Chapter 3: Processes

Process Concept

- An operating system executes a variety of programs that run as a process.

- **Process** – a program in execution; process execution must progress in sequential fashion. No parallel execution of instructions of a single process

- 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 variables

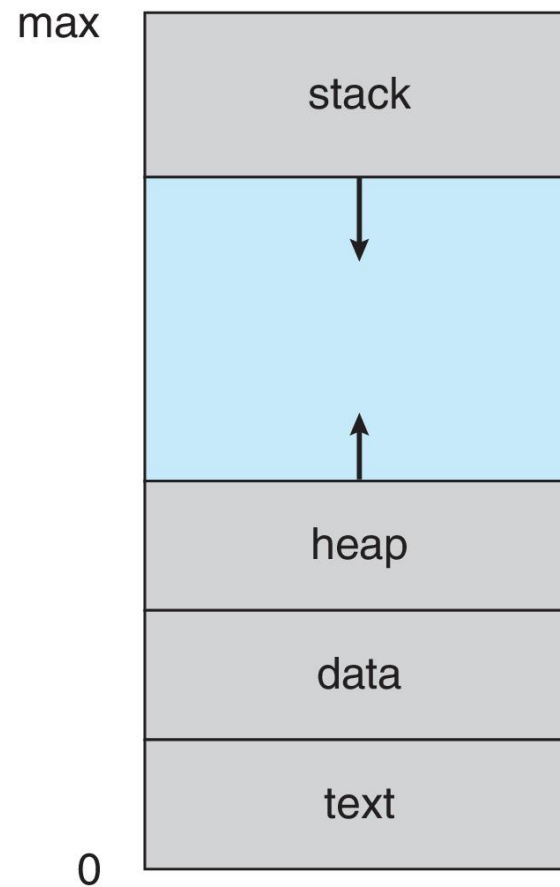 - **Heap** containing memory dynamically allocated during run time

Process Concept (Cont.)

- Program is **passive** entity stored on disk (**executable file**); process is **active**

  - Program becomes process when an executable file is loaded into memory

- Execution of program started via GUI mouse clicks, command line entry of its name, etc.

- One program can be several processes

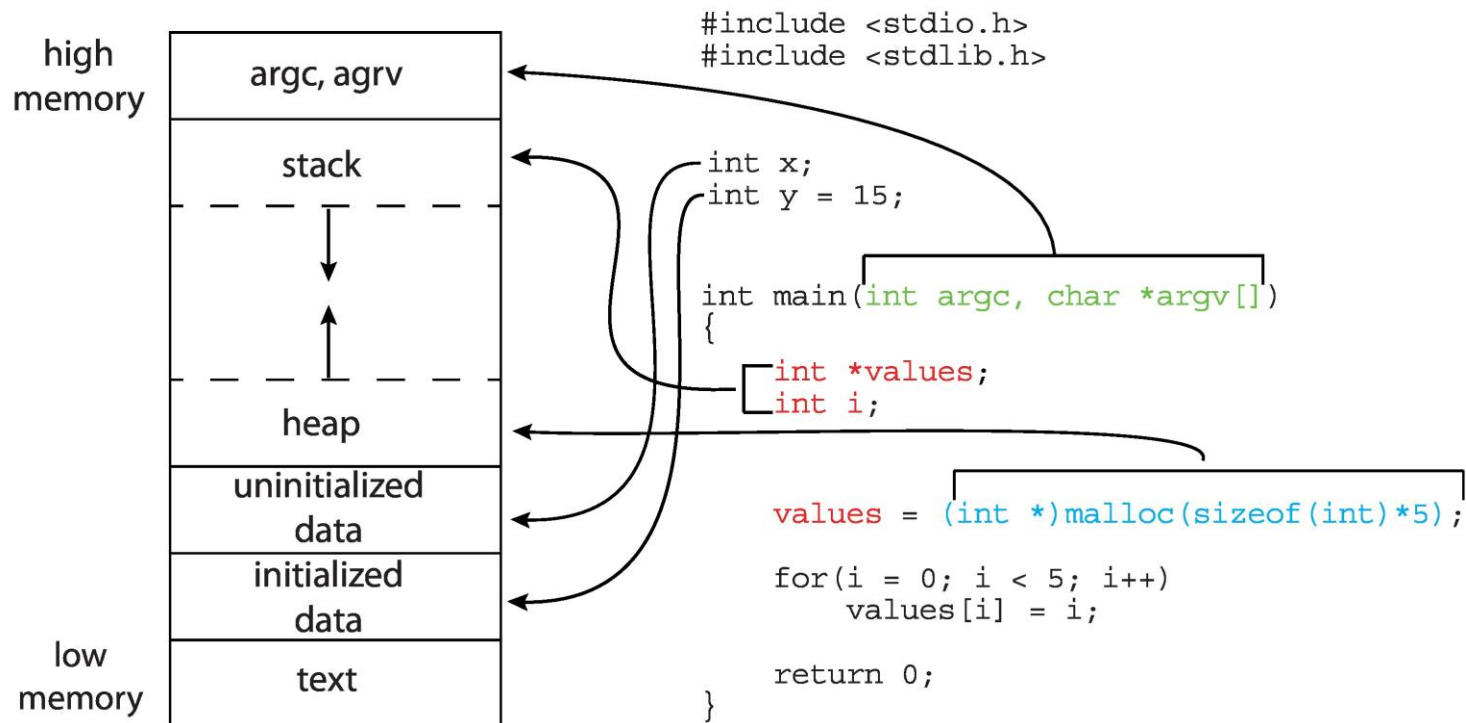 - Consider multiple users executing the same program

Process in Memory

Memory Layout of a C Program


```
high          argc, agrv
memory

              stack

                |
                v
                ^
                |

              heap

           uninitialized
               data
            initialized
                data
low
memory        text
```

```c
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 15;


int main(int argc, char *argv[])
{
    int *values;
    int i;


    values = (int *)malloc(sizeof(int)*5);

    for(i = 0; i < 5; i++)
        values[i] = i;

    return 0;
}
```
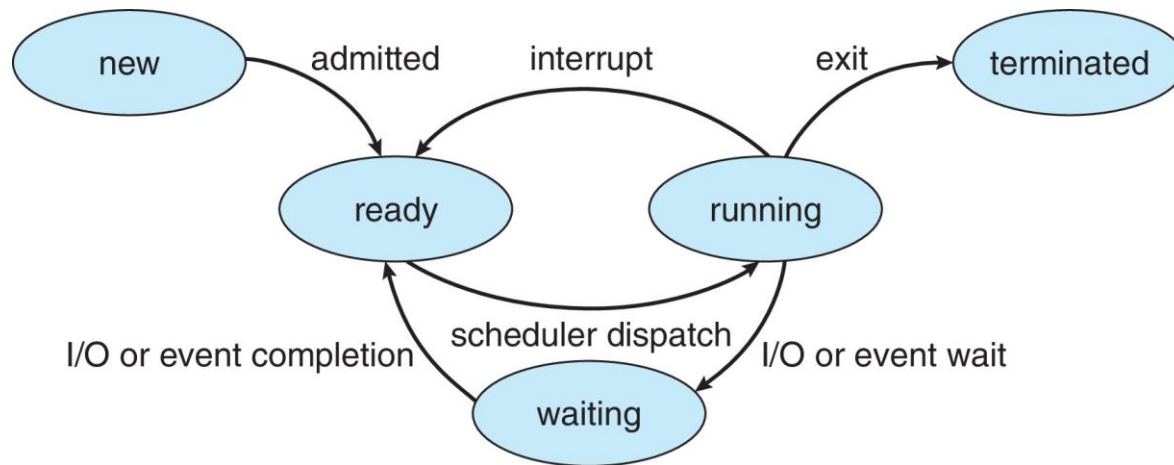
Process State

- 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(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

| process state |
|---|
| process number |
| program counter |
| registers |
| memory limits |
| list of open files |
| • • • |

Process Representation in Linux

Represented by the C structure `task_struct`

```
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 */
```
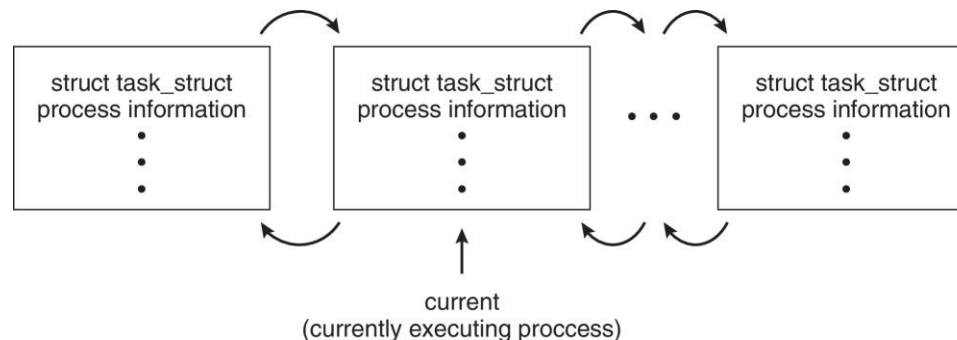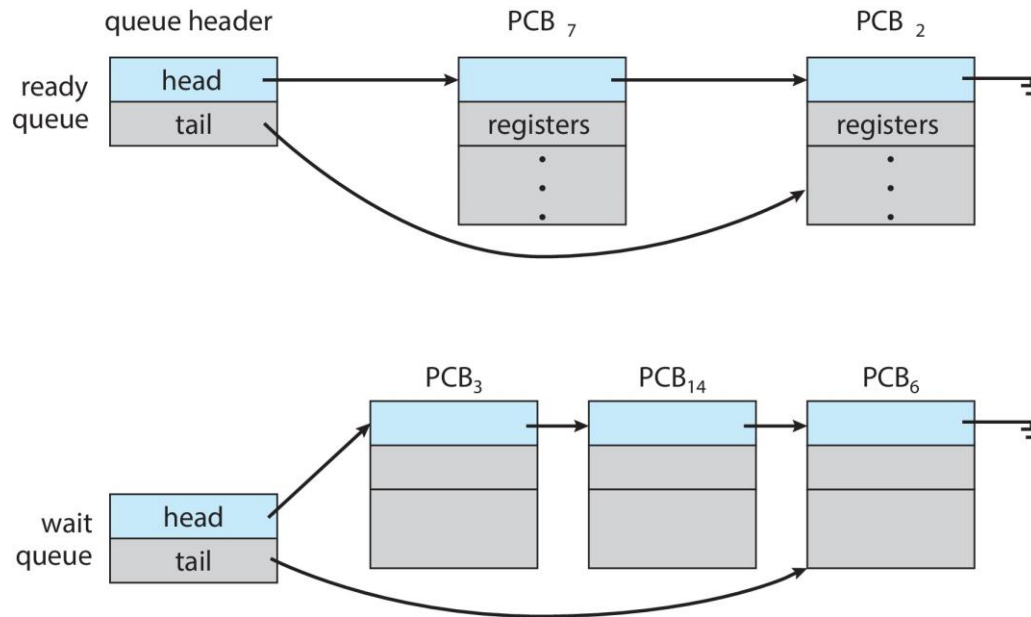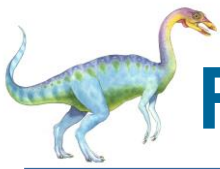
Process Scheduling

- **Process scheduler** selects among available processes for next execution on CPU core

- Goal -- Maximize CPU use, quickly switch processes onto CPU core

- Maintains **scheduling queues** of processes

  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute

  - **Wait queues** – set of processes waiting for an event (i.e., I/O)

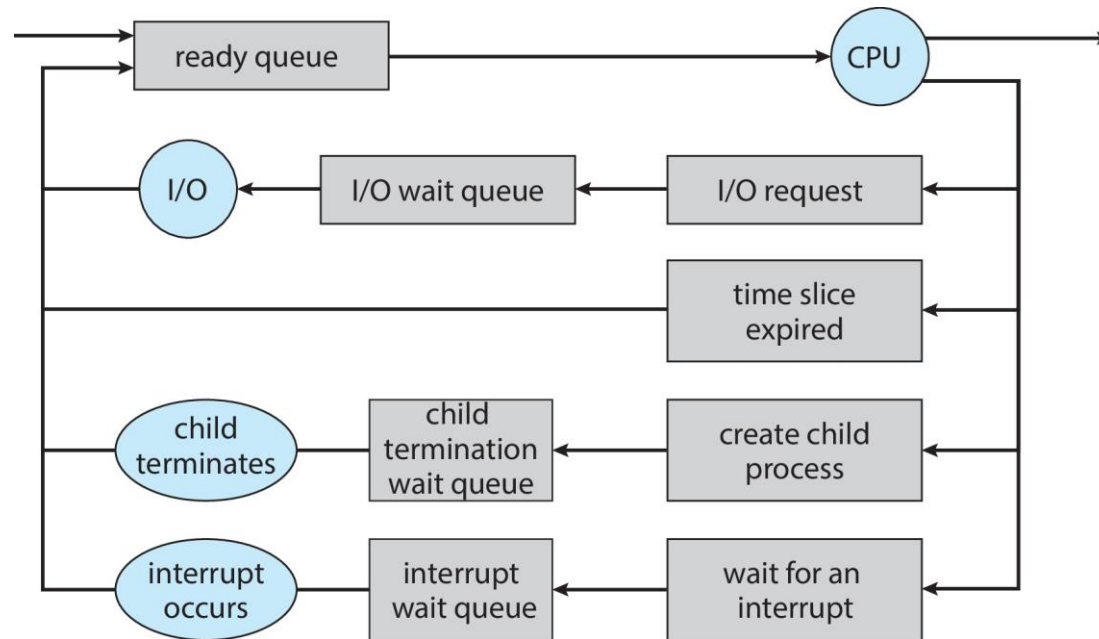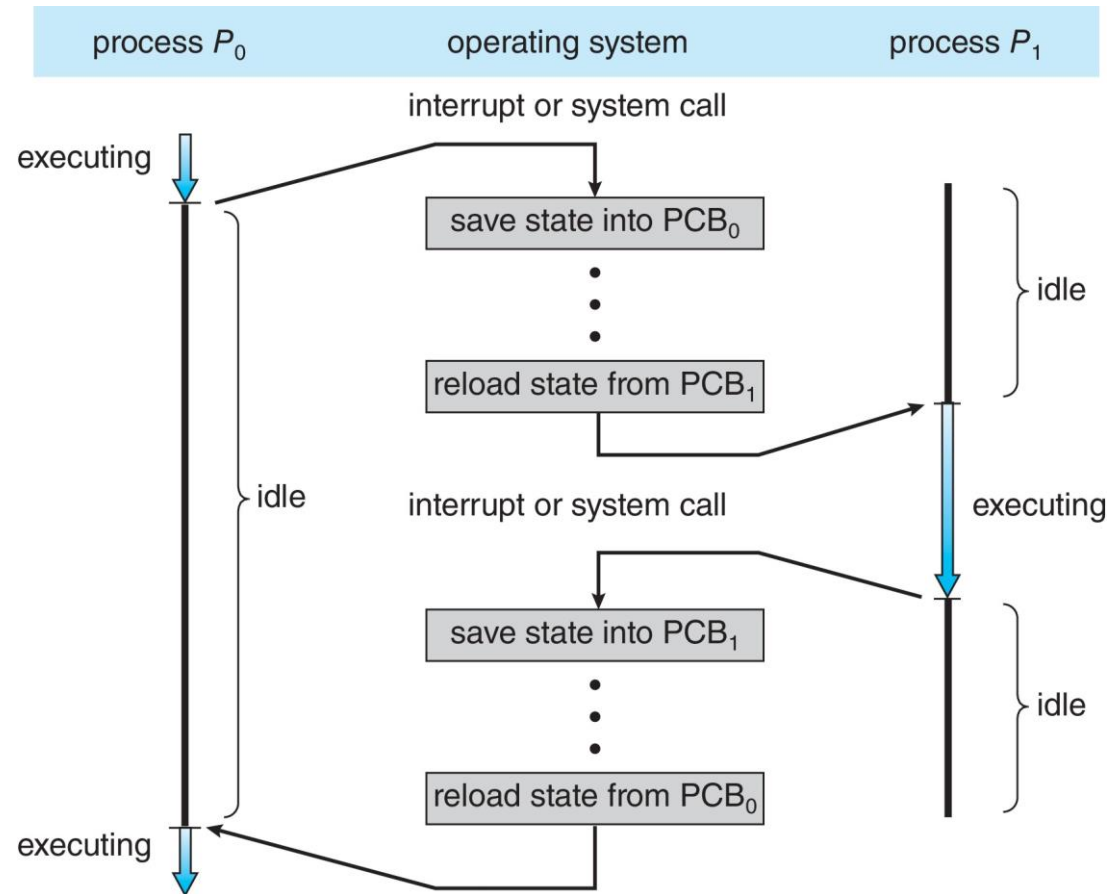 - Processes migrate among the various queues

Representation of Process Scheduling

CPU Switch From Process to Process

A **context switch** occurs when the CPU switches from one process to another.

process P_0	operating system	process P_1

interrupt or system call

executing

save state into PCB_0

⋮

reload state from PCB_1

idle

idle

interrupt or system call

executing

save state into PCB_1

⋮

reload state from PCB_0

executing

idle

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 represented in the PCB

- Context-switch time is pure 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

Operations on Processes

- System must provide mechanisms for:
  - Process creation
 - Process termination

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
  - Parent process calls `wait()` waiting for the child to terminate
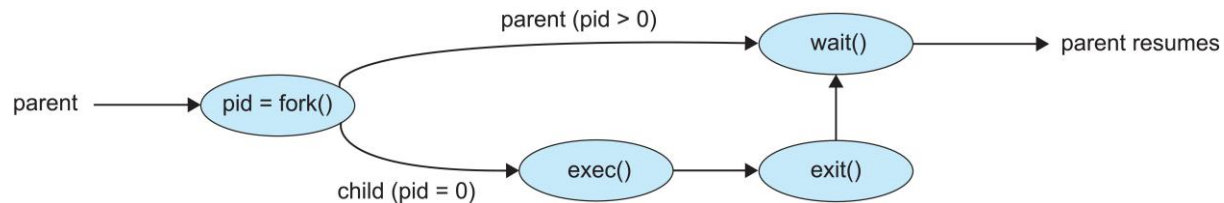
A Tree of Processes in Linux
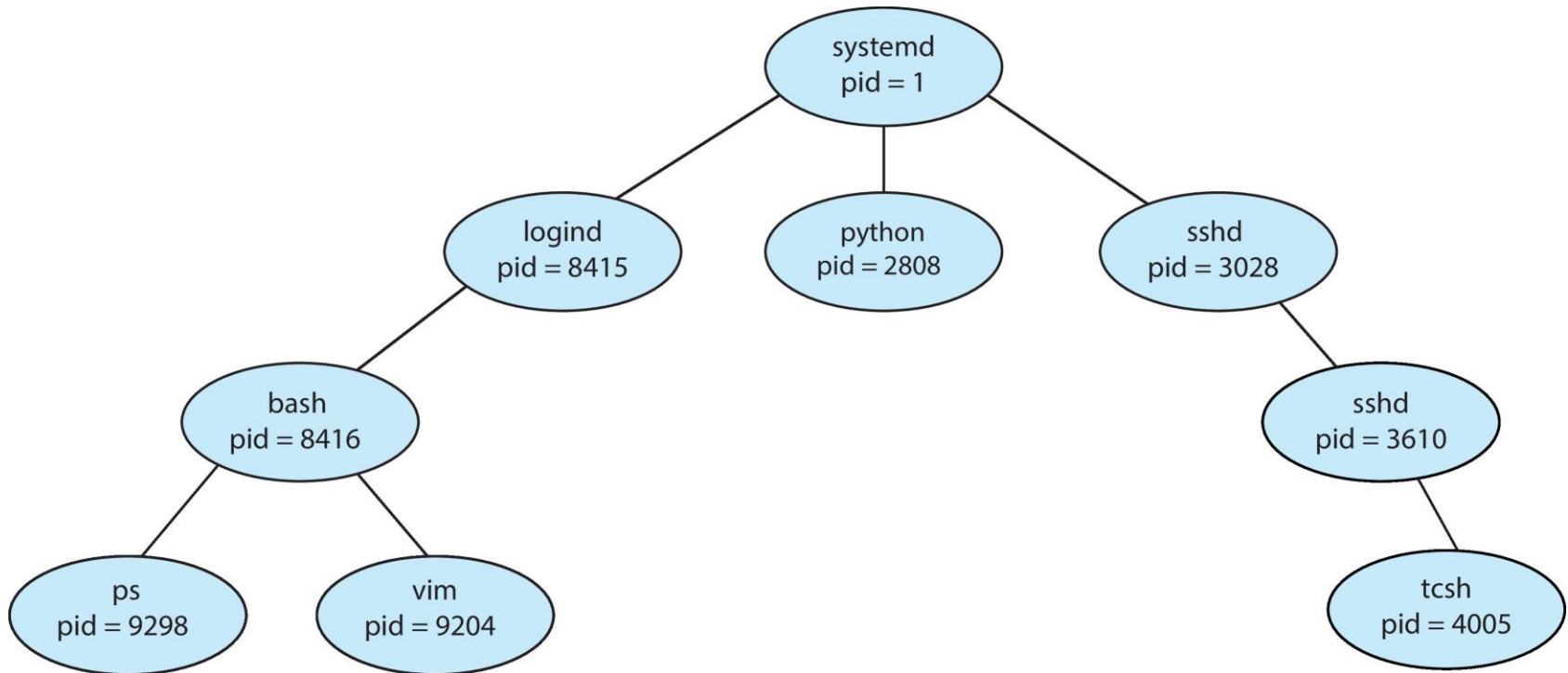
C Program Forking Separate Process

```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;
}
```

```c
#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;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Process Termination

- Process executes last statement and then asks the operating system to delete it using the **exit()** system call.
    - Returns status data from child to parent (via **wait()**)
    - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the **abort()** system call. 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

Process Termination

- 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.

- The parent process may wait for termination of a child process by using the `wait()` system call. The call returns status information and the pid of the terminated process

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

- If no parent waiting (did not invoke `wait()`) process is a **zombie**

- If parent terminated without invoking `wait()`, process is an **orphan**

Multiprocess Architecture – Chrome Browser

- Many web browsers ran as single process (some still do)

  - If one web site causes trouble, entire browser can hang or crash

- Google Chrome Browser is multiprocess with 3 different types of processes:

  - **Browser** process manages user interface, disk and network I/O

  - **Renderer** process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened

    ‣ Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits

  - **Plug-in** process for each type of plug-in
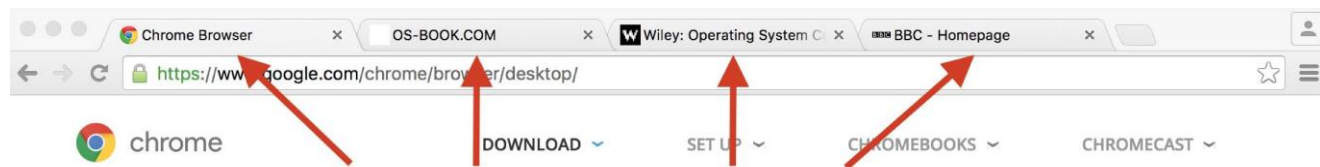
Each tab represents a separate process.
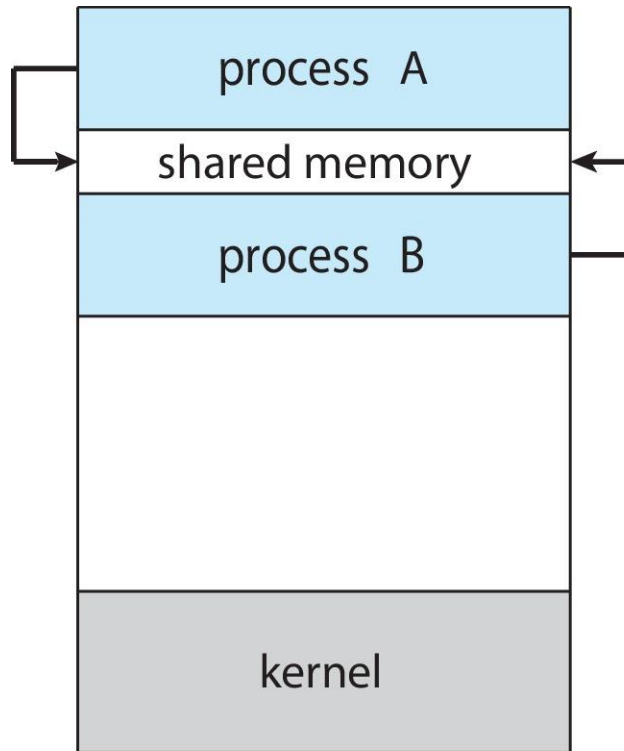
Interprocess Communication

- 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

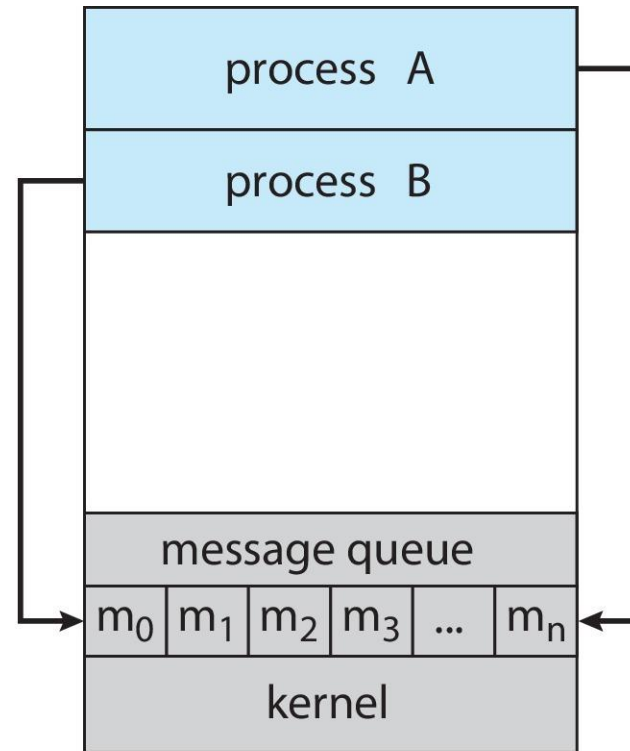 - **Shared memory**

  - **Message passing**

Communications Models

(a) Shared memory.   (b) Message passing.

(a)   (b)

End of Chapter 3