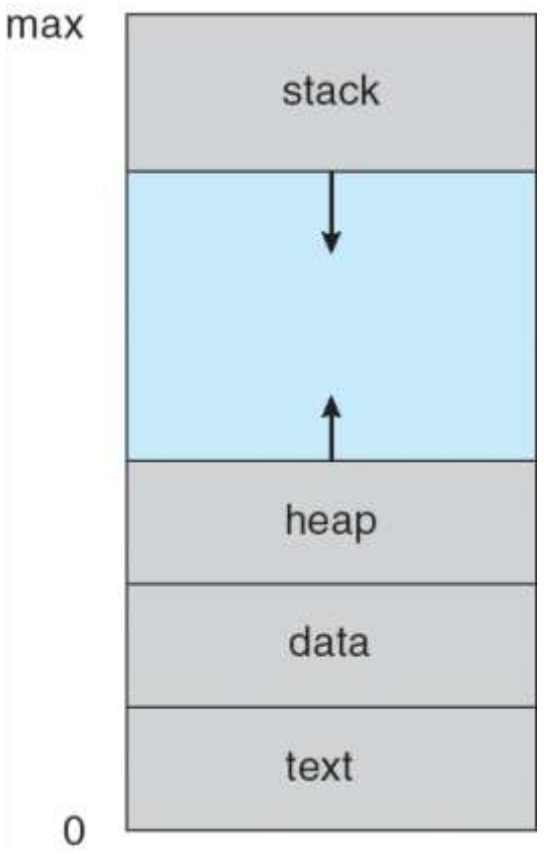
Processes

1. **Process Concept**
2. Process Scheduling
3. Operations on Processes. Interprocess communication and synchronization

Last modification date: 09.10.2016

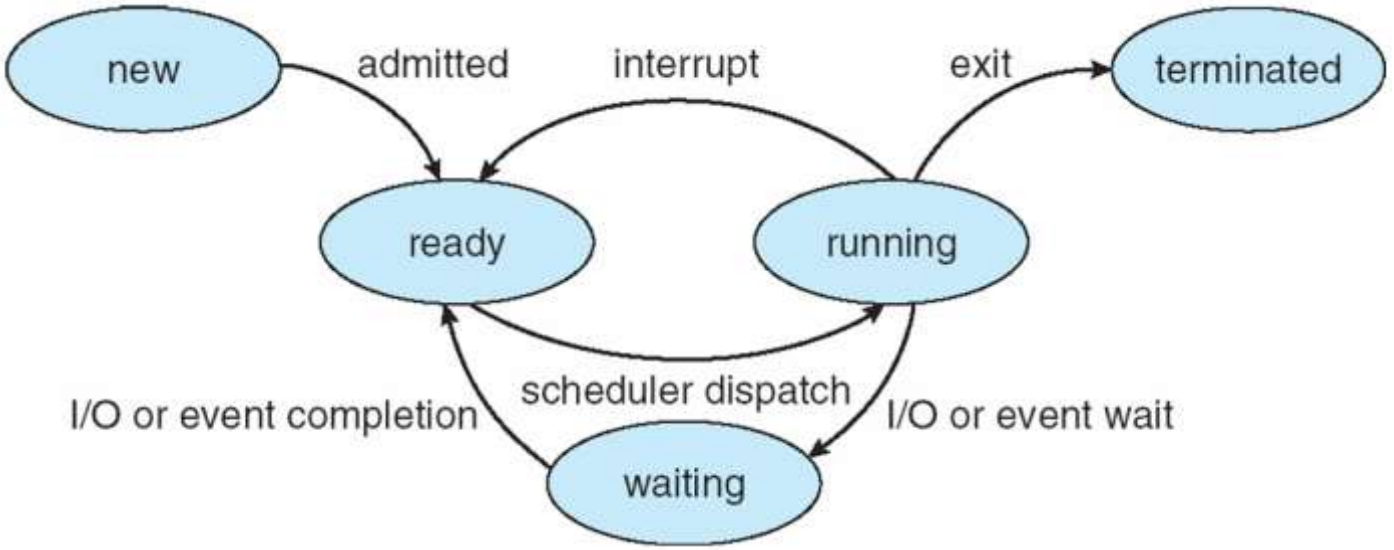
Process Concept

- 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 sequential fashion (one exec. thread)
- Multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global & static variables
  - **Heap** containing memory dynamically allocated during run time



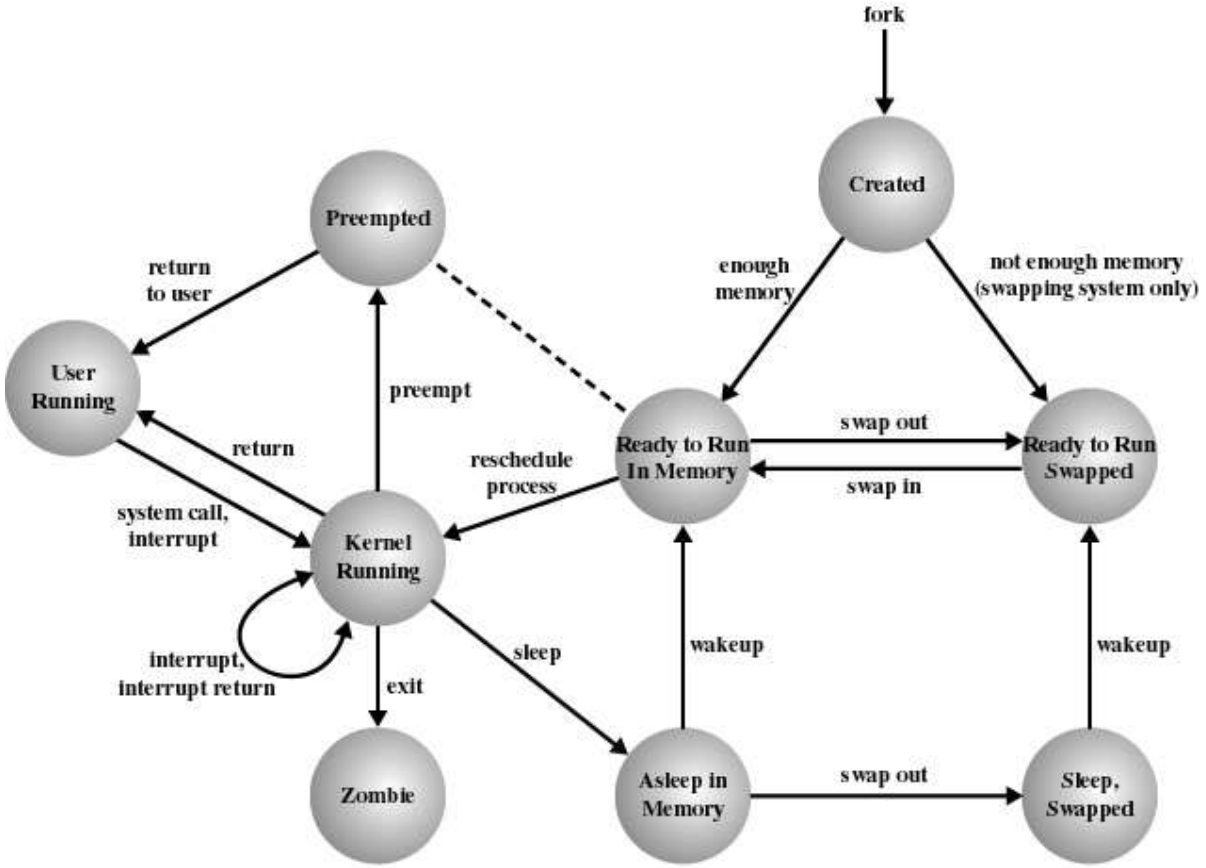
Process States

- 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



Process states – cont.

Diagram of process state for a system with virtual memory



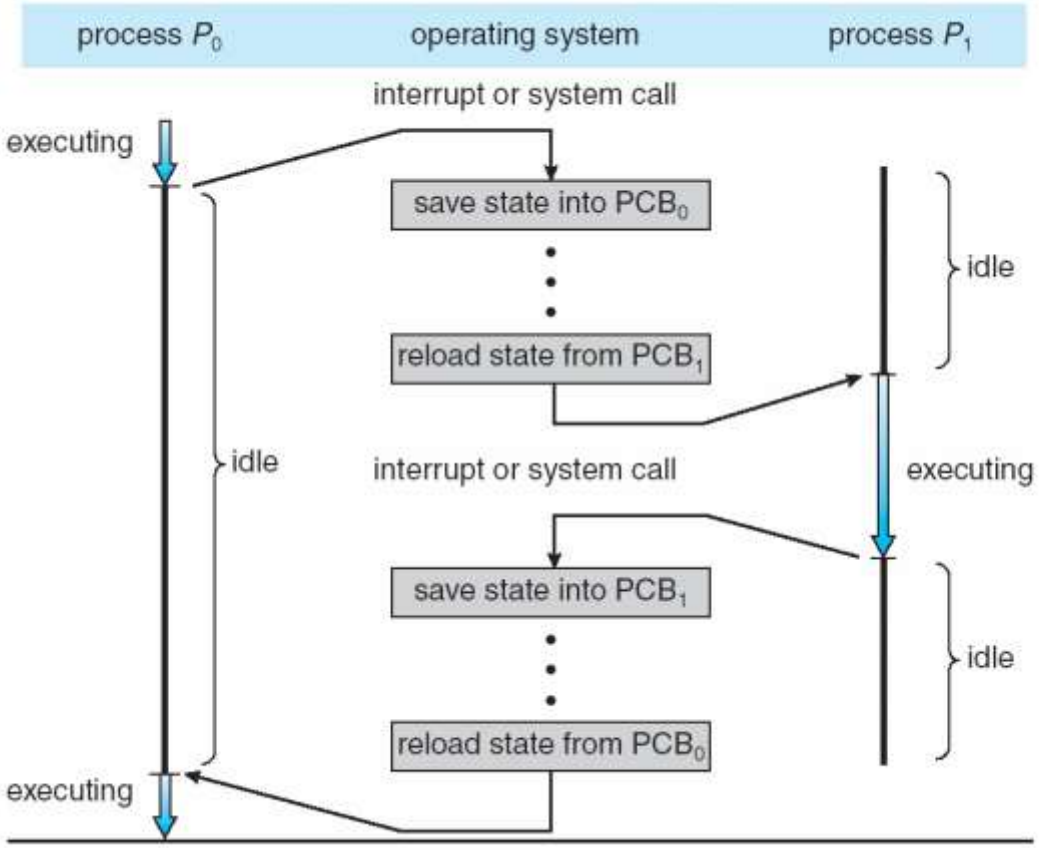
Process Control Block (PCB)

Information associated with each process

(also called **task control block**)

- Process state – running, waiting, etc..
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files

CPU Switch From Process to Process



Reasons for process interrupt:

- Timer interrupt
- Device interrupt
- System function call.
- Triggering a trap

The above events can cause CPU switch from process to process; decision is made by **CPU scheduler** .

Context Switch

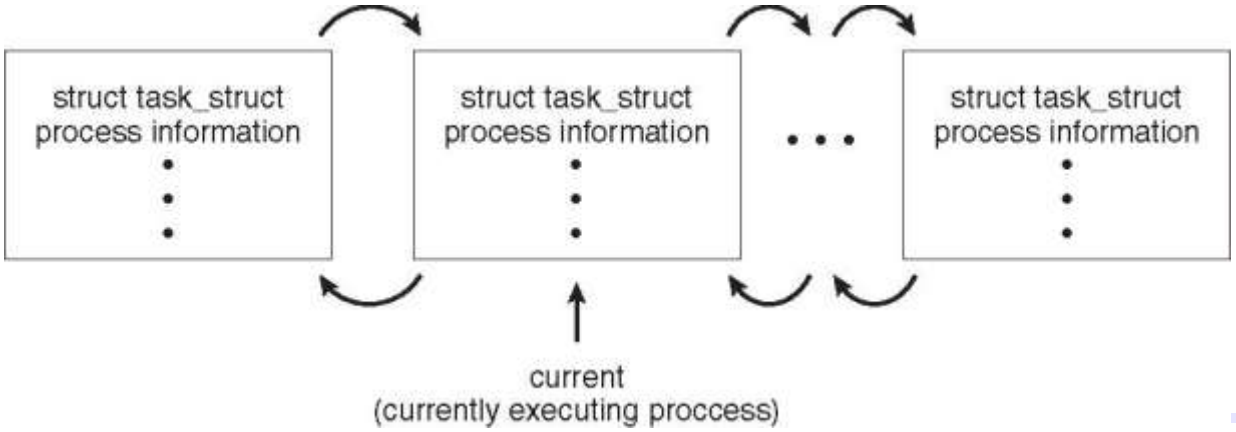
- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process is represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
  - The more complex the OS and the PCB ➔ the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU ➔ multiple contexts loaded at once

Process Representation in Linux

Represented by the C structure `task_struct` (<linux/include/linux/sched.h>)

Selected fields

```
pid t_pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```



state values

```
#define TASK_RUNNING              0
#define TASK_INTERRUPTIBLE        1
#define TASK_UNINTERRUPTIBLE      2
#define  TASK_STOPPED             4
#define  TASK_TRACED              8
/* in tsk->exit_state */
#define EXIT_DEAD                 16
#define EXIT_ZOMBIE               32
#define EXIT_TRACE (EXIT_ZOMBIE|EXIT_DEAD)
/* in tsk->state again */
#define TASK_DEAD 64
#define TASK_WAITKILL             128
#define TASK_WAKING               256
#define  TASK_PARKED              512
#define TASK_STATE_MAX            1024
```

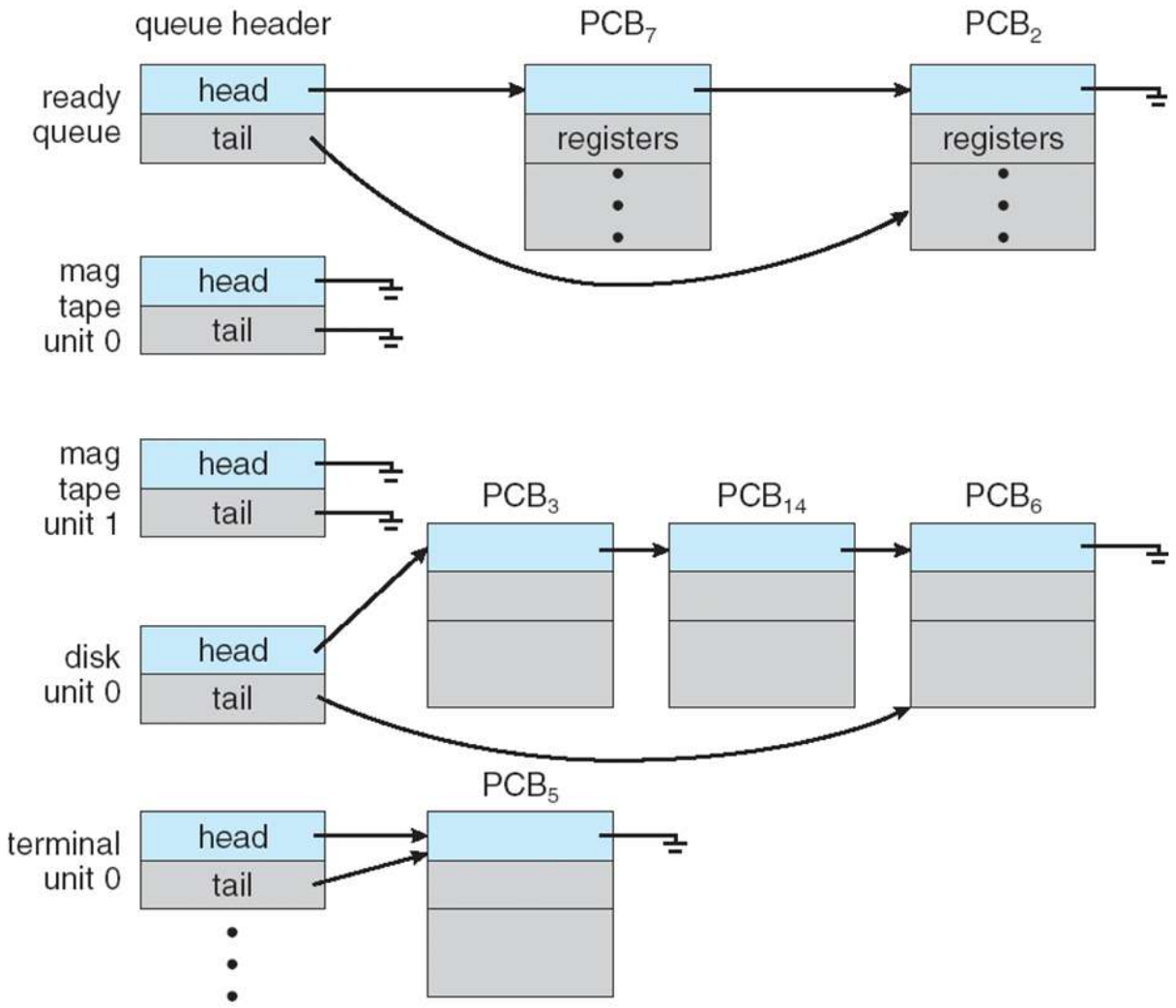
Processes

1. Process Concept
2. **Process Scheduling**
3. Operations on Processes. Interprocess communication and synchronization

Process Scheduling

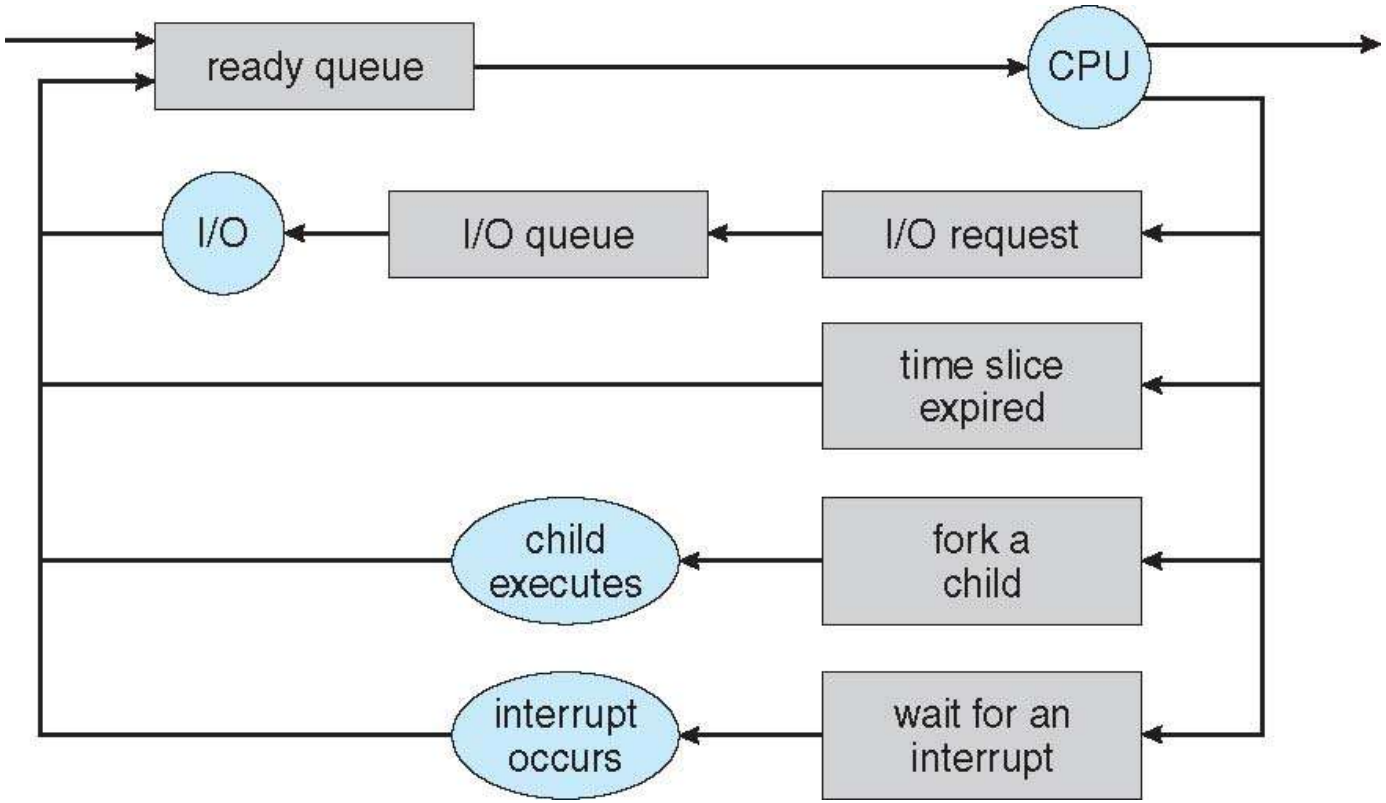
- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – set of processes waiting for an I/O device
  - Processes migrate among the various queues

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling

- **Queueing diagram** represents queues, resources, flows

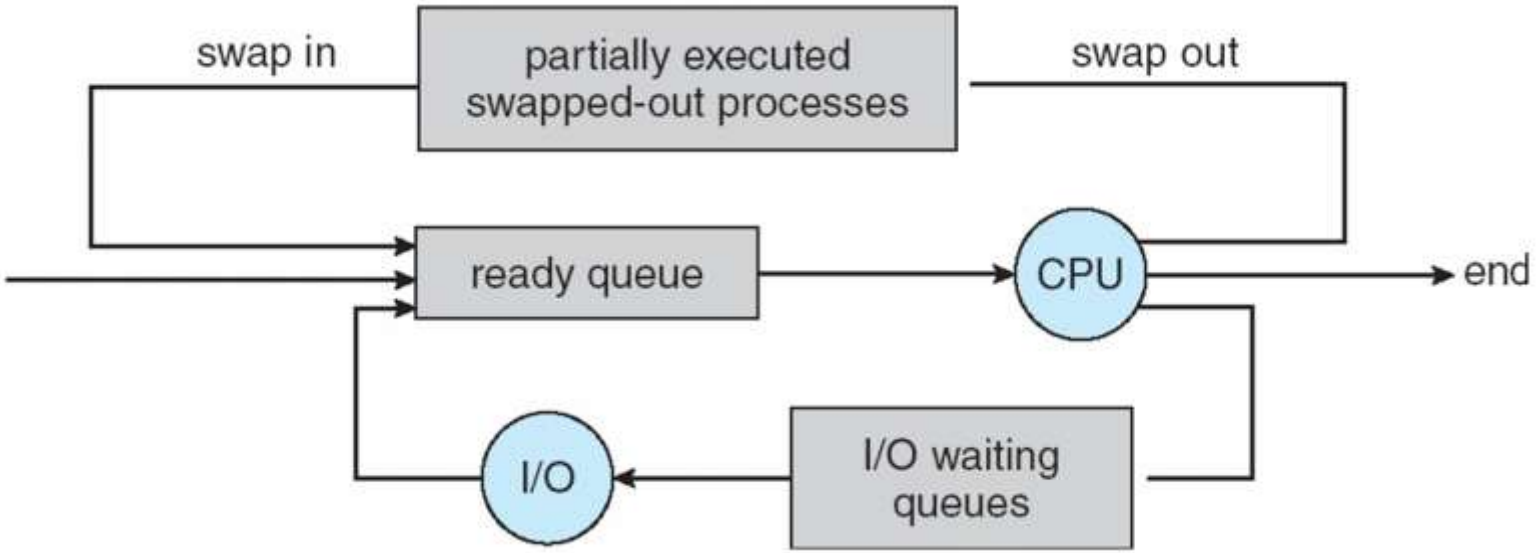


Schedulers

- A process 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
- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming**
  - Strives for good ***process mix***

Addition of Medium Term Scheduling

In some systems **medium term scheduler** is used to protect the system from too large degree of multiprogramming, by removing a process from memory, storing it on disk and bringing it back later from disk to memory to continue execution (**swapping in/out**)



Processes

1. Process Concept
2. Process Scheduling
3. **Operations on Processes. Interprocess communication and synchronization**

Operations on Processes

System must provide mechanisms for:

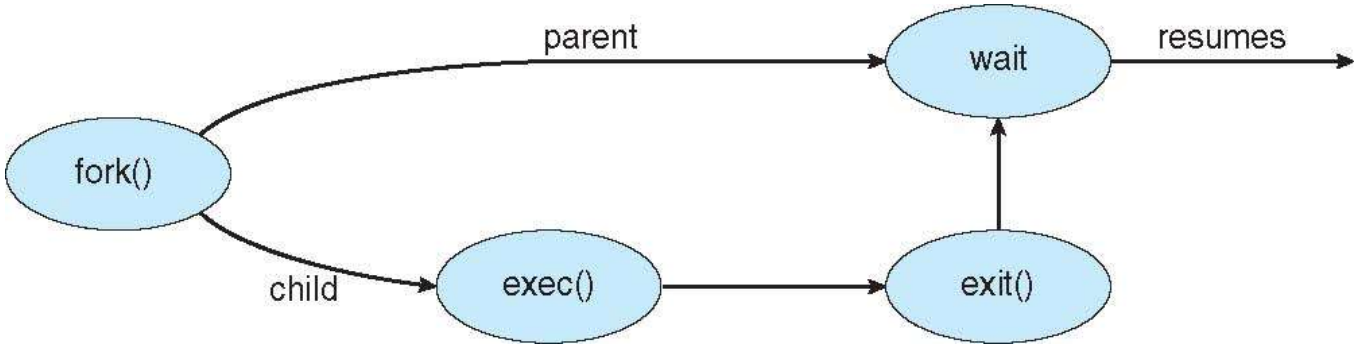
- process creation,
- process termination,
- communication and synchronization

Process Creation

- **Parent** process create **children** processes, which in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Creation (Cont.)

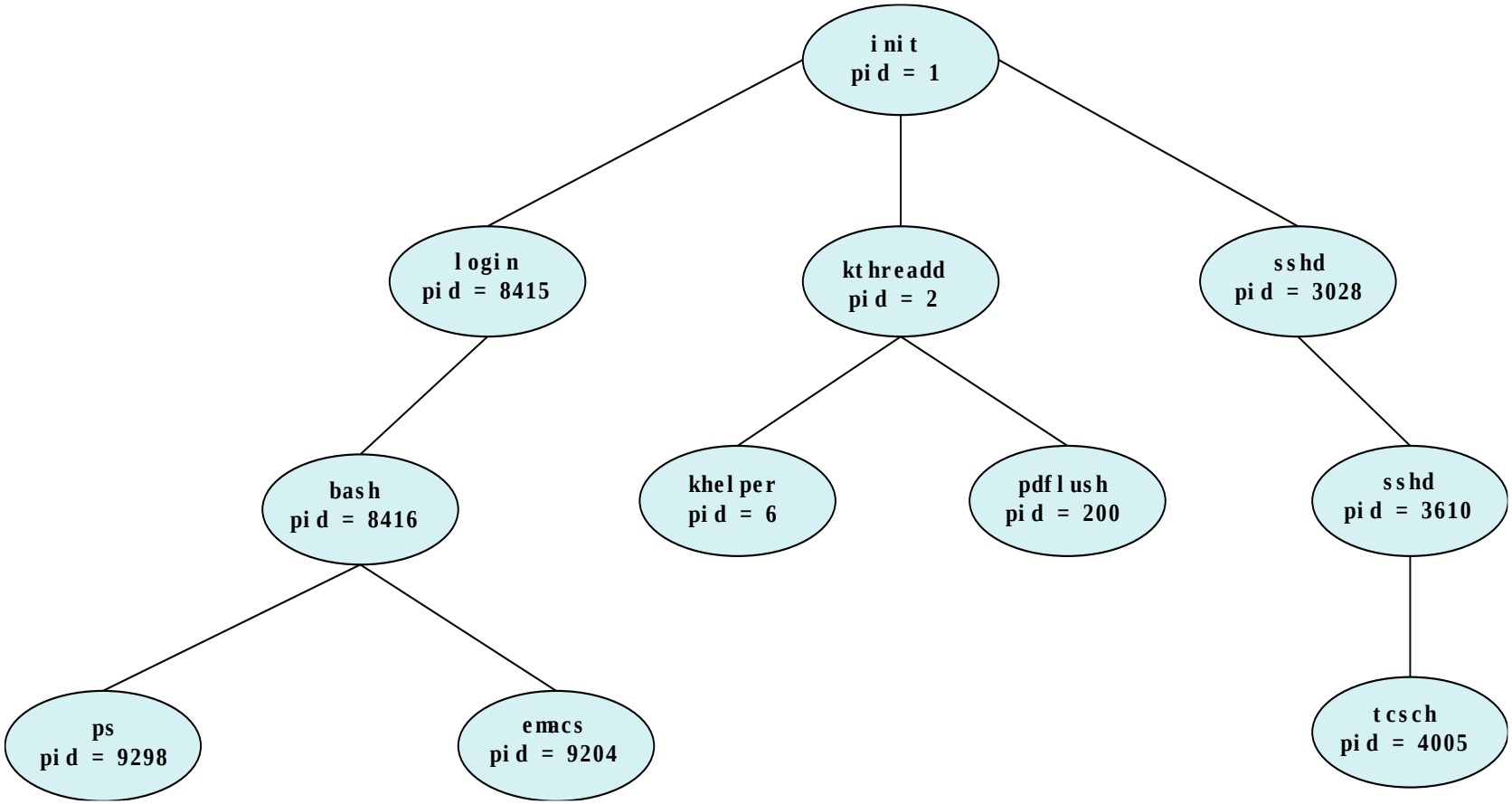
- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **`fork()`** system call creates new process
  - **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program



POSIX proces creation

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(){
pid_t  pid1, pid2;
int status;
  pid1 = fork(); /* fork a child process*/
  if (pid1 < 0) { /* error occurred */
        perror("Fork failed");    exit(1);
  } else if (pid1 == 0) { /* child process */
        execl("/bin/ls", "ls", " -l",NULL); /* overwriting the child proces
                                              memory with new program */
        perror("execlp");  exit(2);/* executed only if execl fails */
  } else { /* parent process */
        if((pid2=wait (&status))>0){/* waiting for the child proces to complete */
              fprintf (stderr,„Potomek: PID=%d, status=%d,(int)pid2,status);
        } else {
                perror("wait");  exit(3); /* wait() error */
        }
        return(0); /* alternatively: exit(0) */
  }
}
```

A Tree of Processes in Linux



Note: in contemporary systems **systemd** process can be run with pid=1, instead of **init**

Creating a Separate Process via Windows API

```
#include <windows.h>
#include <stdio.h>
int main( VOID ){
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    ZeroMemory( &si, sizeof(si) ); si.cb = sizeof(si);ZeroMemory( &pi, sizeof(pi) );
//  Start the child process.
    if( !CreateProcess( NULL,   // No module name (use command line).
        "C:\\WINDOWS\\system32\\notepad.exe", // Command line.
        NULL,             // Process handle not inheritable.
        NULL,             // Thread handle not inheritable.
        FALSE,            // Set handle inheritance to FALSE.
        0,                // No creation flags.
        NULL,             // Use parent's environment block.
        NULL,             // Use parent's starting directory.
        &si,              // Pointer to STARTUPINFO structure.
        &pi )             // Pointer to PROCESS_INFORMATION structure.
    ) {        fprintf(stderr, "CreateProcess failed (%d).\n", GetLastError() );
  return -1;    }
    WaitForSingleObject( pi.hProcess, INFINITE ); // Wait until child process exits.
    CloseHandle( pi.hProcess );    CloseHandle( pi.hThread ); // Close process and thread
  handles.
}
```

Execution of a system command - standard C library

```
#include <stdlib.h>
#include <stdio.h>
int main( void){
    char buf[512];
    if(fgets(buf, sizeof(buf), stdin)){/* reading cmd */
       int ret;
/* execution of a command in a system shell subprocess */
       ret=system(buf);
       if(ret) fprintf(stderr,"ret=%d\n",ret);
       return ret;
    }
   return 0;
}
```

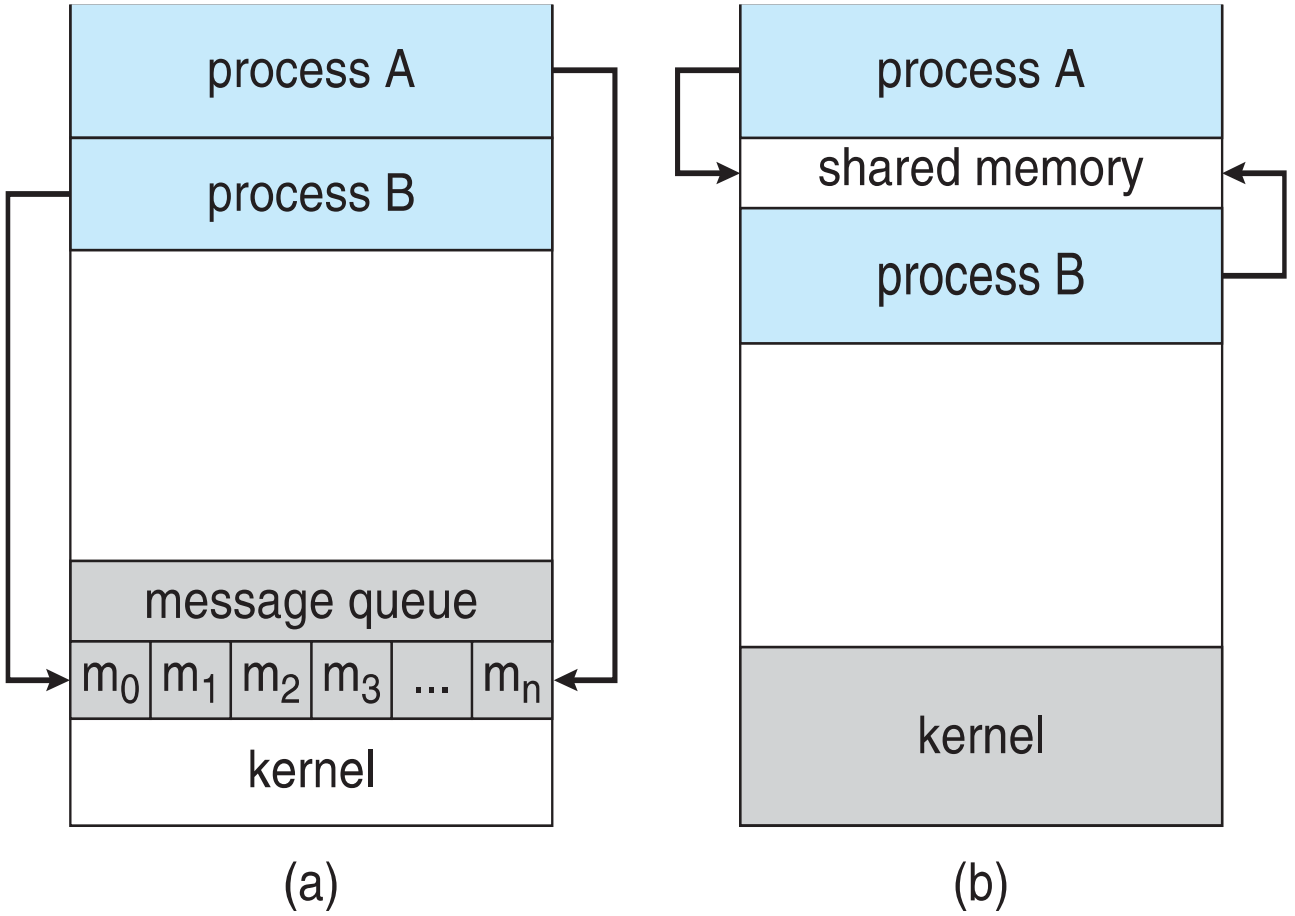
Process Termination

- Process can ask the operating system to delete it using a system call (POSIX/C: **`exit()`**).
  - Returns status data from child to parent (POSIX: via **`wait()`**). Also In POSIX:
    - if no parent waiting (did not invoke **`wait()`**) process is a **zombie**
    - If parent terminated without invoking **`wait`**, process is an **orphan**
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using a system call (e.g. **`TerminateProcess()`** in Win32). Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates
- Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - **cascading termination.** All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system

Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Communications Models



(a) Message passing. (b) shared memory. Correct use of R/W operations requires synchronization (**critical section**)