is *running*, or it is *not running*
- State transition diagram:



- Queuing diagram:



- CPU scheduling (round-robin)
  - Queue is first-in, first-out (FIFO) list
  - CPU scheduler takes process at head of queue, runs it on CPU for one time slice, then puts it back at tail of queue

Process Transitions in the Two-State Process Model

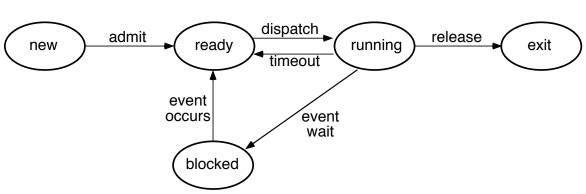
- When the OS creates a new process, it is initially placed in the **not-running** state
  - It's waiting for an opportunity to execute
- At the end of each time slice, the *CPU scheduler* selects a new process to run
  - The previously running process is *paused* — moved from the **running** state into the **not-running** state (at tail of queue)
  - The new process (at head of queue) is *dispatched* — moved from the **not-running** state into the **running** state
    - If the running process completes its execution, it exits, and the CPU scheduler is invoked again
    - If it doesn't complete, but its time is up, it gets moved into the **not-running** state anyway, and the CPU scheduler chooses a new process to execute

Waiting on Something to Happen…

- Some reasons why a process that might otherwise be running needs to wait:
  - Wait for user to type the next key
  - Wait for output to appear on the screen
  - Program tried to read a file — wait while OS decides which disk blocks to read, and then actually reads the requested information into memory
  - Netscape tries to follow a link (URL) — wait while OS determines address, requests data, reads packets, displays requested web page
- OS must distinguish between:
  - Processes that are ready to run and are waiting their turn for another time slice
  - Processes that are waiting for something to happen (OS operation, hardware event, etc.)

A Five-State Process Model

- 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:



State Transitions in Five-State Process Model

- new → ready
  - Admitted to ready queue; can now be considered by CPU scheduler
- ready → running
  - CPU scheduler chooses that process to execute next, according to some scheduling algorithm
- running → ready
  - Process has used up its current time slice
- running → blocked
  - Process is waiting for some event to occur (for I/O operation to complete, etc.)
- blocked → ready
  - Whatever event the process was waiting on has occurred

Process State

- The *process state* consists of (at least):
  - Code for the program
  - Program's static and dynamic data
  - Program's procedure call stack
  - Contents of general purpose registers
  - Contents of Program Counter (PC)
  - Contents of Stack Pointer (SP)
  - Contents of Program Status Word (PSW) — interrupt status, condition codes, etc.
  - OS resources in use (e.g., memory, open files, active I/O devices)
  - Accounting information (e.g., CPU scheduling, memory management)

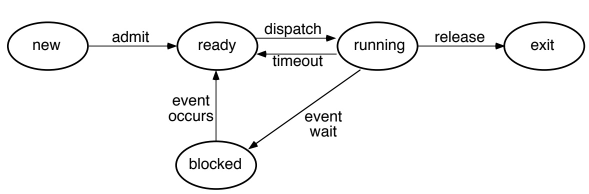
↪ Everything necessary to resume the process' execution if it is somehow put aside temporarily

Process Control Block (PCB)

- For every process, the OS maintains a *Process Control Block* (*PCB*), a data structure that represents the process and its state:
  - Process id number
  - Userid of owner
  - Memory space (static, dynamic)
  - Program Counter, Stack Pointer, general purpose registers
  - Process state (running, not-running, etc.)
  - CPU scheduling information (e.g., priority)
  - List of open files
  - I/O states, I/O in progress
  - Pointers into CPU scheduler's state queues (e.g., the waiting queue)
  - …

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

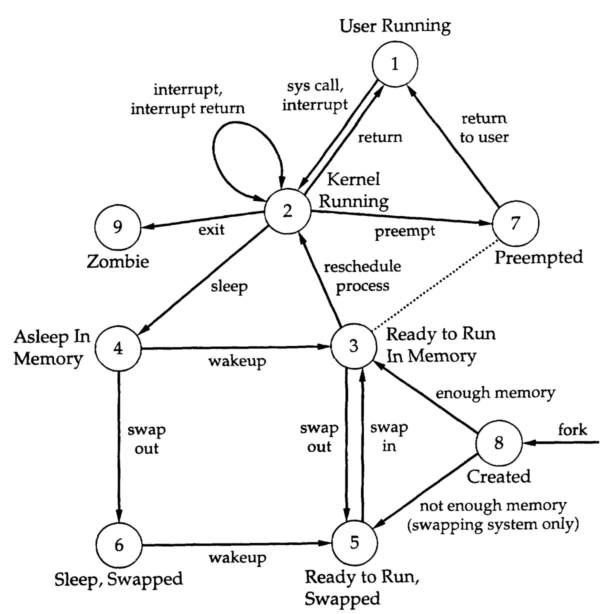


FIGURE 3.16 UNIX process state transition diagram [BACH86]

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

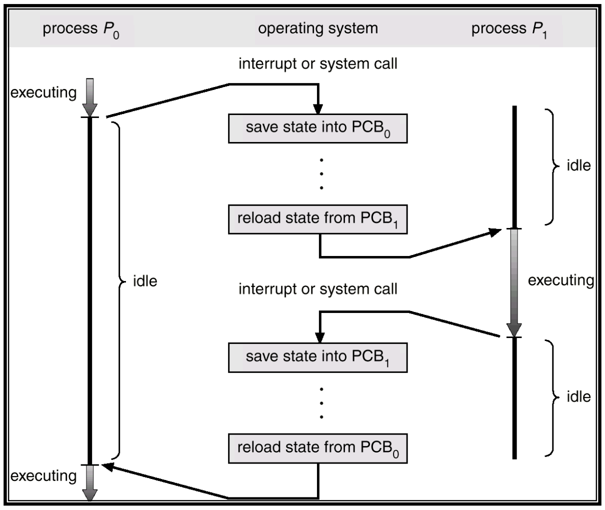
Example of UNIX Process Creation

```
#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?

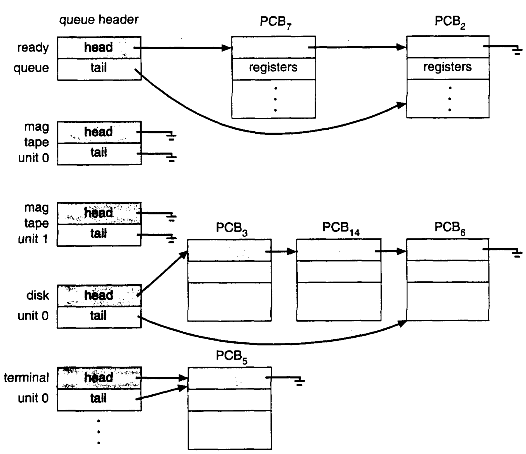
Context Switching



Schedulers

- Medium-term scheduler (demand paging)
  - 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
- 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

Ready Queue and Various I/O Device Queues



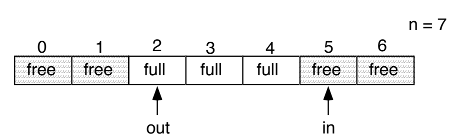
From *Operating System Concepts*, Silberschatz & Galvin., Addison-Wesley, 1994

- 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)

The Producer-Consumer Problem

- One process is a producer of information; another is a consumer of that information
- Processes communicate through a bounded (fixed-size) circular buffer

```
var buffer:  array[0..n-1] of items;   /* circular array */
in = 0
out = 0
```




```
/* producer */
repeat forever
        …
        produce item nextp
        …
        while (in+1 mod n == out)
                do nothing
        buffer[in] = nextp
        in = in+1 mod n
end repeat
```

```
/* consumer */
repeat forever
        while (in == out)
                do nothing
        nextc = buffer[out]
        out = out+1 mod n
        …
        consume item nextc
        …
end repeat
```

Message Passing using Send & Receive

- Blocking send:
  - send(*destination-process*, *message*)
  - Sends a message to another process, then *blocks* (i.e., gets suspended by OS) until message is received
- Blocking receive:
  - receive(*source-process*, *message*)
  - Blocks until a message is received (may be minutes, hours, …)
- Producer-Consumer problem:

```
/* producer */
repeat forever
    …
    produce item nextp
    …
    send(consumer, nextp)
end repeat
```

```
/* consumer */
repeat forever
    receive(producer,nextc)
    …
    consume item nextc
    …
end repeat
```

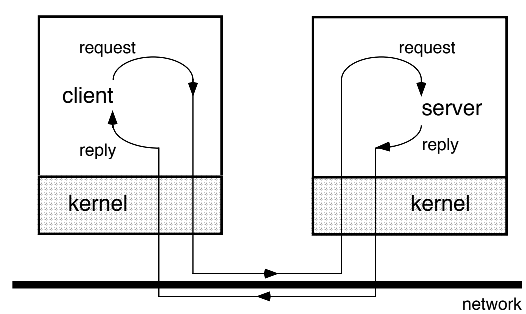

Direct vs. Indirect Communication

- Direct communication — explicitly name the process you're communicating with
  - send(*destination-process*, *message*)
  - receive(*source-process*, *message*)
  - Variation: receiver may be able to use a "wildcard" to receive from any source
  - Receiver <u>can not</u> distinguish between multiple "types" of messages from sender
- Indirect communication — communicate using mailboxes (owned by receiver)
  - send(*mailbox*, *message*)
  - receive(*mailbox*, *message*)
  - Variation: … "wildcard" to receive from any source into that mailbox
  - Receiver <u>can</u> distinguish between multiple "types" of messages from sender
  - Some systems use "tags" instead of mailboxes

Buffering

- Link may be able to temporarily queue some messages during communication
- Zero capacity: (queue of length 0)
  - Blocking send operation
    - Sender must wait until receiver receives the message — this synchronization to exchange data is called a *rendezvous*
- Bounded capacity: (queue of length *n*)
  - Blocking send operation
    - If receiver's queue is has free space, new message is put on queue, and sender can continue executing immediately
    - If queue is full, sender must block until space is available in the queue
- Unbounded capacity: (infinite queue)
  - Non-blocking send operation
    - Sender can always continue

Client / Server Model using Message Passing

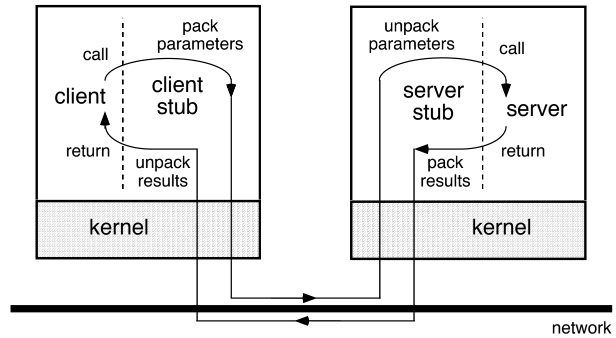


- Client / server model
  - *Server* = process (or collection of processes) that provides a *service*
    - Example: name service, file service
  - *Client* — process that uses the service
  - Request / reply protocol:
    - Client sends **request** message to server, asking it to perform some service
    - Server performs service, sends **reply** message containing results or error code

Remote Procedure Call (RPC)

- RPC mechanism:
  - Hides message-passing I/O from the programmer
  - Looks (almost) like a procedure call — but client invokes a procedure on a server
- RPC invocation (high-level view):
  - Calling process (client) is suspended
  - Parameters of procedure are passed across network to called process (server)
  - Server executes procedure
  - Return parameters are sent back across network
  - Calling process resumes
- Invented by Birrell & Nelson at Xerox PARC, described in February 1984 *ACM Transactions on Computer Systems*

Client / Server Model using Remote Procedure Calls (RPCs)



- Each RPC invocation by a client process calls a *client stub*, which builds a message and sends it to a *server stub*
- The server stub uses the message to generate a local procedure call to the server
- If the local procedure call returns a value, the server stub builds a message and sends it to the client stub, which receives it and returns the result(s) to the client

Remote Method Invocation (RMI)

- RMI mechanism:
  - A Java mechanism similar to RPCs
  - Allows a Java program on one machine to invoke a method on a remote object
  - Client *stub* creates a *parcel*, sends to *skeleton* on the server side