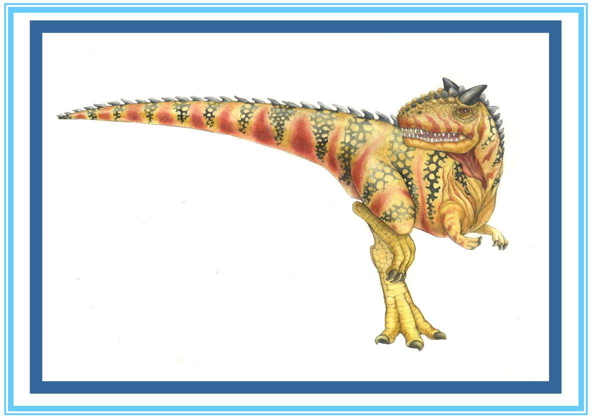
# Chapter 3: Processes



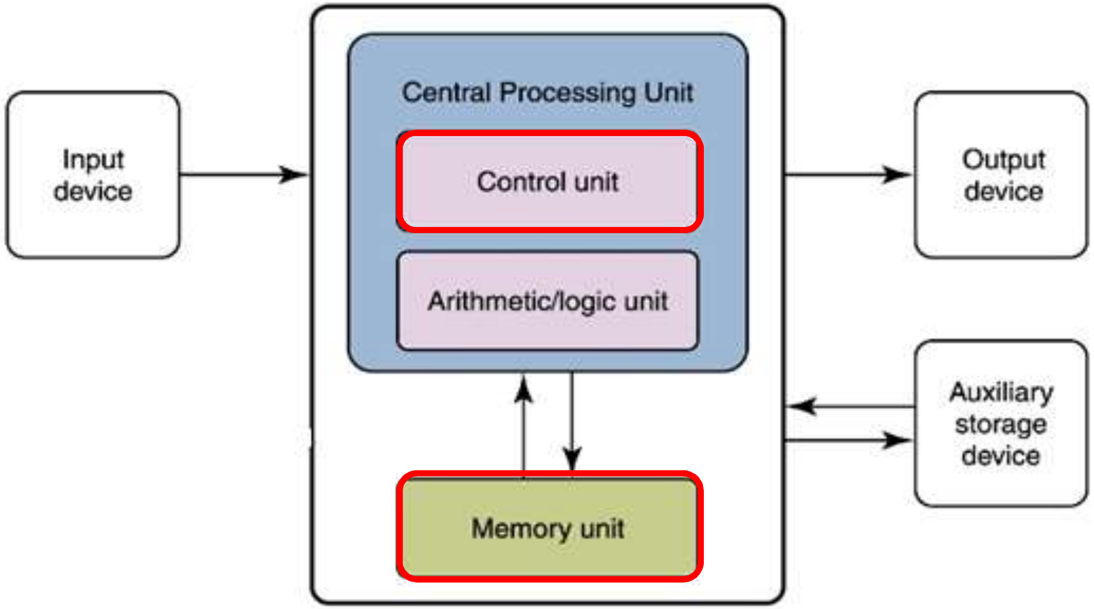
# Chapter 3:  Processes

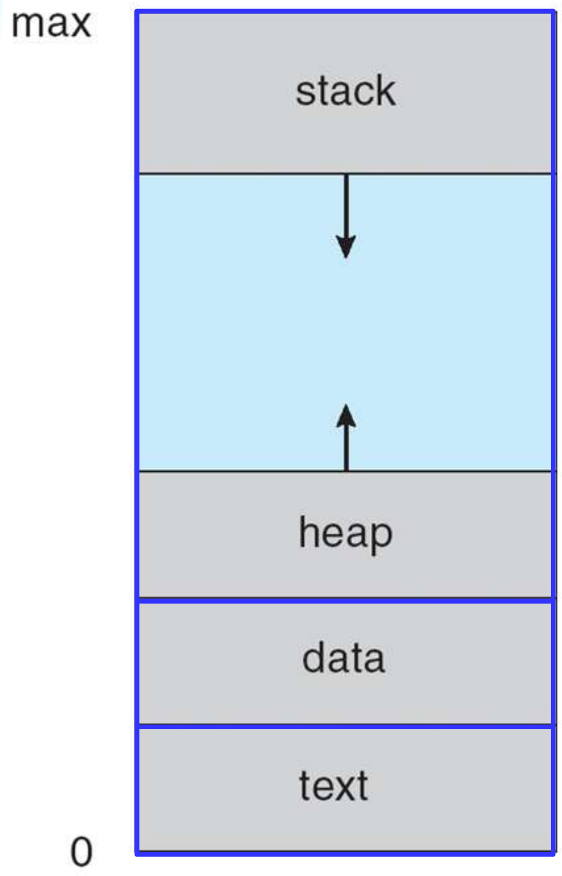
- Process Concept
- Operations on Processes
- Inter-process Communication(IPC)

# Process Concept

- **Process** – **a program in execution**; process execution must **progress in sequential fashion**
- Multiple parts
  - The program code, also called **text section**
  - **Data section** containing global variables, static local variables
  - **Stack** containing temporary data
    - Function parameters, return addresses, automatic local variables
  - **Heap** containing memory dynamically allocated during run time
  - Current activity including **program counter**, processor registers
- Program is ***passive*** entity stored on disk (executable file), **process is *active***
  - **Program becomes process when executable file 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



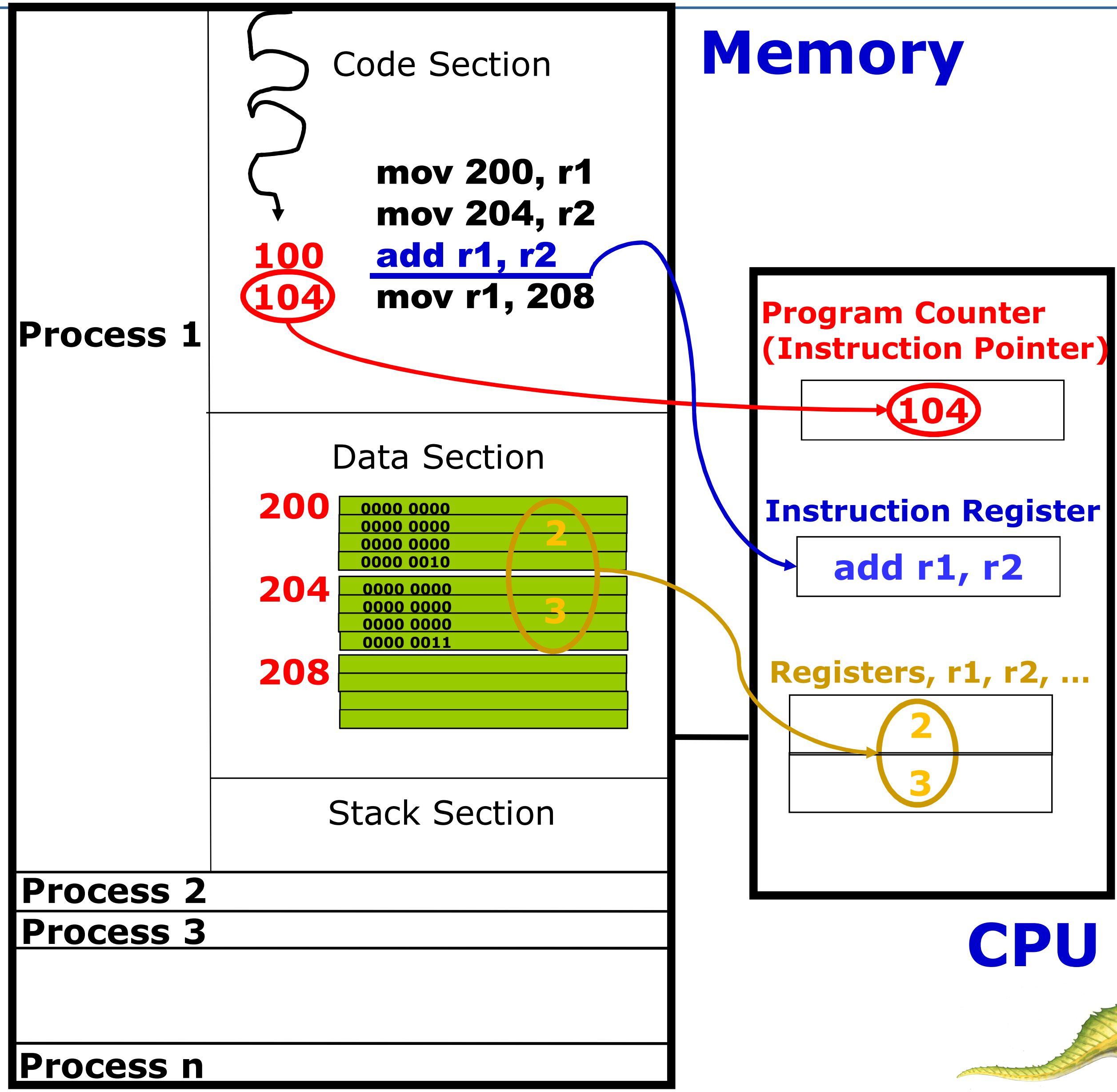
# Stored Program Concept

**High-level Language**
int i, j, k;
k = i + j;

**Assembly Language**
*i => address 200*
*j => address 204*
*k => address 208*

```
mov   200, r1
mov   204, r2
add   r1, r2
mov   r1, 208
```

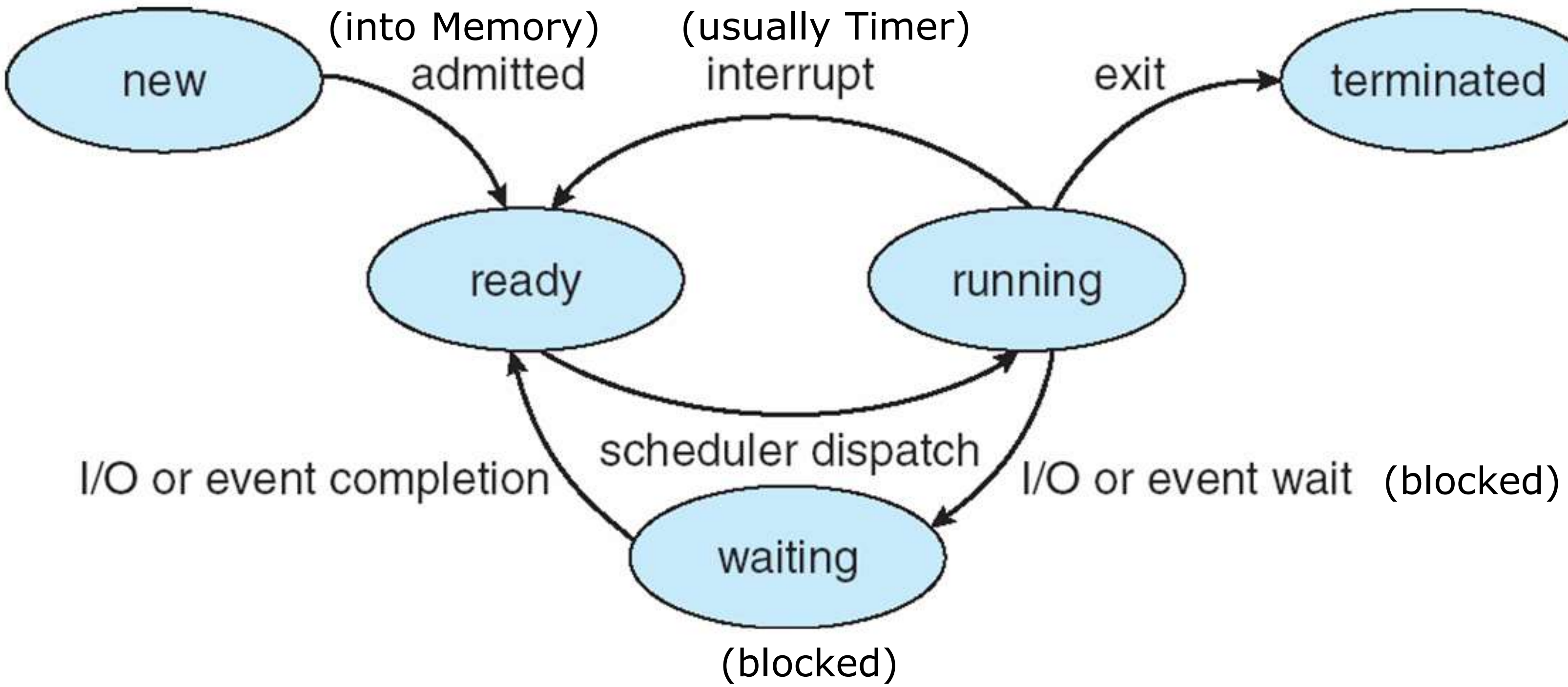
**Machine Language**
010101…..00011
010110…..01010
010101…..00011
010110…..01010



# Process State

- As a process executes, it changes **state**
  - **new**: The process is being created
  - **ready**: The process is **waiting to be assigned to a processor**
  - **running**: **Instructions are being executed**
    - The only state that hold the CPU
  - **waiting(blocked)**: The process is **waiting(blocked) for some event to occur**
    - Cannot run on CPU
  - **terminated**: The process has finished execution

# Diagram of Process State

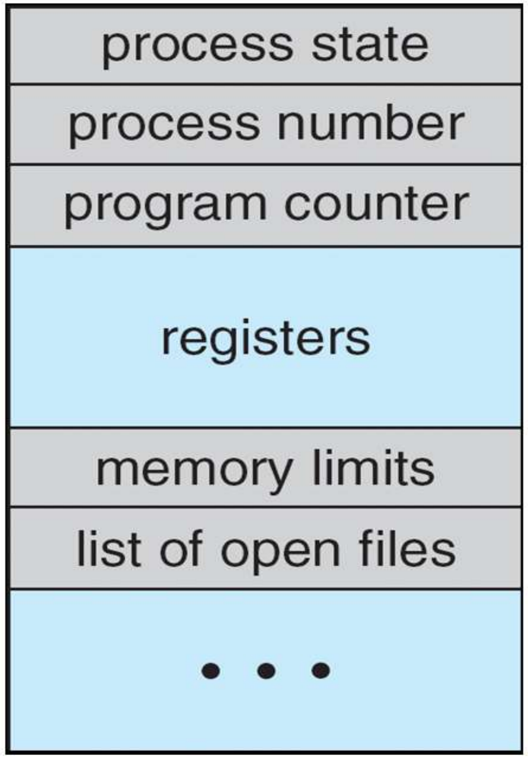


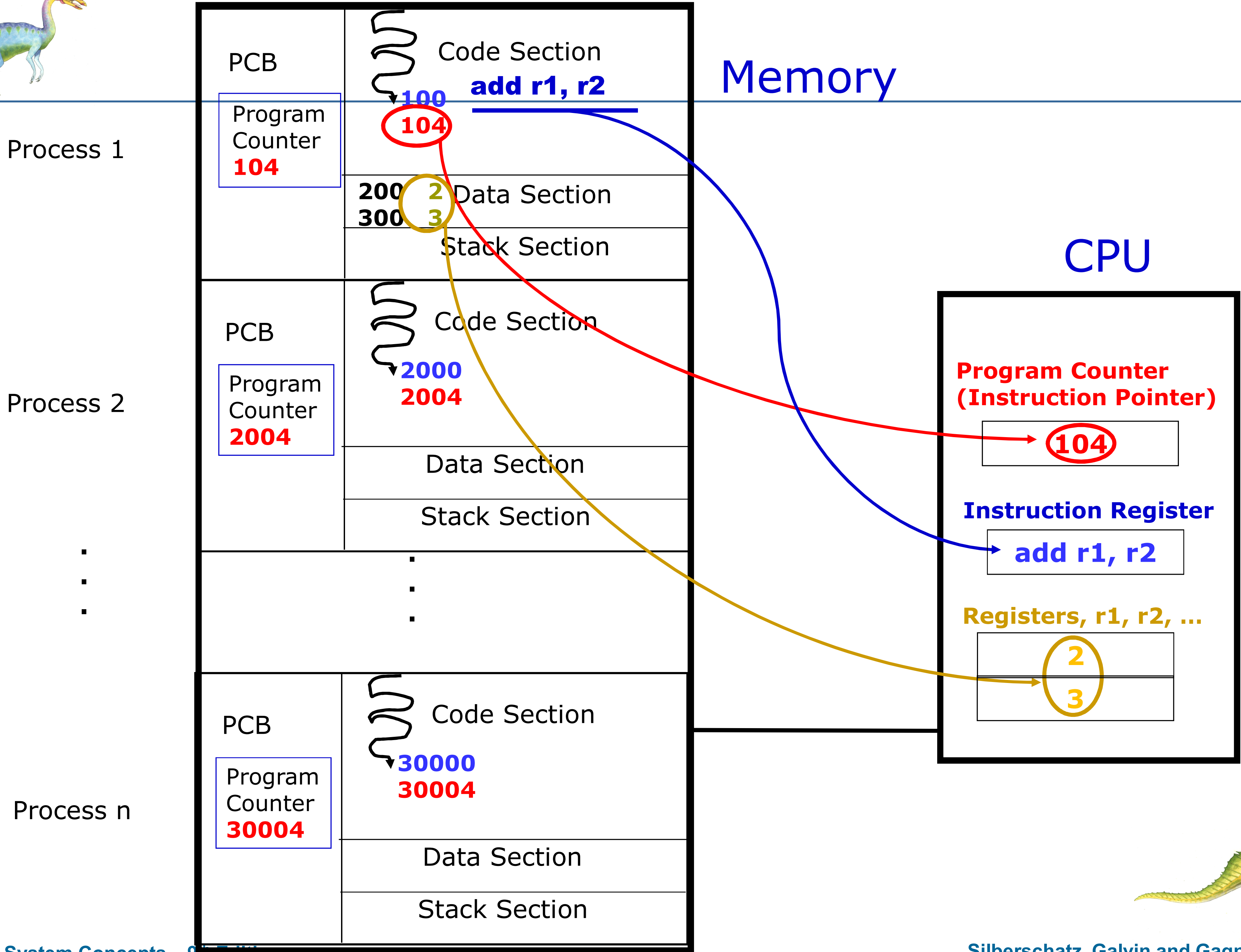
# Process Control Block (PCB)

**Information associated with each process**

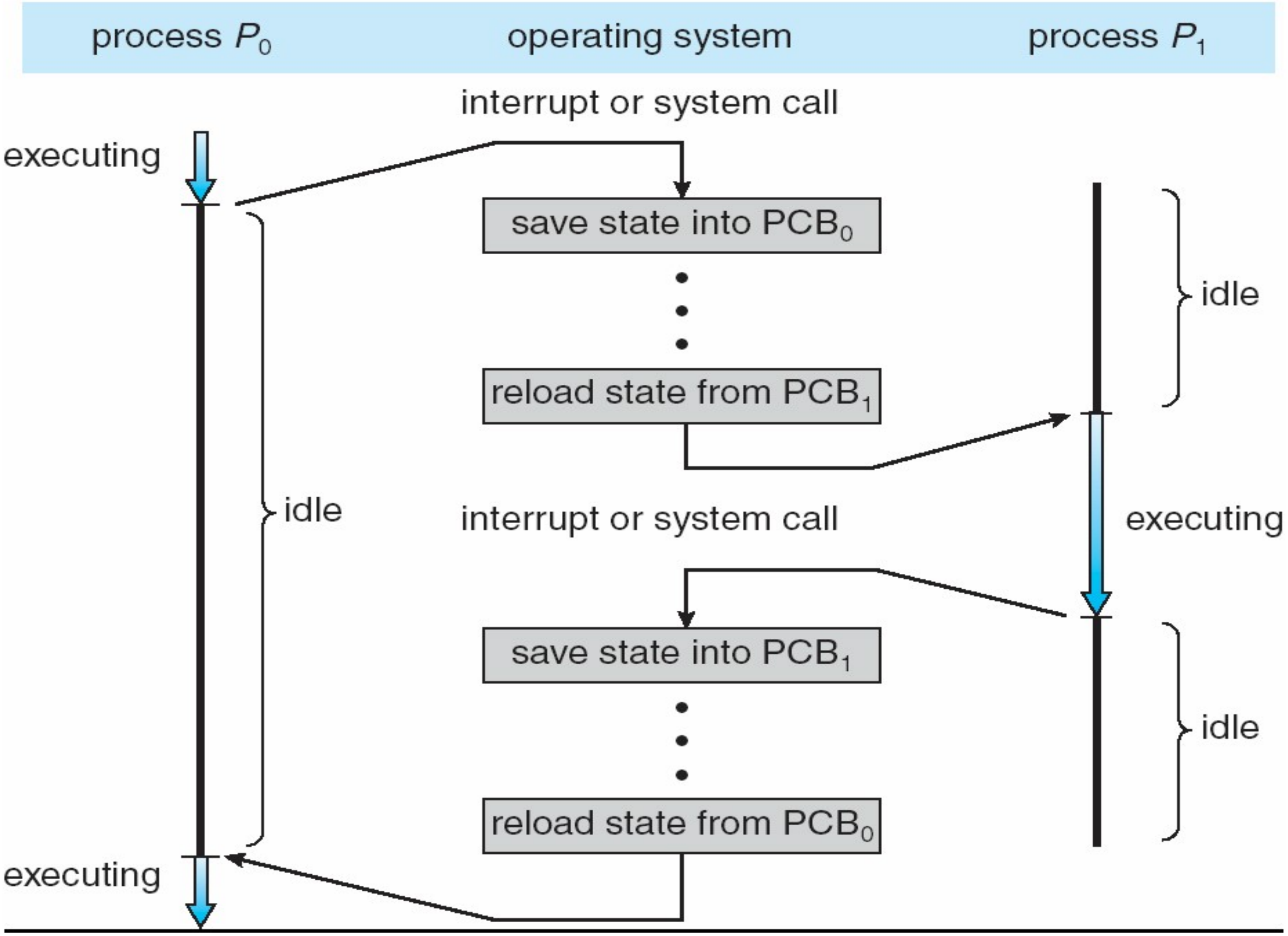
- **Stored in main memory**

- **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




Memory
Process 1
PCB
Program Counter 104
Code Section
add r1, r2
100
104
200 2 Data Section
300 3
Stack Section
Process 2
PCB
Program Counter 2004
Code Section
2000
2004
Data Section
Stack Section
Process n
PCB
Program Counter 30004
Code Section
30000
30004
Data Section
Stack Section
CPU
Program Counter (Instruction Pointer)
104
Instruction Register
add r1, r2
Registers, r1, r2, …
2
3

# CPU Switch From Process to Process



# 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 overhead; the system does no useful work while switching
  - The more complex the OS and the PCB -> longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU -> multiple contexts loaded at once

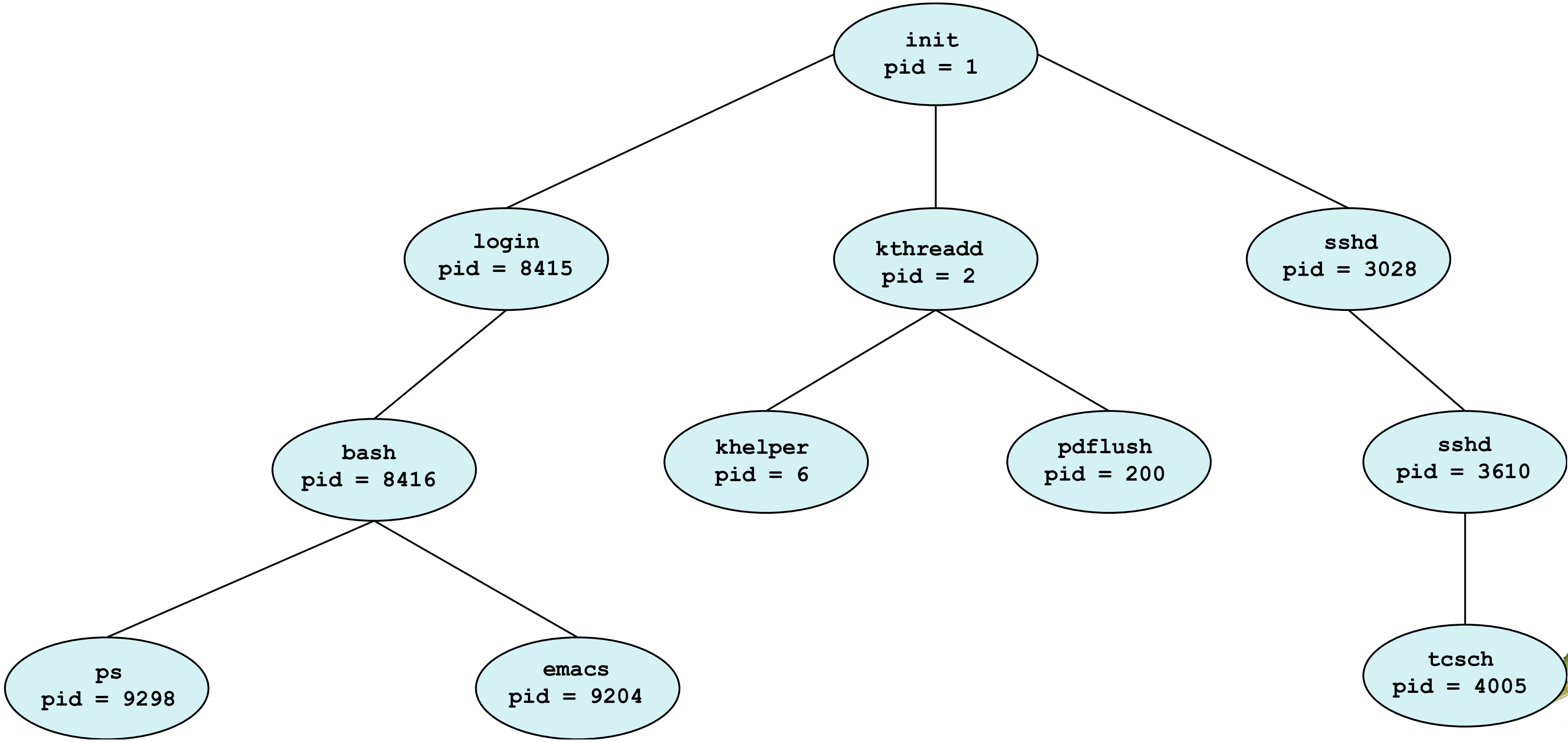
# Operations on Processes

- System must provide mechanisms for process **creation, termination**, and so on as detailed next

- Four Steps of Process creation
  - Create  PCB within OS kernel
  - Allocate memory space
  - Load binary program
  - Initialization of  program

# 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**)
- A Tree of Processes in Linux

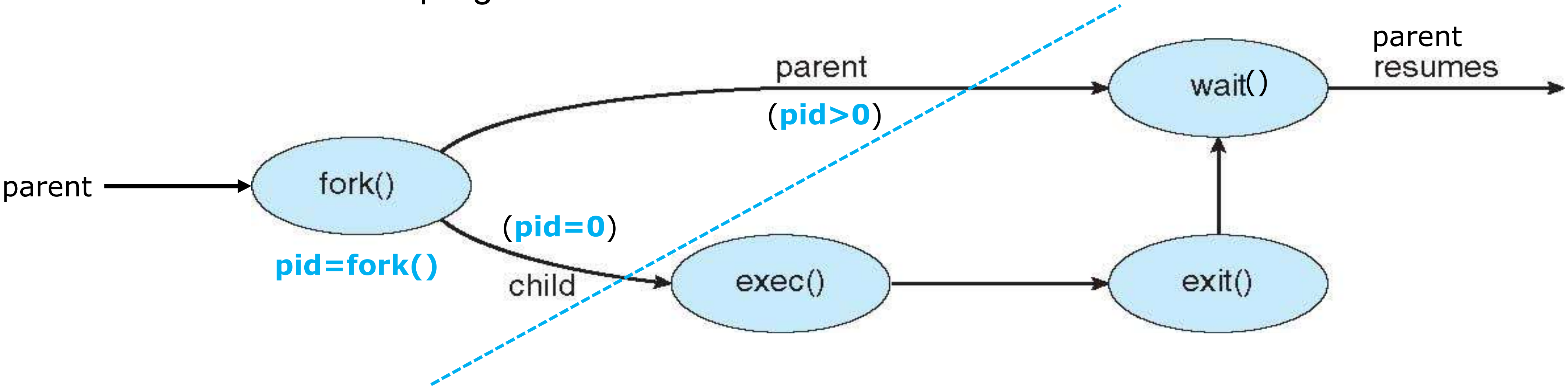


# Process Creation

- Resource(CPU time, memory, files, I/O devices) 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 some or all of its children have terminated
- **Address-space** possibilities for the new process:
  - Child process is **a duplicate of parent process** (it has the same program and data as the parent)
  - Child process has **a new program loaded into it**

# Process Creation (Cont.)

- **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



# C Program Forking Separate Process in UNIX

```
#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;
}
```

# Creating a Separate Process via Windows API

```
#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 terminates when it **executes last statement** and asks the OS to delete it by `exit()` system call
- Process may return a status value to its parent process via `wait()` system call
- Process' resources are deallocated by OS
- Parent process may wait for termination of a child process by using `wait()` :

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

- If no parent waiting, then terminated process is a **zombie**
- If parent terminated, processes are **orphans**

# Multiprocess Architecture – Chrome Browser

- Tabbed browsing and Active content
  - A **single instance of a web browser** application may open **several websites** at the same time, each site in a separate tab
  - Website may contain **active content** such as JavaScript, Flash, HTML5 which may contain software bugs, resulting in **sluggish response times** and even **crash**

- If web browsers ran as a **single process**(some still do)
  - one web site causes trouble, entire browser can hang or crash

# Multiprocess Architecture – Chrome Browser

- Google Chrome Browser is **multiprocess** with three different types
  - **Browser** process manages user interface, disk and network I/O
    - Created when Chrome is started; only one browser process is created
  - **Renderer** process renders web pages, deals with HTML, JavaScript, images
    - A new renderer process is created for each website opened in a new tab
  - **Plug-in** process for each type of plug-in such as Flash  or QuickTime in use
- The **advantage** of multiprocess approach
  - Websites **run in isolation** from one another; if one website crashes, only its renderer process is affected

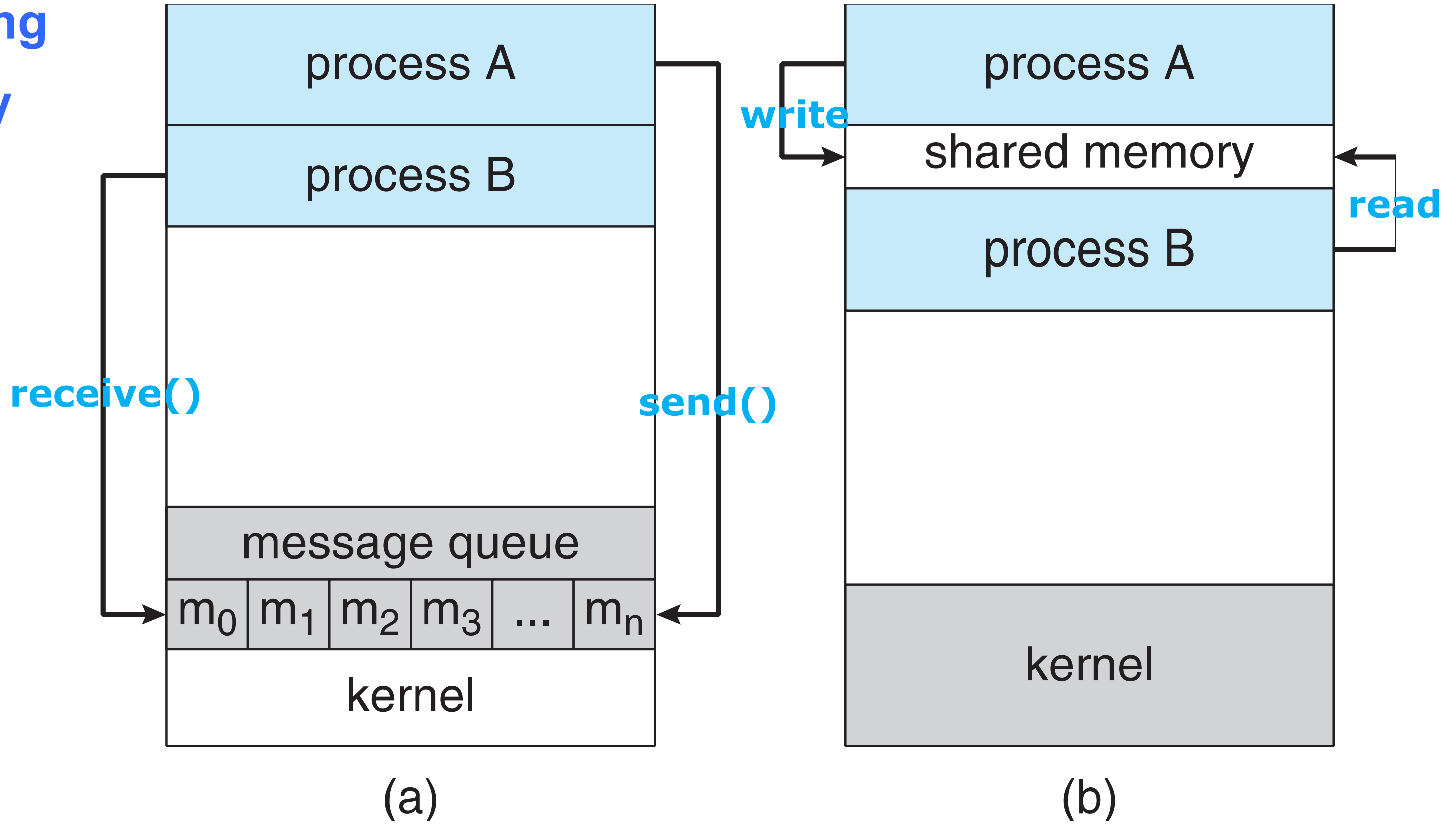
# Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
- ***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
- **Reasons** for cooperating processes:
  - **Information sharing**
  - **Computation speedup**
  - **Modularity**
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
  - Mechanism for processes to **communicate data** and to **synchronize their actions**

# Interprocess Communication

- Two Communications models of IPC
  - **Message passing**
  - **Shared memory**

- **Synchronization**
  - **Mutex**
  - **Lock,**
  - **Semaphore**
  - **Critical Section**
  - **See Ch.6**




**Message Passing** **Shared Memory**

# Interprocess Communication – Message Passing

- Mechanism for 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*) – message size fixed or variable
  - **`receive`**(*message*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a ***communication link*** between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., direct or indirect, synchronous or asynchronous, automatic or explicit buffering)

# 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 bi-directional

# 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 processes share a common mailbox**
  - A link may be associated **with many processes**
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

# Indirect Communication

- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox

- Primitives are defined as:

  **`send`**(*A, message*) – send a message to **mailbox *A***

  **`receive`**(*A, message*) – receive a message from **mailbox *A***

# Message Passing - Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking** send has the sender send the message and continue
  - **Non-blocking** receive has the receiver receive a valid message or null

# Synchronization (Cont.)

- Different combinations possible
  - If **both send and receive are blocking**, we have a **rendezvous** between the sender and the receiver
- Producer-consumer problem becomes trivial

```
message next_produced;
while (true) {
      /* produce an item in next_produced */
   send(next_produced);
}
message next_consumed;
while (true) {
   receive(next_consumed);

   /* consume the item in next_consumed */
}
```

# Buffering

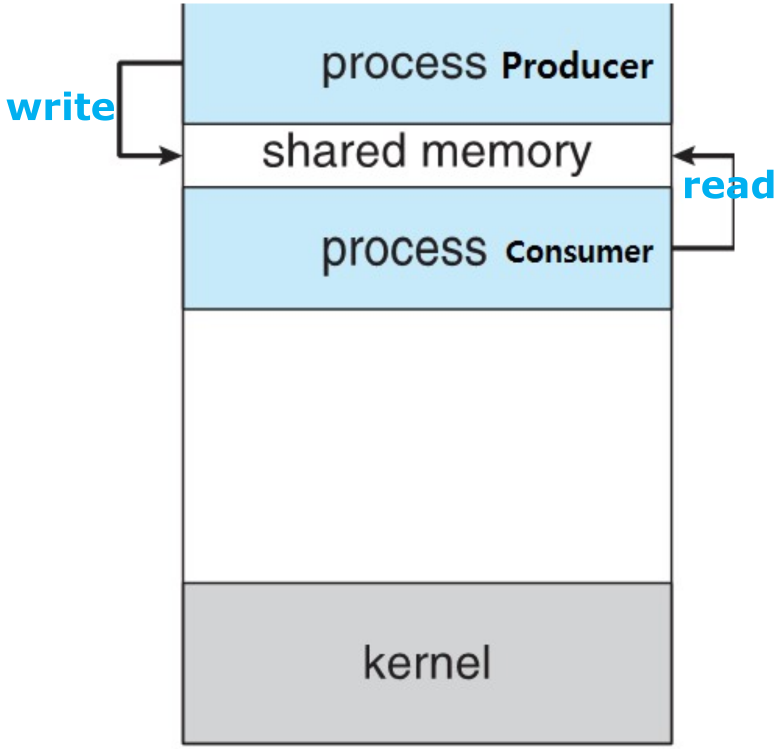
- Queue of messages attached to the link; implemented in one of three ways
    1. Zero capacity – 0 messages
       Sender must wait for receiver (rendezvous)
    2. Bounded capacity – finite length of *n* messages
       Sender must wait if link full
    3. Unbounded capacity – infinite length
       Sender never waits

# Examples of IPC Systems - POSIX

- **POSIX Shared Memory**
  - Process first creates shared memory segment
    ```
    shm_fd = shm_open(name, O CREAT | O RDWR, 0666);
    ```
  - Also used to open an existing segment to share it
  - Set the size of the object
    ```
    ftruncate(shm_fd, 4096);
    ```
  - Now the process could write to the shared memory
    ```
    sprintf(shared memory, "Writing to shared memory");
    ```

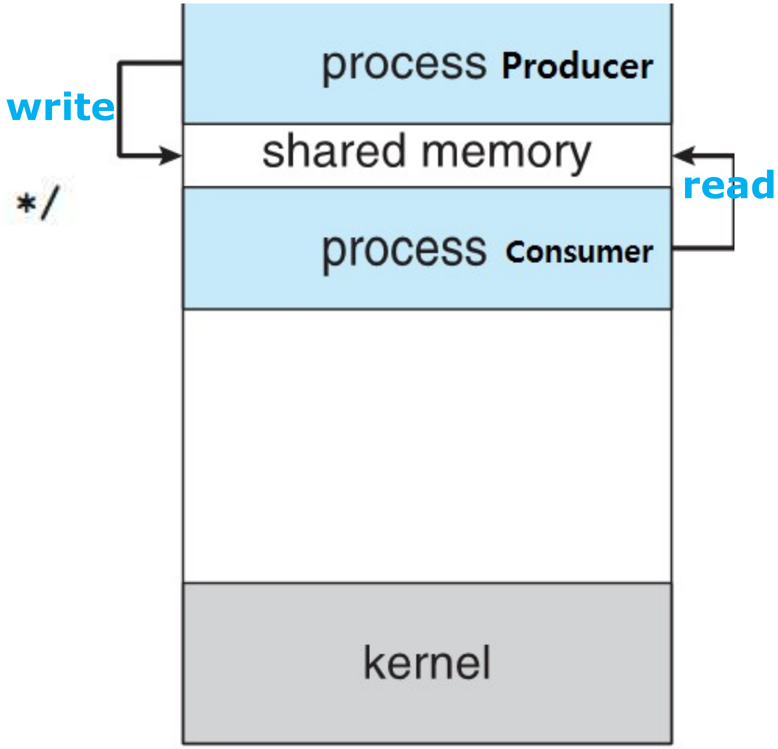
# IPC POSIX Producer

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* create the shared memory object */
   shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

   /* configure the size of the shared memory object */
   ftruncate(shm_fd, SIZE);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

   /* write to the shared memory object */
   sprintf(ptr,"%s",message_0);
   ptr += strlen(message_0);
   sprintf(ptr,"%s",message_1);
   ptr += strlen(message_1);

   return 0;
}
```



# IPC POSIX Consumer

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* open the shared memory object */
   shm_fd = shm_open(name, O_RDONLY, 0666);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

   /* read from the shared memory object */
   printf("%s",(char *)ptr);

   /* remove the shared memory object */
   shm_unlink(name);

   return 0;
}
```



# End of Chapter 3