Processes

Process concept

- A process is a program in execution
  - Program is a sequence of instruction in a file (executable file) on the disk
  - Program is passive
  - Process is “alive”
  - A single program can have multiple process
  - A single process can spawn multiple processes
- Batch systems – jobs
- Time-sharing systems – user programs or tasks

processes

Process concept

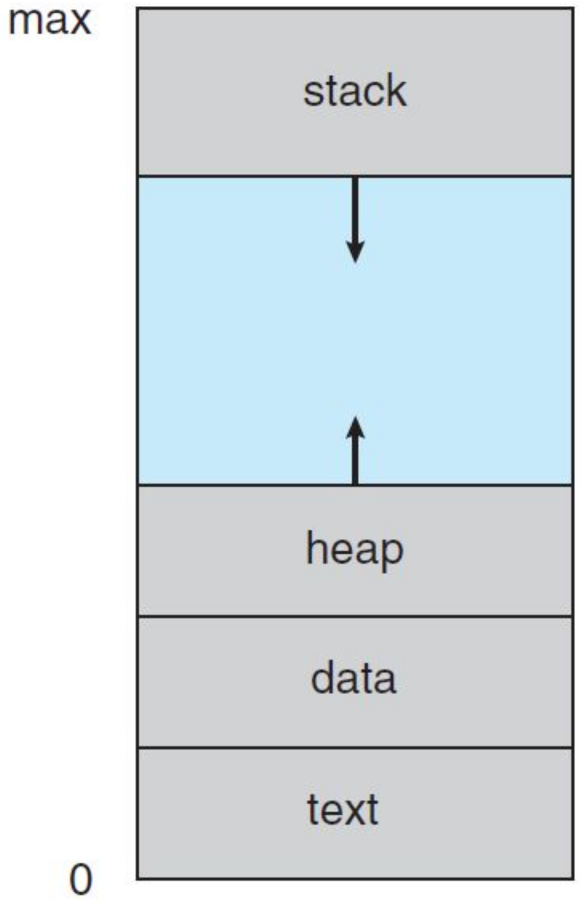
- A process includes
  - program counter – address of next instruction
  - contents of CPU registers

  CPU

  - text section
  - stack
  - heap
  - data section

  Memory

Process concept – memory



facilitates function calls within the process

dynamically allocated data

data fixed at compile time

program instructions

Single program with multiple process

- Each process has distinct
  - program counter
  - CPU register contents
  - stack
  - heap
  - data section
- The processes share the text section only

Process as an execution environment

- A process can act as an execution environment for another process
  - Emulation
- Example: JVM

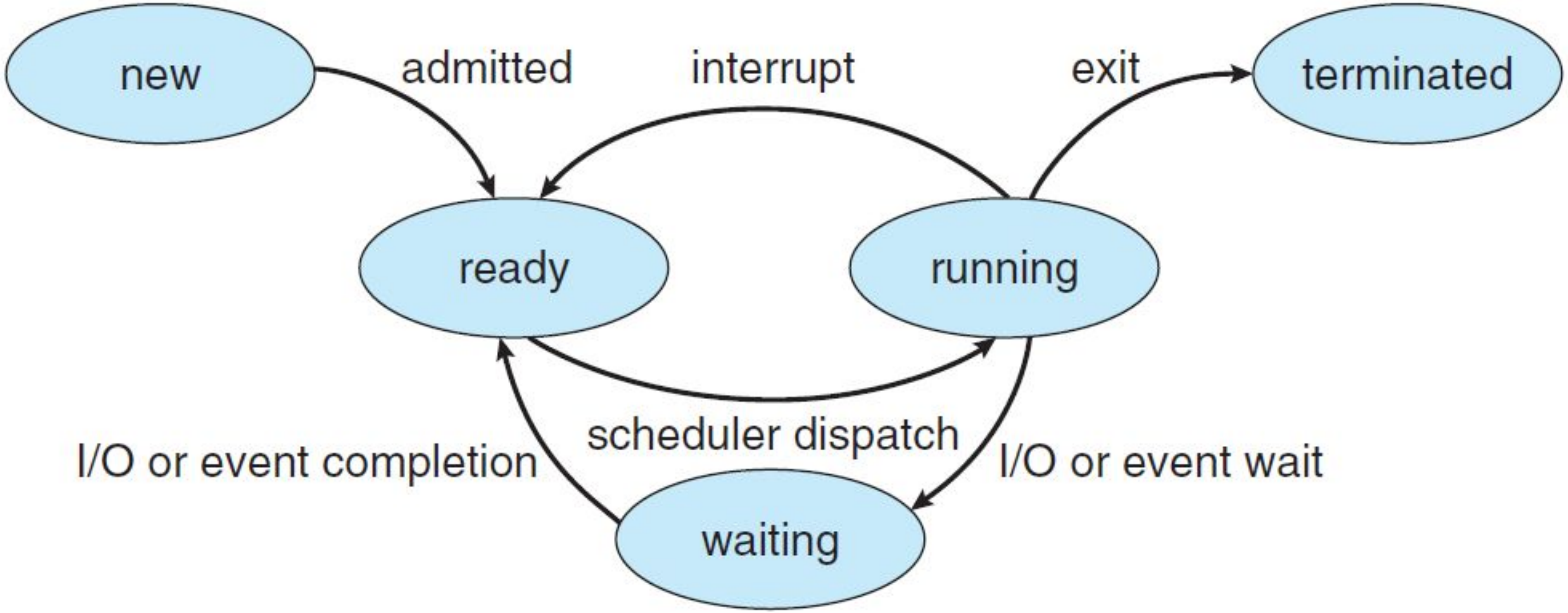
  `java myProgam`

  - **`java`** runs the JVM process which is responsible interpreting **`myProg`** (a processe also in memory) instructions into native instructions on the CPU

Process states

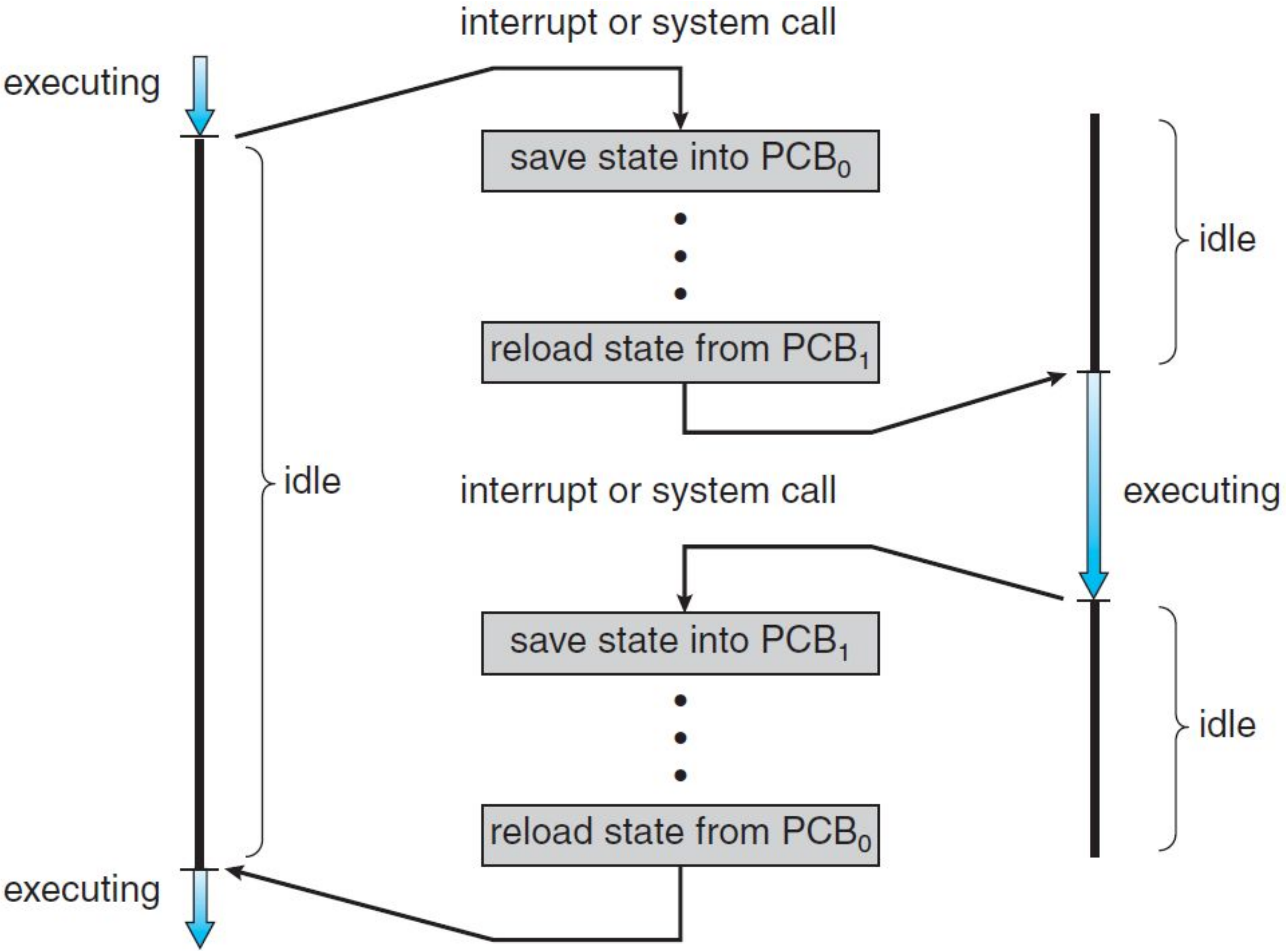
- **new**
- **running** – currently executing in the CPU
- **waiting** – waiting for some event
- **ready –** waiting for a turn to use the CPU
- **terminated**

Process states



Process control block (PCB)

- Also called **task control block**
- OS maintains a PCB for each process
- PCB contains information about the process including
  - process state
  - CPU register contents
  - CPU scheduling information
  - Memory management info – e.g. values of base address and limit registers etc
  - Accounting info: real time used, time limits, process numbers, etc
  - I/O status information: lists of allocated devices, open files, etc


executing
interrupt or system call
save state into PCB_0
reload state from PCB_1
idle
idle
executing
interrupt or system call
save state into PCB_1
reload state from PCB_0
idle
executing

Threads

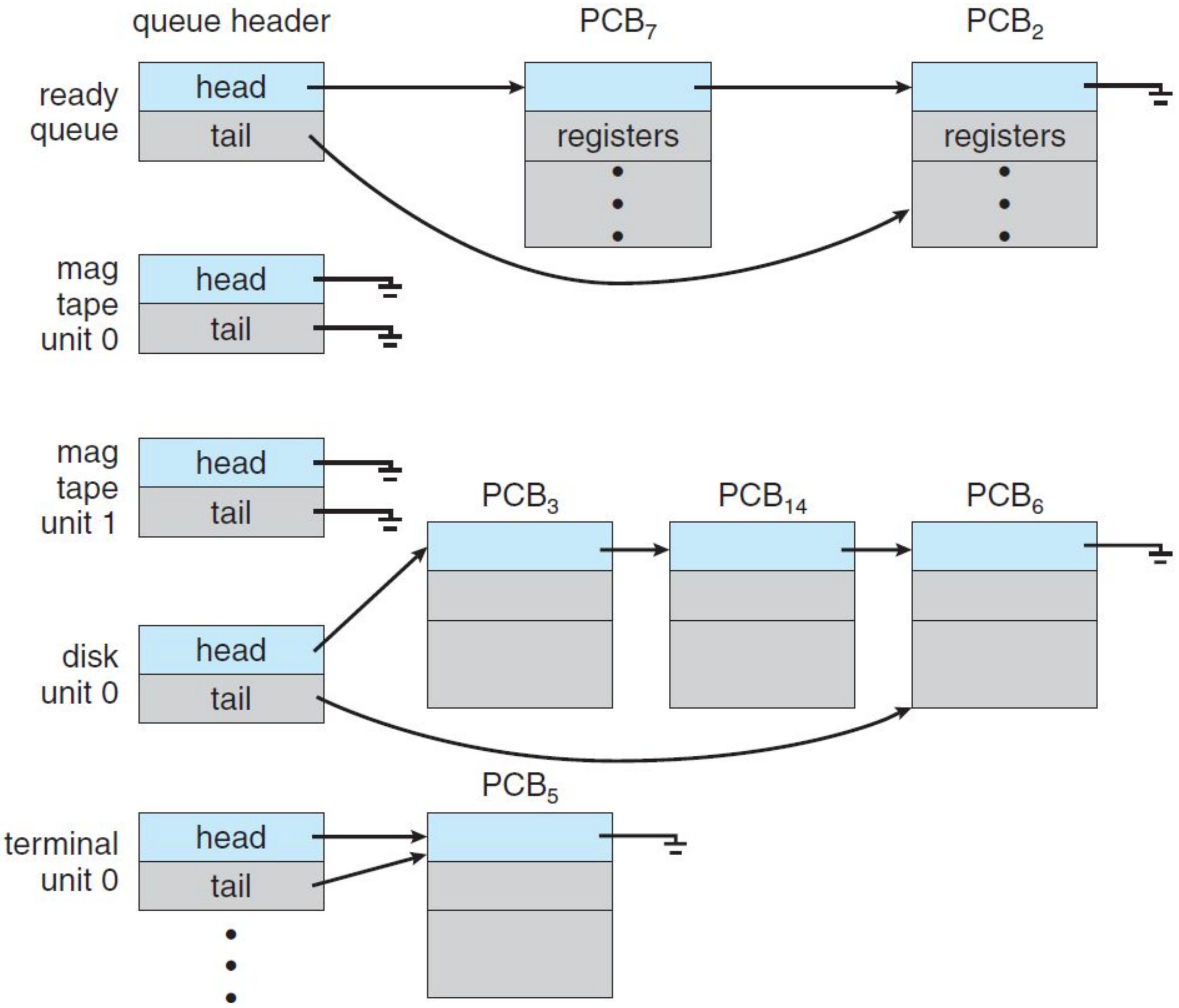
- A process is a single instance of a program in execution
- A process can be multithreaded – multiple separate sequences of instructions, each requiring a separate program counter, etc
- PCBs for multithreaded process are expanded to include additional information
- Multithreaded processes can take advantage of multiprocessor system to run threads in parallel

Process scheduling

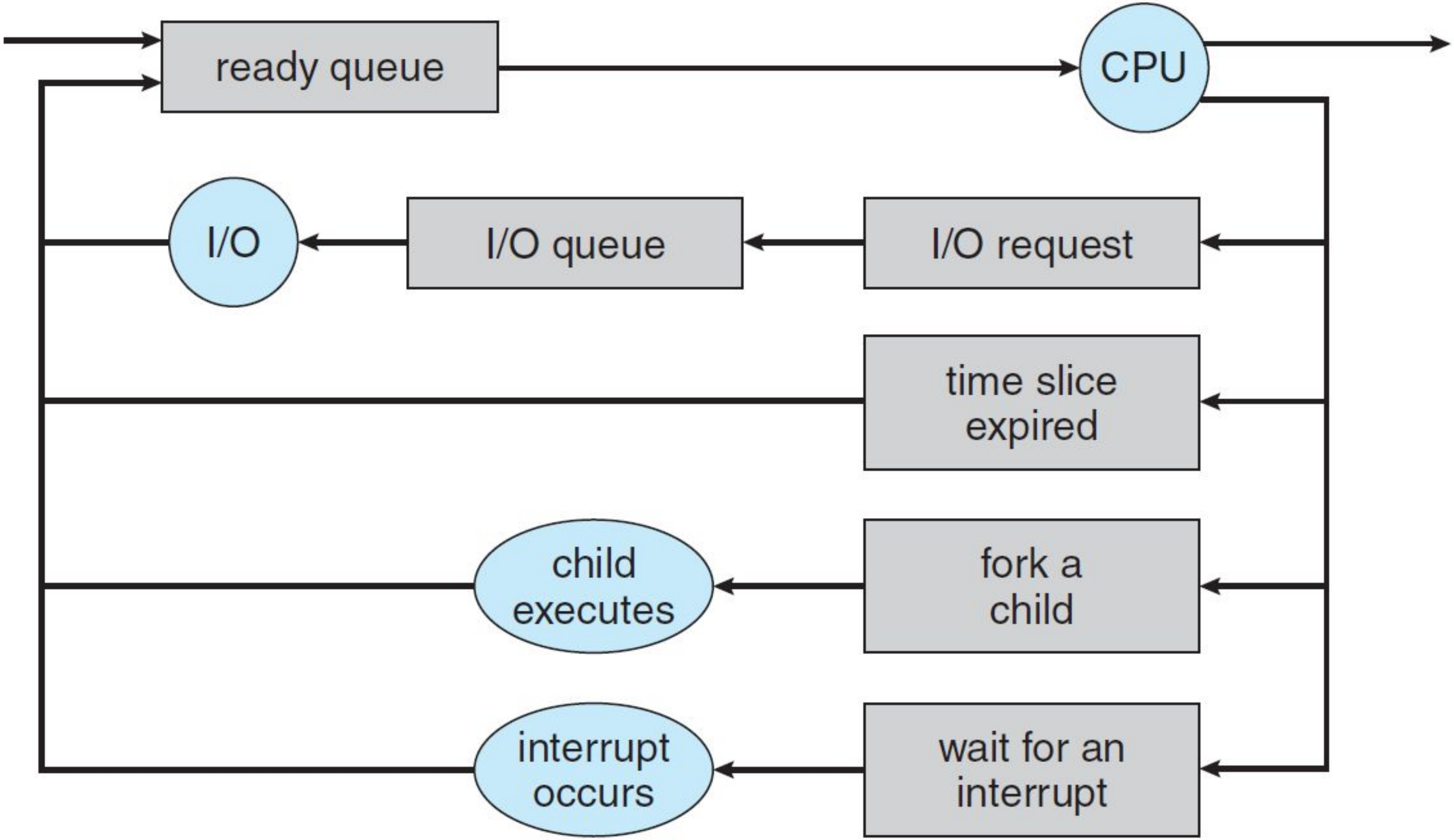
- A must in a multiprogramming environment
- Multiprogramming promotes high resource utilization
  - E.g. Let another process run while the current one is waiting for IO
- Allows user to run multiple applications “simultaneously”
- Allows multiple users to use a system “concurrently”
- A single CPU can only execute one process at any given time
  - The other processes must wait until **process scheduler** select one for execution

Scheduling queues

- OS maintains several queues for processes waiting for some resource
  - Job queue – newly created processes
  - Ready queue
  - Device queues
- Each queue is implemented as a linked-list of PCBs
  - Queue header has pointers to first and last PCB



queue header
PCB7
PCB2
ready queue
head
tail
registers
registers
mag tape unit 0
head
tail
mag tape unit 1
head
tail
PCB3
PCB14
PCB6
disk unit 0
head
tail
PCB5
terminal unit 0
head
tail


ready queue
CPU
I/O
I/O queue
I/O request
time slice expired
child executes
fork a child
interrupt occurs
wait for an interrupt

Schedulers

- Long-term scheduler (job scheduler)
  - For jobs spooled on the disk
  - Selects jobs from disk for loading into memory
  - Controls **degree of multiprogramming** – number of processes in memory
  - Low frequency of use
- Short-term scheduler (CPU scheduler)
  - Selects, from the ready queue, a process to run on the CPU
  - Called frequently – typically, every few milliseconds
  - Must decide very quickly, otherwise it will waste CPU cycles

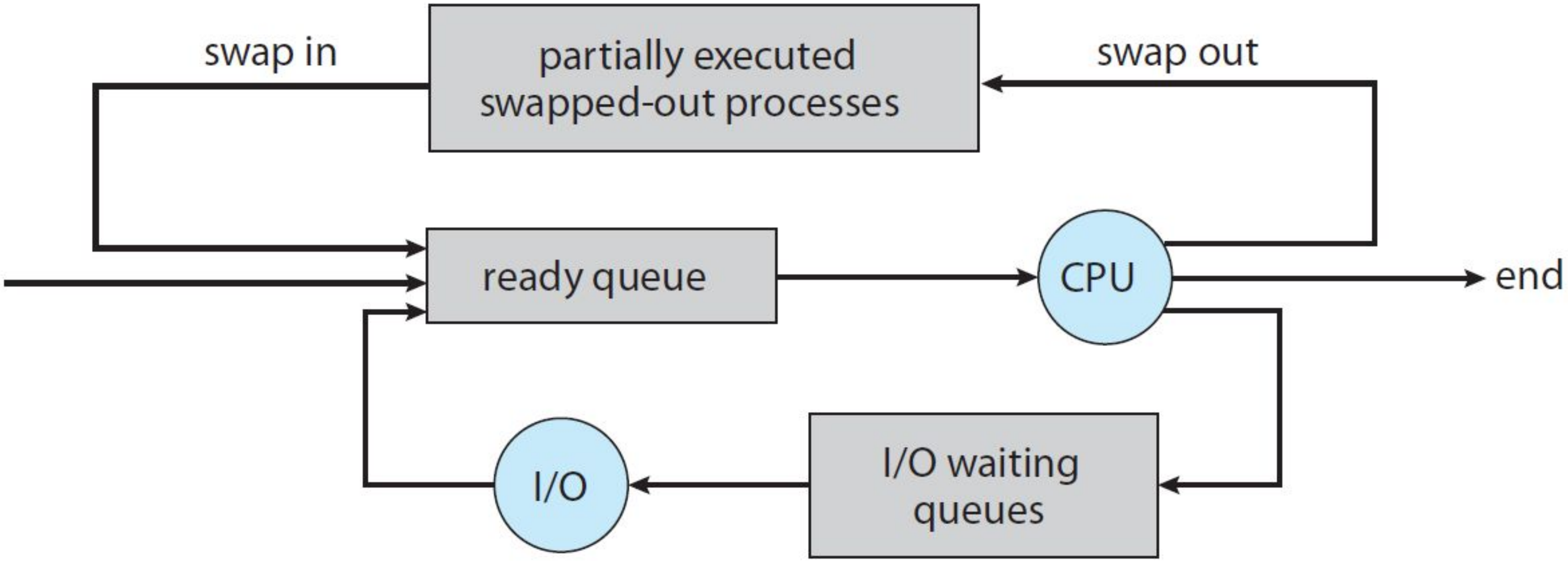
Long-term scheduler

- I/O bound processes – processes that spend most of their time doing I/O
- CPU bound processes – processes that are computation intensive
- long-term scheduler must balance between the two
  - if too many CPU bound processes – I/O devices will be idle
  - if too many I/O bound processes – CPU will be mostly idle
- Some OSs (e.g. Windows and Unix) don’t have long-term scheduler
  - new processes immediately join the ready queue
  - degree of multiprogramming is controlled by self adjusting nature of users

Medium-term scheduler

- In OSs that support swapping
- Invoked when the degree of multiprogramming becomes to high for the available amount of memory
  - Some processes are swapped out and later swapped back in when memory becomes more available
- Might also be used to balance the process mix

Medium-term scheduler in context



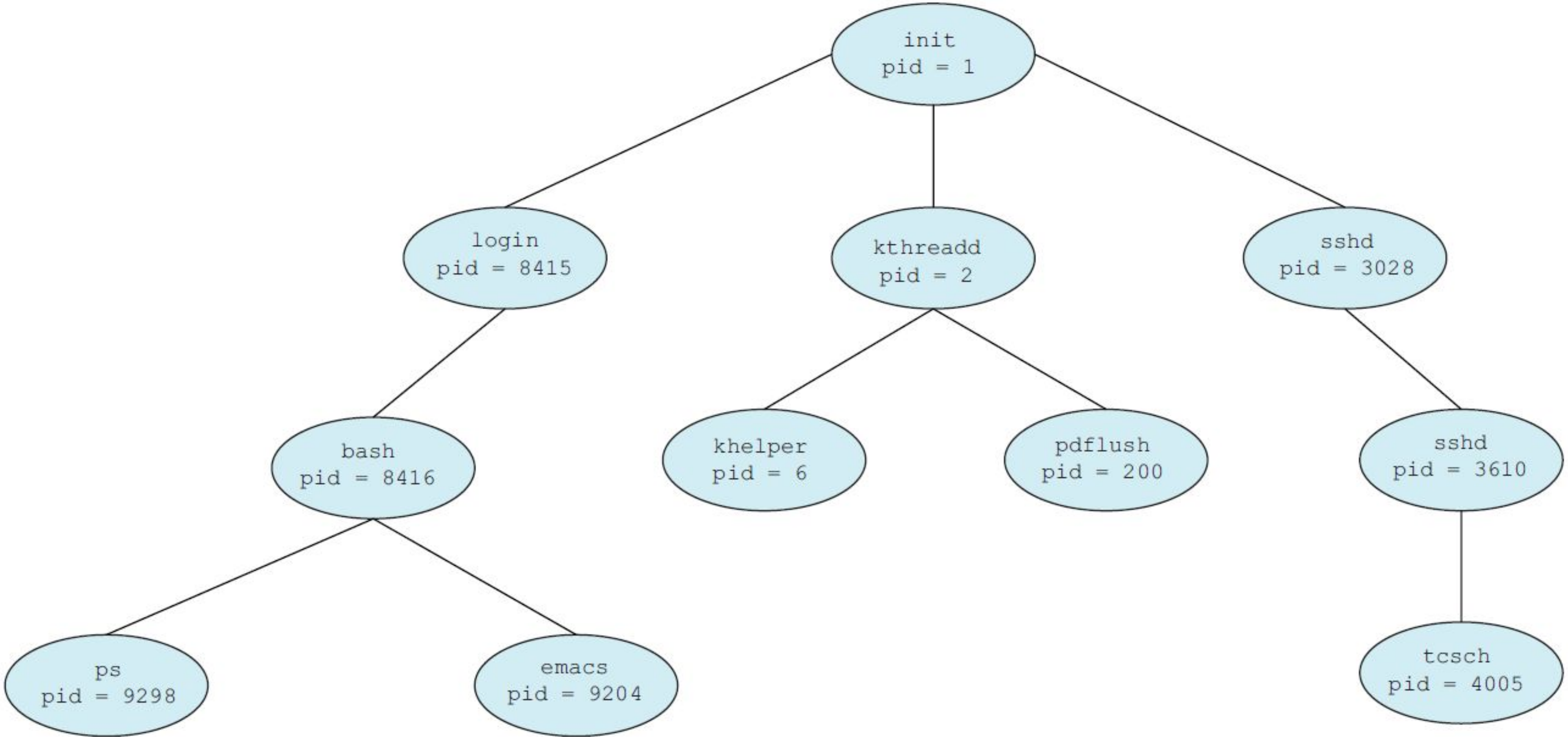
Context switch

- Context = the contents of a PCB
- context switch happens at each interrupt or system call
  - state save of the outgoing process
  - state restore of the incoming process
- Speed of a context switch depend on
  - complexity of the OS
  - the hardware architecture
    - E.g. some CPU have register sets – can simply switch to another register set instead of copying register values to memory (in PCB)

Operations on Processes - Creation

- Parent/child processes
  - Process tree
- Each process is identified by a process ID (PID)
- After a process creates a new process, either
  - Concurrent execution of parent process and its children
  - Parent process waits for the child process to complete execution
- Address space possibilities
  - Child process duplicates parent's address space
  - Child process loads a new program

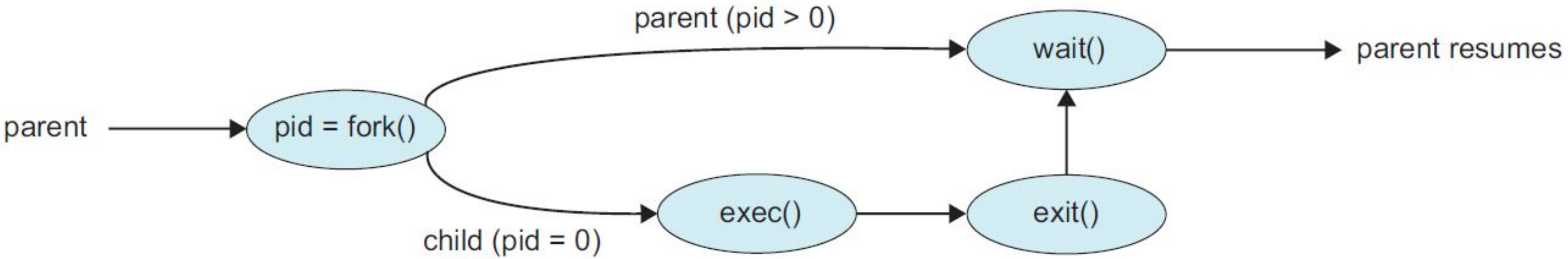
Process tree example: UNIX



Process creation example: UNIX

- **`fork()`** – system call to create a child process
  - Creates a child process with a copy of the parent's address space
  - parent and child run concurrently, unless parent chooses to call **`wait()`**
    - **`fork()`** in parent returns PID of the newly created process
    - **`fork()`** in child return 0
- **`exec()`** – loads a new program in a process

Process creation example: UNIX



```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main(){
    pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }else if (pid == 0) { /* child process */
        execlp("/bin/ls","ls",NULL);
    }else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }

    return 0;
}
```

Process creation example: Windows

- **`CreateProcess()`** – system call to create process
  - has 8 parameters
    - includes the program that must be run in the process

```
#include <stdio.h>
#include <windows.h>
int main(VOID){
    STARTUPINFO si; PROCESS INFORMATION pi;
    /* allocate memory */
    ZeroMemory(&si, sizeof(si)); si.cb = sizeof(si); ZeroMemory(&pi, sizeof(pi));
    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
            "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
            NULL, /* don’t inherit process handle */
            NULL, /* don’t inherit thread handle */
            FALSE, /* disable handle inheritance */
            0, /* no creation flags */
            NULL, /* use parent’s environment block */
            NULL, /* use parent’s existing directory */
            &si, &pi)){
        fprintf(stderr, "Create Process Failed"); return -1;
    }
    WaitForSingleObject(pi.hProcess, INFINITE); /* parent to wait for child */
    printf("Child Complete");
    /* close handles */ CloseHandle(pi.hProcess); CloseHandle(pi.hThread);
}
```

Operations on process: termination

- exit() – system call made by process to request OS to delete it
    - May return a value (typically an integer) representing status to parent process – assuming parent called wait() on the process
    - Deallocate all resources
    - Can be called explicitly
    - Can also be called implicitly – return statement
- TerminateProcess() – system call to terminate a given process
    - Windows OS
    - A process can only terminate its children

Operations on process: termination

- Use cases for terminating a process
  - Child exceed usage of an allocated resource
  - Task assigned to child is no longer needed
  - Parent process is exiting (**Cascading termination**) - some OSs do not allow orphans
- Zombie processes – a process that has terminated, but, its parent has not yet called wait
  - Its entry in the process table still exists – so that the parent is able to get the return status
  - Can only be deleted from the process table if a parent calls wait()

Operations on process: termination

Child process

```
/* exit with status 1 */
exit(1);
```

Parent process

```
pid_t pid;
int status;
pid = wait(&status);
```

Operations on process: termination

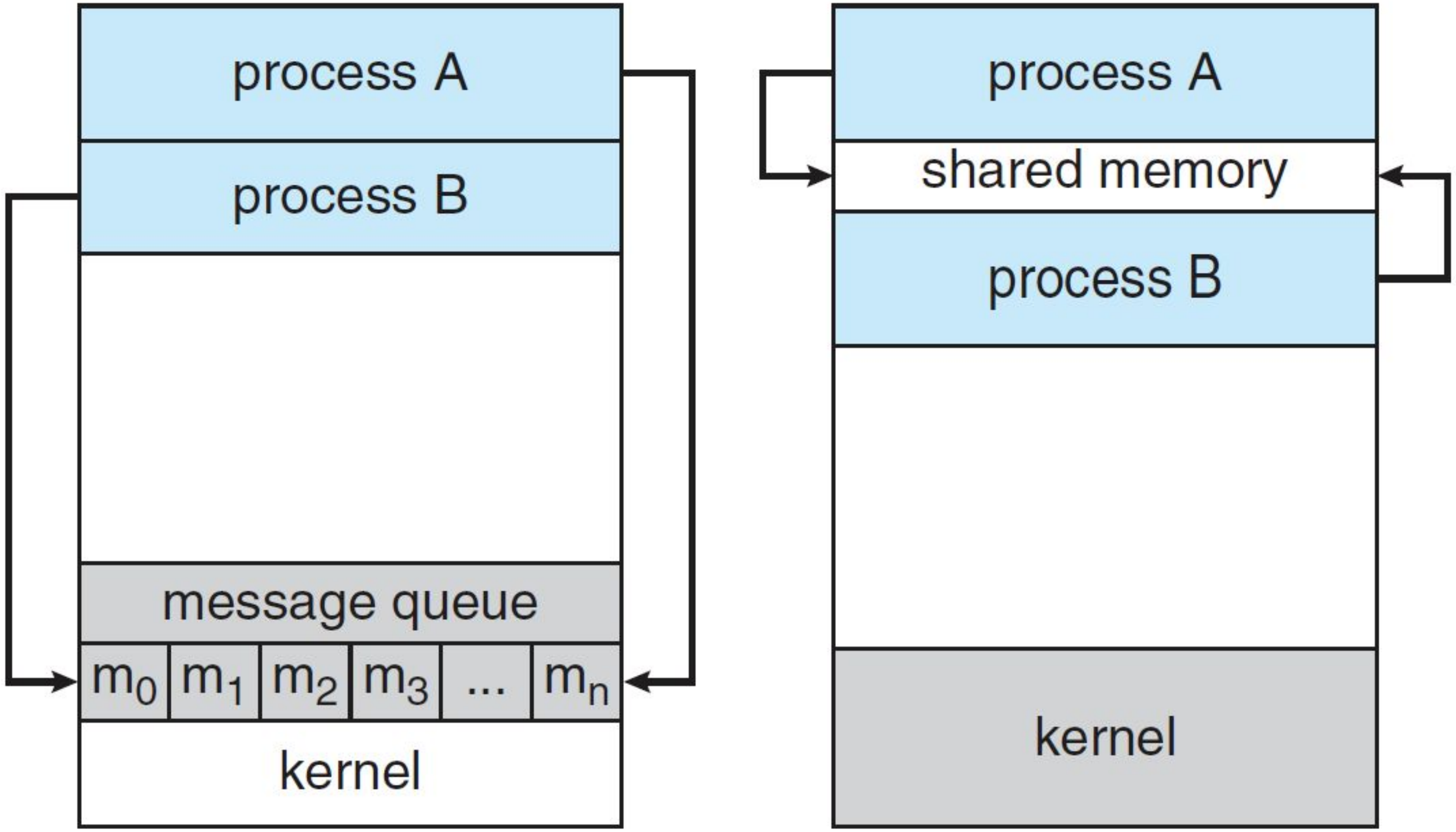
- Orphan process
  - Process whose parent has exited
  - Could litter the process table if not handled
  - On UNIX and Linux orphans are adopted by the `init` process
    - `init` periodically calls `wait()`

Interprocess communication (IPC)

- **Independent processes** – cannot affect (or be affected by) other processes
  - Do not share data with other processes
- **Cooperating processes** – opposite of independent processes
  - **Information sharing**
  - **Computation speedup** – assuming multiprocessor system
  - **Modularity**
  - **Convenience** – single user running processes concurrently
  - Require Interprocess communication (IPC)

Models of Interprocess communication

- **Shared memory**
- **Message passing**



IPC: shared memory

- Communicating processes establish a shared memory region
  - Belongs to the initiator of communication
  - OS does not arbitrate access to the shared region by the communicating processes

Classic example: the producer-consumer problem

- Producer – generates information
- Consumer – consumes the information
- Metaphor for client-server systems

Producer-consumer – Shared memory

- Shared memory contents

```
#define BUFFER_SIZE 10
typedef struct {
. . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Producer

```
item next_produced;

while (true) {
   /* produce an item in next produced */
   while (((in + 1) % BUFFER_SIZE) == out)
      ; /* do nothing */

   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
}
```

Consumer

```
item next consumed;

while (true) {
   while (in == out)
      ; /* do nothing */

   next consumed = buffer[out];
   out = (out + 1) % BUFFER SIZE;
   /* consume the item in next consumed */
}
```

Message passing

- Logical communication link
  - Direct vs. indirect communication
  - Synchronous vs. asynchronous communication
  - Automatic vs. explicit buffering
- **`send(message)`**
- **`receive(message)`**
- Communicating processes have names (IDs)

Message passing implementation questions

- How are links established?
- Can a link be associated with more than two processes?
- How many links 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 bi-directional?

Message passing – direct communication

- Sender/receiver name the other process
  - **`send(P, message)`**
  - **`receive(Q, message)`**

  Symmetric addressing
- Asymmetric addressing – receiver does not name sender
  - **`send(P, message)`**
  - **`receive(id, message)`** – receive from any process
- Properties of communication link
  - link established automatically – assuming the process know each other's ID
  - link associated with exactly two processes
  - Only one link between processes
  - Can be either unidirectional or bidirectional

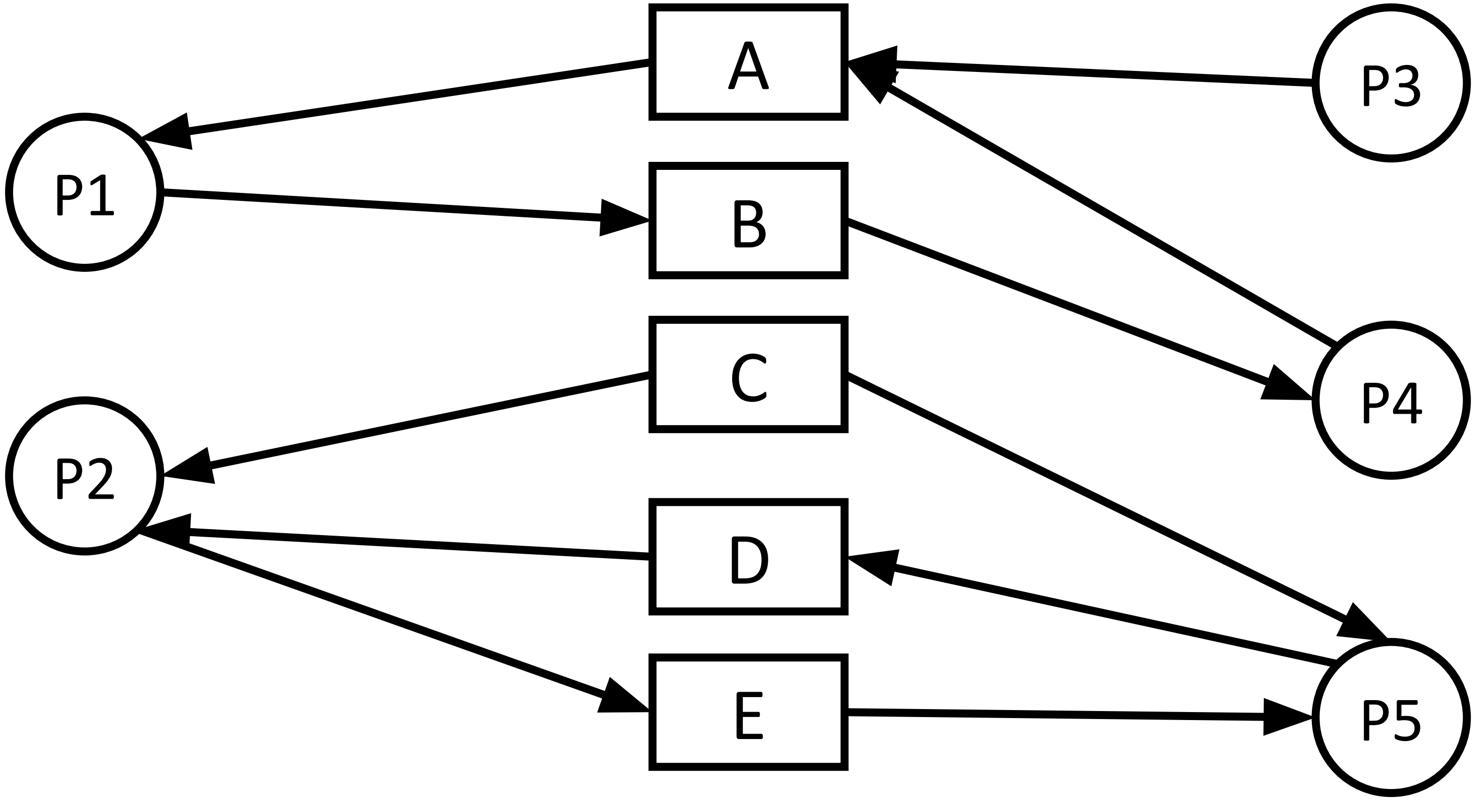
Direct Communication – disadvantage

- Communicating processes are tightly coupled
  - process IDs are hard coded in send/receive calls
  - if process ID changes, the code of all processes that communicate with it must replace references to the old ID

Message passing – Indirect communication

- Use mailboxes (also called ports)
  - **`send(A, message)`**
  - **`receive(A, message)`**
- Link properties
  - communicating processes must share a mailbox
  - link may be associated by more than two processes
  - May be multiple links between two processes – each via a separate box
  - Unidirectional/bidirectional

Indirect communication links



Message passing – indirect communication

- Operations
  - Create/destroy mailbox
  - Send/receive messages through mailbox
- Mailbox sharing
  - *P1, P2* and *P3* share mailbox A
  - P1 sends; P2 and P3 receive
  - Who gets the message?
- Solutions
  - Allow a link to associate at most two processes
  - Allow only one process at a time to execute a receive operation
  - System arbitrarily selects the receiver. Sender is notified who the receiver was

Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** operations are **synchronous**
  - **Blocking send** – sender blocks until the message is received
  - **Blocking receive** – receiver blocks until a message is available
- **Non-blocking** operations are **asynchronous**
  - **Non-blocking send** – sender sends the message and continue
  - **Non-blocking receive** – receiver receives a valid message or null
- All combinations are possible
  - blocking send + blocking receive = rendezvous

Producer-consumer rendezvous

Producer

```
message next_produced;

while (true) {/* produce an item in next produced */
    send(next_produced);
}
```

Consumer

```
message next_consumed;

while (true) {
    receive(next consumed);
    /* consume the item in next consumed */
}
```

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 (*n* messages)
    - Sender must wait if link full
  - Unbounded capacity – infinite length
    - Sender never waits