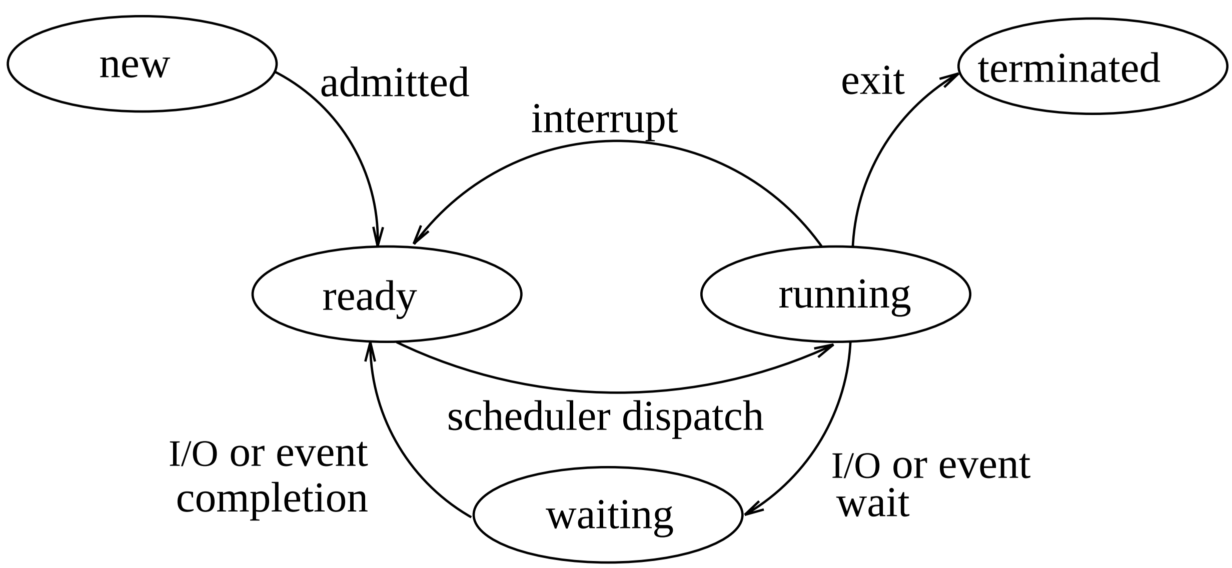
CHAPTER 4: PROCESSES

- Process Concept
- Process Scheduling
- Operation on Processes
- Cooperating Processes
- Threads
- Interprocess Communication

Process Concept

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Textbook uses the terms *job* and *process* almost interchangeably.
- Process – a program in execution; process execution must progress in a sequential fashion.
- A process includes:
  - *program counter*
  - *stack*
  - *data section*

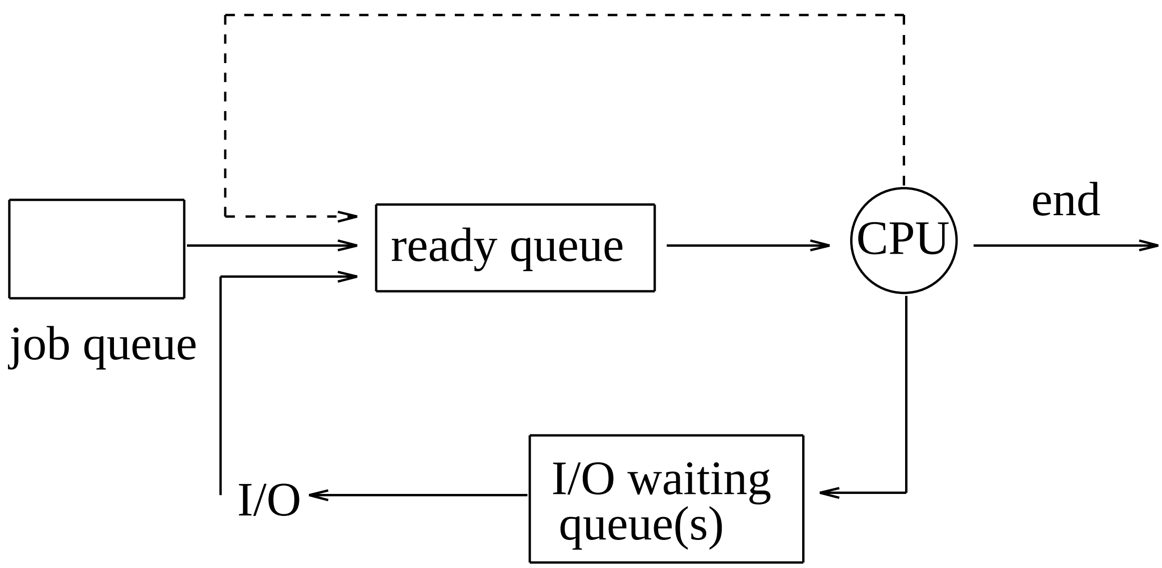
- As a process executes, it changes *state*.
  - **New:** The process is being created.
  - **Running:** Instructions are being executed.
  - **Waiting:** The process is waiting for some event to occur.
  - **Ready:** The process is waiting to be assigned to a processor.
  - **Terminated:** The process has finished execution.
- Diagram of process state:



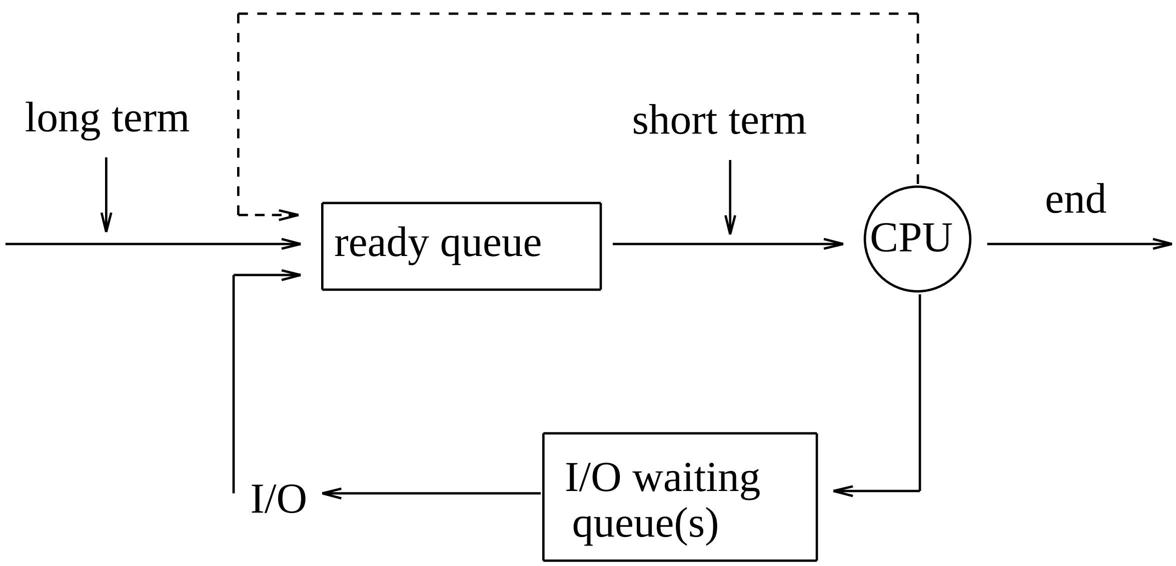
- Process Control Block (PCB) – Information associated with each process.
  - Process state
  - Program counter
  - CPU registers
  - CPU scheduling information
  - Memory-management information
  - Accounting information
  - I/O status information

- Process scheduling queues
  - *job queue* – set of all processes in the system.
  - *ready queue* – set of all processes residing in main memory, ready and waiting to execute.
  - *device queues* – set of processes waiting for a particular I/O device.

- Process migration between the various queues.



- Schedulers
  - *Long-term scheduler* (*job scheduler*) – selects which processes should be brought into the ready queue.
  - *Short-term scheduler* (*CPU scheduler*) – selects which process should be executed next and allocates CPU.



- Short-term scheduler is invoked very frequently (milliseconds) $\Rightarrow$ (must be fast).

- Long-term scheduler is invoked very infrequently (seconds, minutes) $\Rightarrow$ (may be slow).

- The long-term scheduler controls the *degree of multiprogramming*.

- Processes can be described as either:

  - *I/O-bound process* – spends more time doing I/O than computations; many short CPU bursts.

  - *CPU-bound process* – spends more time doing computations; few very long CPU bursts.

Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process.

- Context-switch time is overhead; the system does no useful work while switching.

- Time dependent on hardware support.

Process Creation

- Parent process creates children processes, which, in turn create other processes, forming a tree of processes.

- Resource sharing
  - Parent and children share all resources.
  - Children share subset of parent's resources.
  - Parent and child share no resources.

- Execution
  - Parent and children execute concurrently.
  - Parent waits until children terminate.

- Address space
  - Child duplicate of parent.
  - Child has a program loaded into it.

- UNIX examples
  - **fork** system call creates new process.
  - **execve** system call used after a **fork** to replace the process' memory space with a new program.

Process Termination

- Process executes last statement and asks the operating system to delete it (**exit**).
  - Output data from child to parent (via **fork**).
  - Process' resources are deallocated by operating system.

- Parent may terminate execution of children processes (**abort**).
  - Child has exceeded allocated resources.
  - Task assigned to child is no longer required.
  - Parent is exiting.
    - Operating system does not allow child to continue if its parent terminates.
    - *Cascading termination.*

Cooperating Processes

- *Independent* process cannot affect or be affected by the execution of another process.

- *Cooperating* process can affect or be affected by the execution of another process.

- Advantages of process cooperation:
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Producer-Consumer Problem

- Paradigm for cooperating processes; *producer* process produces information that is consumed by a *consumer* process.
  - *unbounded-buffer* places no practical limit on the size of the buffer.
  - *bounded-buffer* assumes that there is a fixed buffer size.

- Shared-memory solution:
  - Shared data

    var *n;*
    type *item* = ... ;
    var *buffer*: **array** [0..*n*–1] **of** *item*;
    in, *out*: 0..*n*–1;
    in := 0;
    out := 0;

- Producer process

```
repeat
        ...
    produce an item in nextp
        ...
    while in+1 mod n = out do no-op;
    buffer[in] := nextp;
    in := in+1 mod n;
until false;
```

- Consumer process

```
repeat
    while in = out do no-op;
    nextc := buffer[out];
    out := out+1 mod n;
            ...
    consume the item in nextc
            ...
until false;
```

- Solution is correct, but can only fill up $n-1$ buffer.

Threads

- A *thread* (or *lightweight process*) is a basic unit of CPU utilization; it consists of:
  - program counter
  - register set
  - stack space

- A thread shares with its peer threads its:
  - code section
  - data section
  - operating-system resources

  collectively known as a *task*.

- A traditional or *heavyweight* process is equal to a task with one thread.

- In a task containing multiple threads, while one server thread is blocked and waiting, a second thread in the same task could run.
  - Cooperation of multiple threads in same job confers higher throughput and improved performance.
  - Applications that require sharing a common buffer (producer–consumer problem) benefit from thread utilization.
- Threads provide a mechanism that allows sequential processes to make blocking system calls while also achieving parallelism.
- Kernel-supported threads (Mach and OS/2).
- User-level threads; supported above the kernel, via a set of library calls at the user level (Project Andrew from CMU).
- Hybrid approach implements both user-level and kernel-supported threads (Solaris 2).

Solaris 2 – version of UNIX with support for threads at the kernel and user levels, symmetric multiprocessing, and real-time scheduling.

- LWP – intermediate level between user-level threads and kernel-level threads.

- Resource needs of thread types:

  - Kernel thread – small data structure and a stack; thread switching does not require changing memory access information, and therefore is relatively fast.

  - LWP – PCB with register data, accounting information, and memory information; switching between LWPs is relatively slow.

  - User-level thread – needs only a stack and a program counter. Switching is fast since kernel is not involved. Kernel only sees the LWPs in the process that support user-level threads.

Interprocess Communication (IPC) – provides a mechanism to allow processes to communicate and to synchronize their actions.

- Message system – processes communicate with each other without resorting to shared variables.

- IPC facility provides two operations:
  - **send***(message)* – messages can be of either fixed or variable size.
  - **receive***(message)*

- If P and Q wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive

- Communication link
  - physical implementation (e.g., shared memory, hardware bus)
  - logical implementation (e.g., logical properties)

Implementation questions:

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bidirectional?

Direct Communication

- Processes must name each other explicitly:
  - **send***(P, message)* – send a message to process P
  - **receive***(Q, message)* – receive a message from process Q
- Properties of communication link
  - Links are established automatically.
  - A link is associated with exactly one pair of communicating processes.
  - Between each pair there exists exactly one link.
  - The link may be unidirectional, but is usually bidirectional.

Indirect Communication

- Messages are directed and received from *mailboxes* (also referred to as *ports*).
  - Each mailbox has a unique *id*.
  - Processes can communicate only if they share a mailbox.
- Properties of communication link
  - Link established only if the two processes share a mailbox in common.
  - A link may be associated with many processes.
  - Each pair of processes may share several communication links.
  - Link may be unidirectional or bidirectional.
- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox

Indirect Communication (Continued)

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A.
  - P_1 sends; P_2 and P_3 receive.
  - Who gets the message?

- Solutions
  - Allow a link to be associated with at most two processes.
  - Allow only one process at a time to execute a receive operation.
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Buffering – queue of messages attached to the link; implemented in one of three ways.

- Zero capacity – 0 messages
  Sender must wait for receiver (*rendezvous*).

- Bounded capacity – finite length of *n* messages
  Sender must wait if link full.

- Unbounded capacity – infinite length
  Sender never waits.

Exception Conditions – error recovery

- Process terminates
- Lost messages
- Scrambled Messages