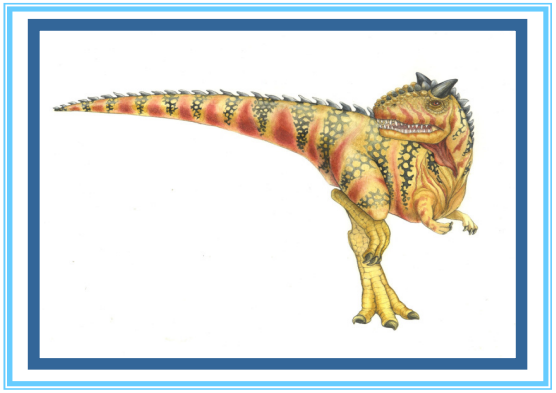
Lecture 2: Processes



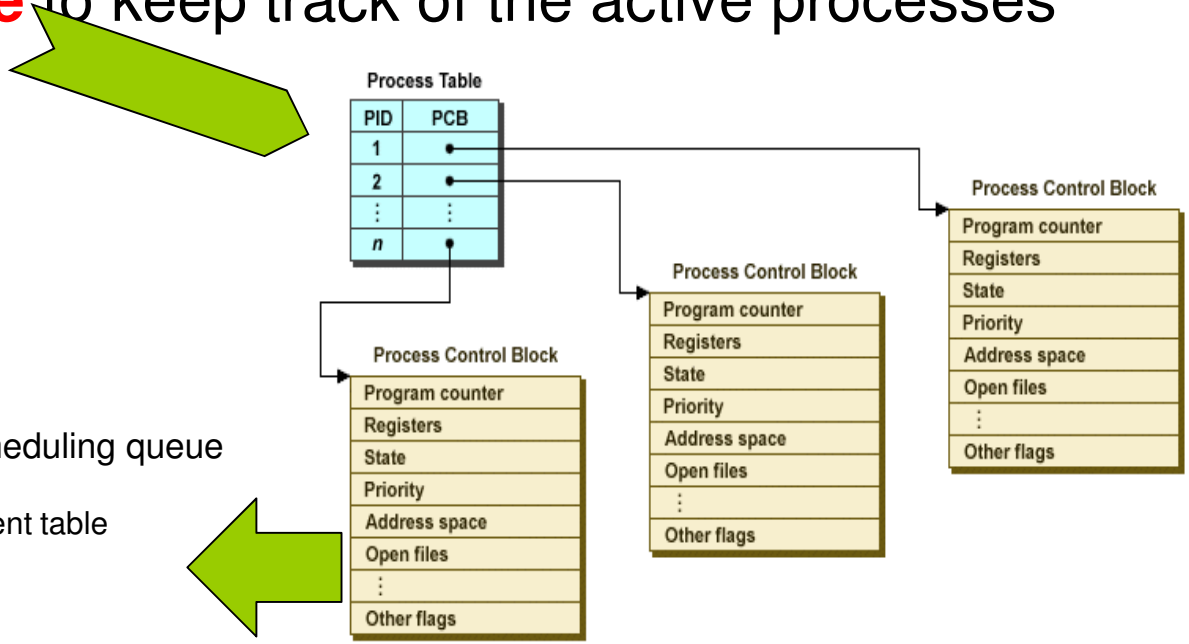
Silberschatz, Galvin and Gagne ©2009

Lecture 2: Processes

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Examples of IPC Systems
- Communication in Client-Server Systems

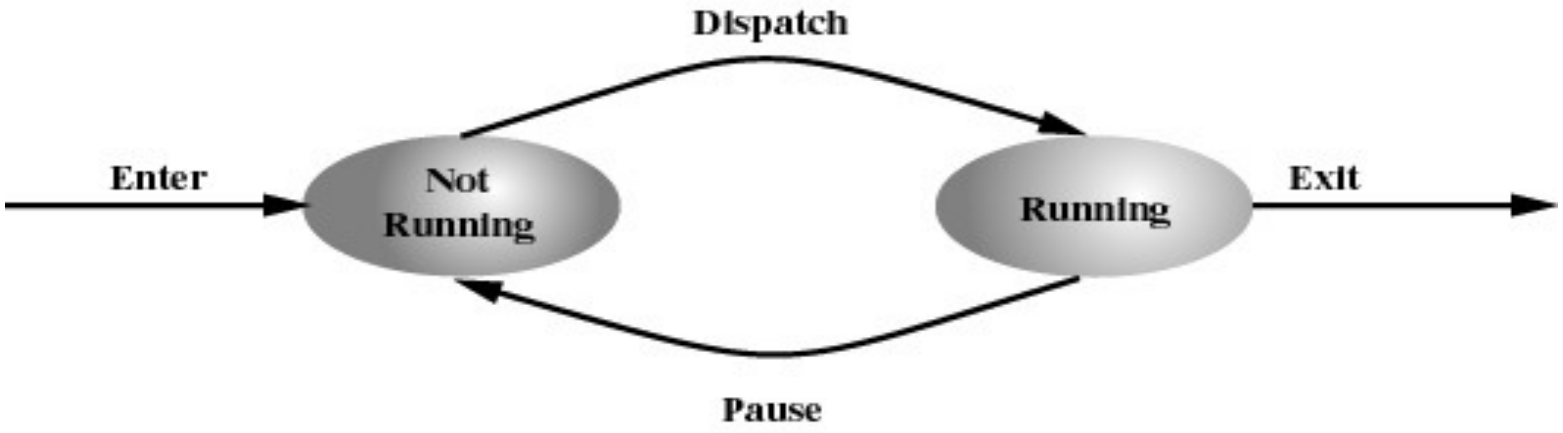
The Process

- A process is - Execution of an individual program.
- Each time a process is created,
  - OS must create a complete independent address space (base, limit)
  - (i.e., processes do not share their heap or stack data)
- OS maintains a **process table** to keep track of the active processes
- Information for each process:
  - Program id, user id, group id
  - Program status word
  - CPU register values
  - CPU Scheduling-process priority, pointer to scheduling queue
  - Memory maps-base/limit register, page table, segment table
  - Stack pointer
  - I/O status Information-allocated I/O devices, list of Open files
  - Accounting information, etc.-amount of CPU & real time used, time limits, account numbers, job/process number



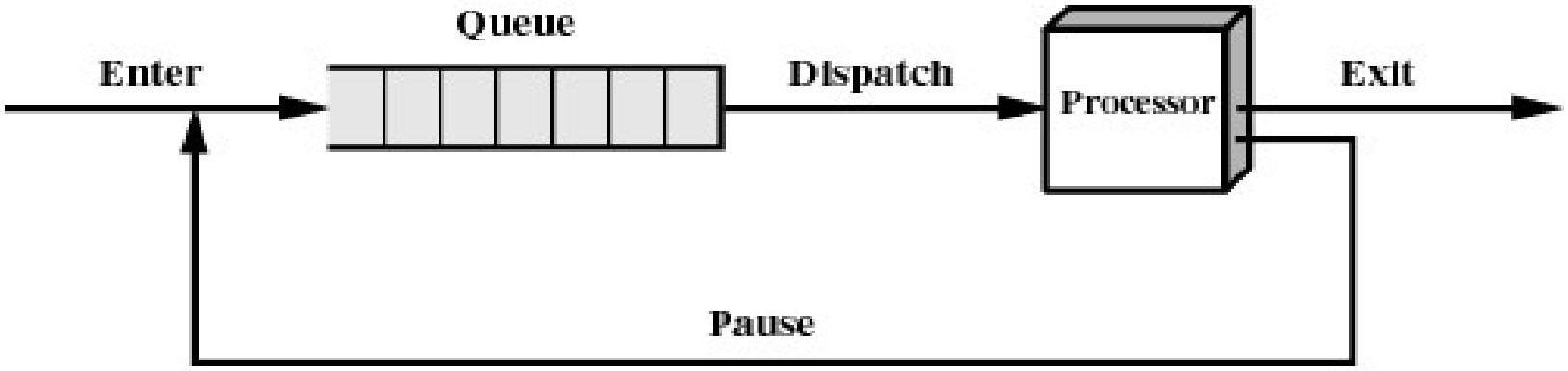
Two-State Process Model

- Process may be in one of two states
  - Running
  - Not-running



(a) State transition diagram

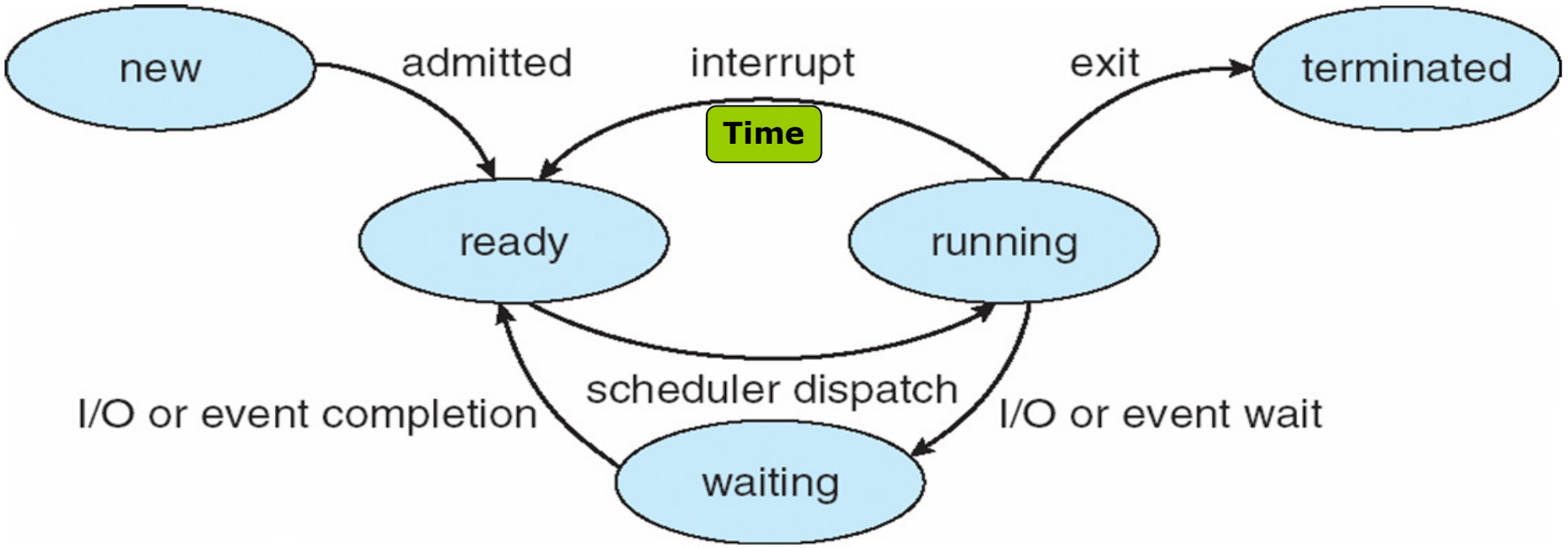
Not-Running Process in a Queue



(b) Queuing diagram

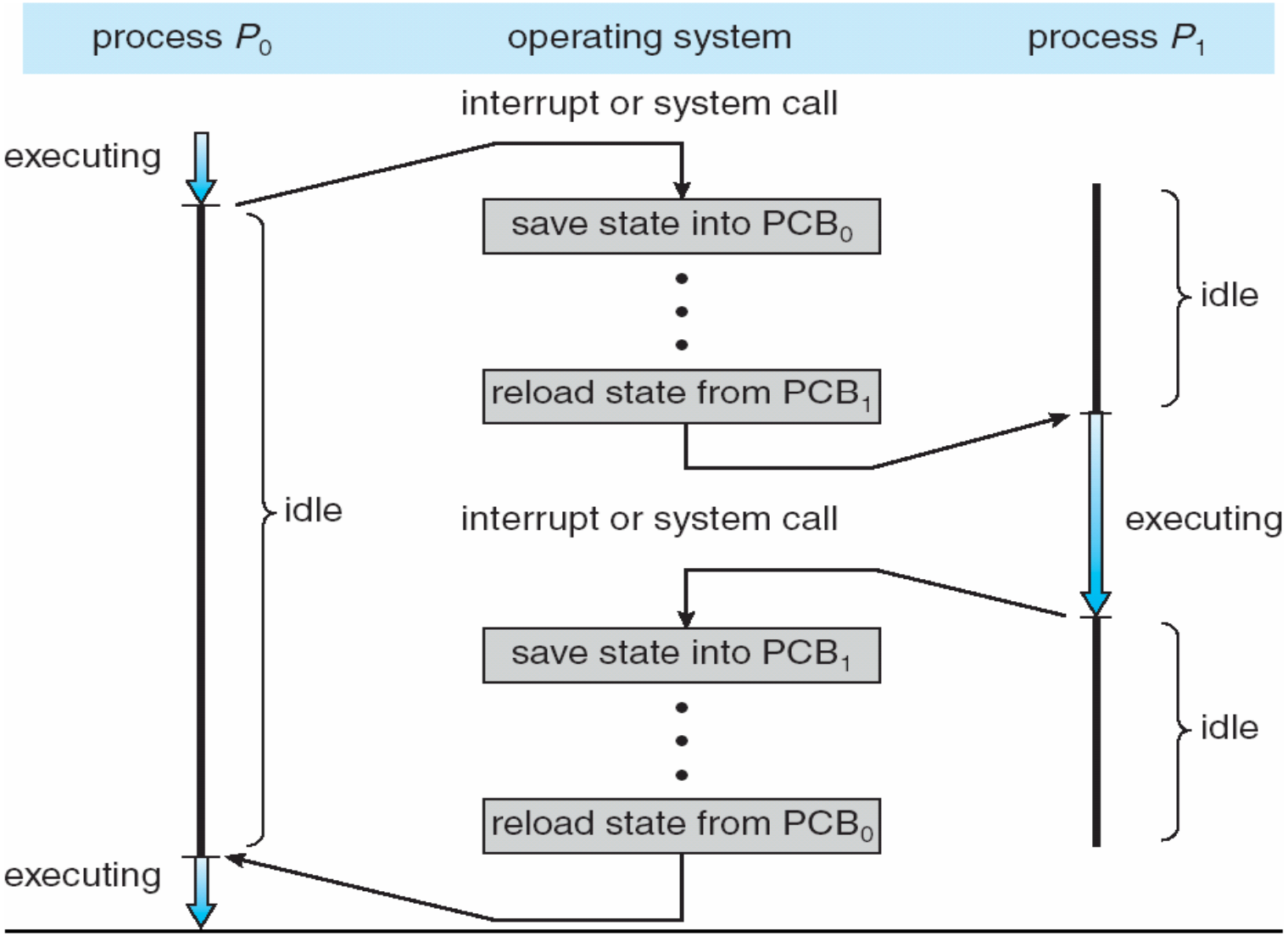
Five States Process Model

- 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



©2009

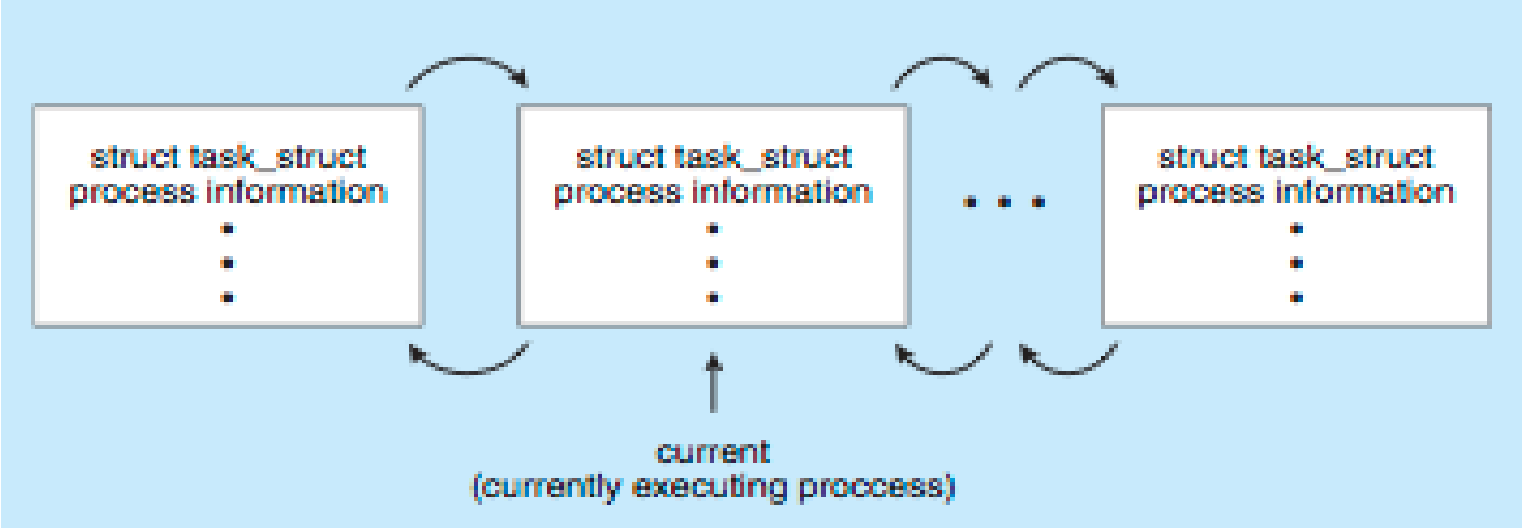
CPU Switch From Process to Process



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 pro */
  ```

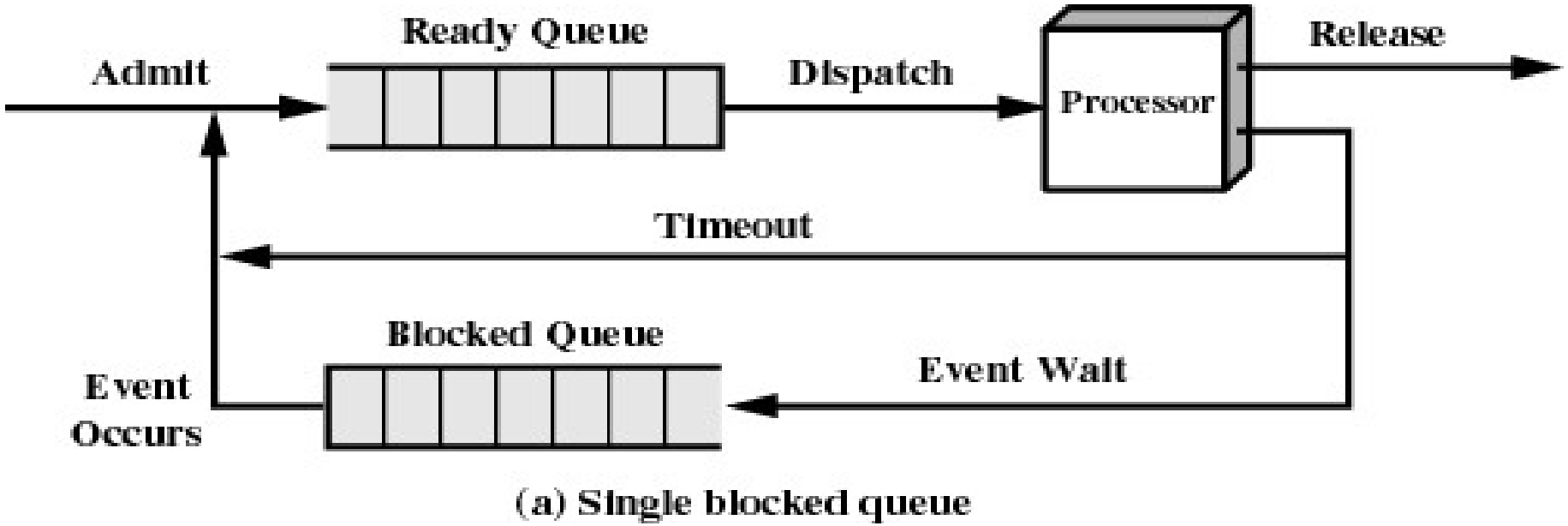



  ```

Process Scheduling

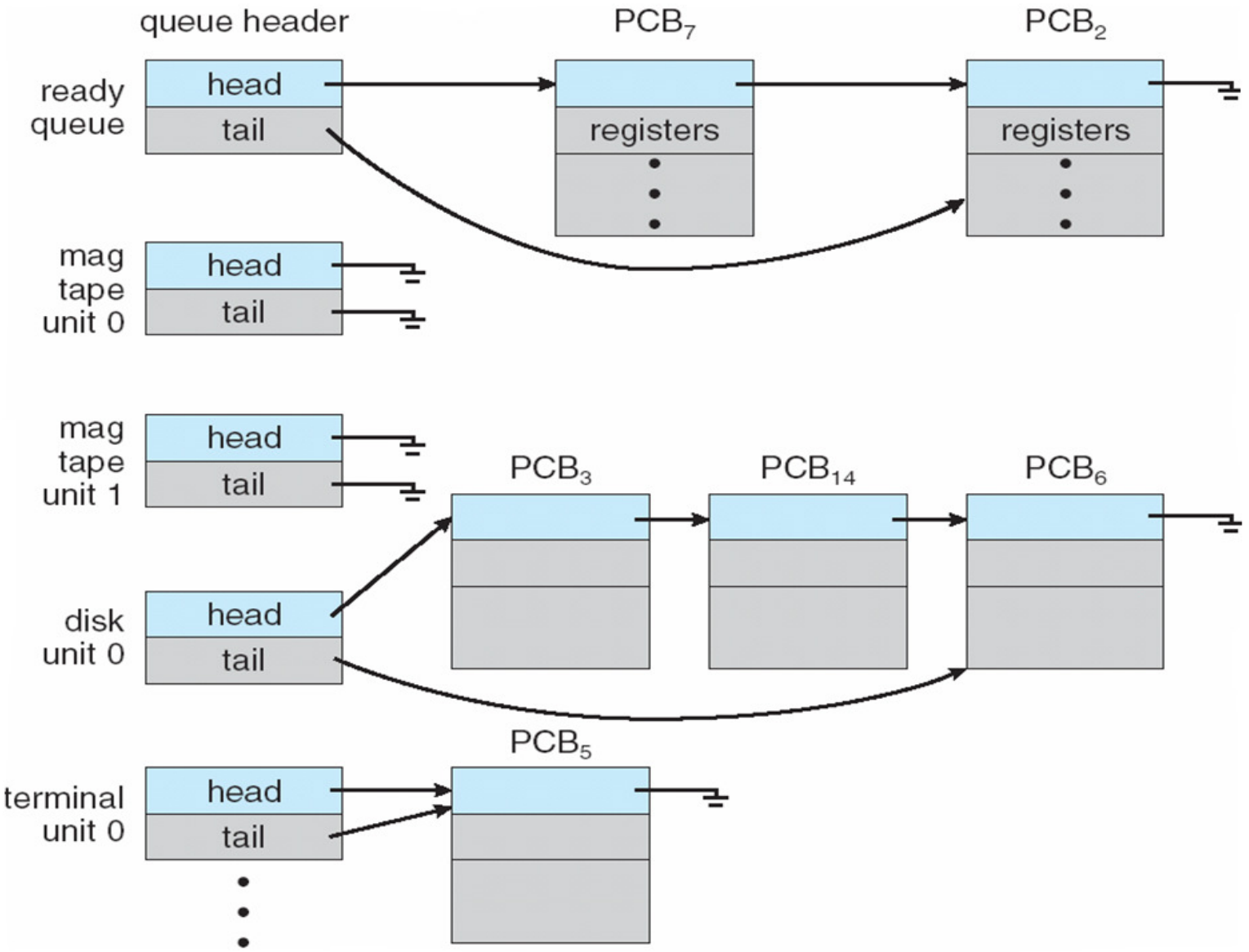
- AIM: Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects process among availables for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system (devices-SPOOLING)
  - **Ready queue** – 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

Using Two Queues

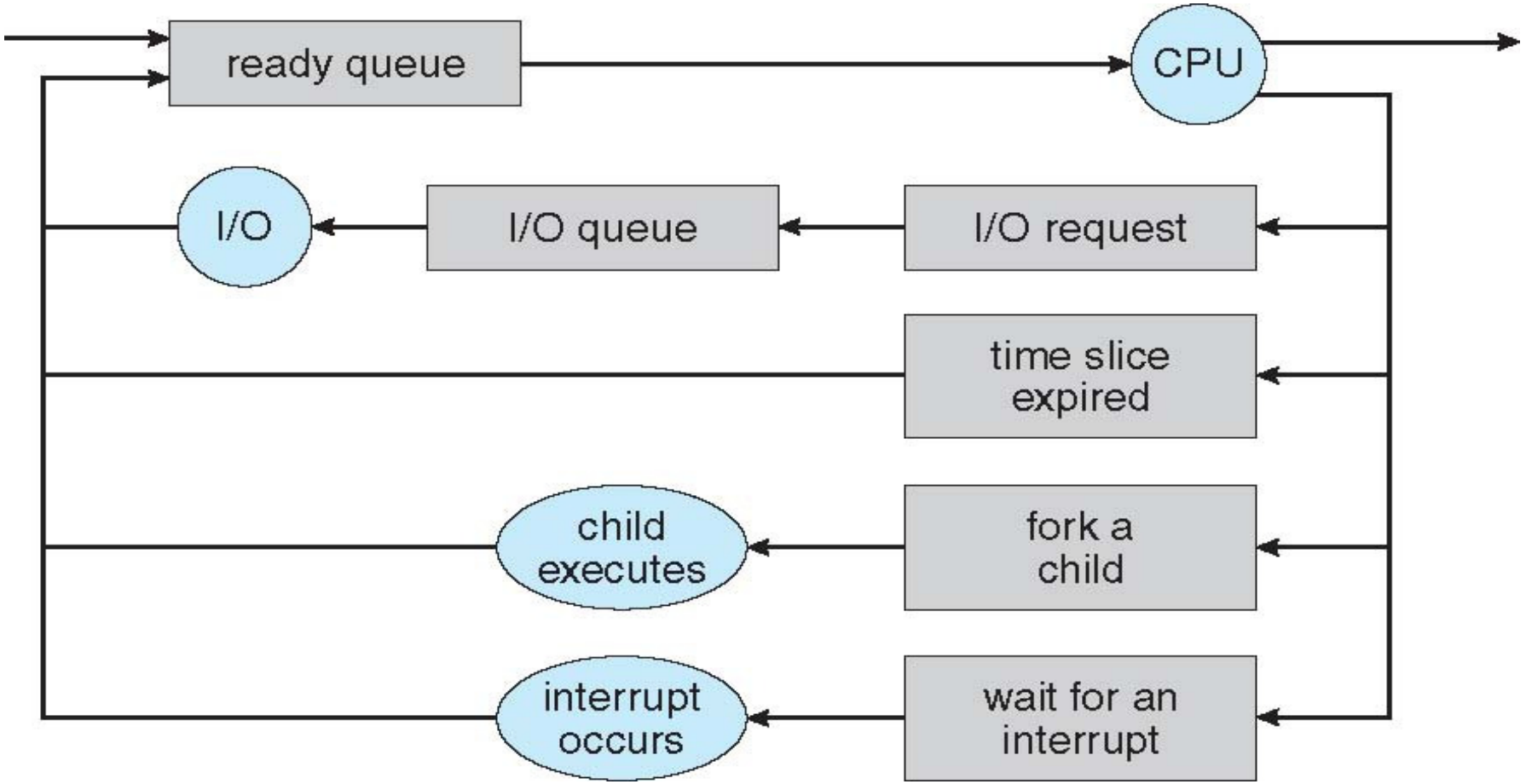


(a) Single blocked queue

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling



Schedulers

Long-term scheduler/Job scheduler

- which processes should be brought into the ready queue
- invoked very infrequently (seconds, minutes) ⇒ (may be slow)
- controls the *degree of multiprogramming*

Short-term scheduler/CPU scheduler

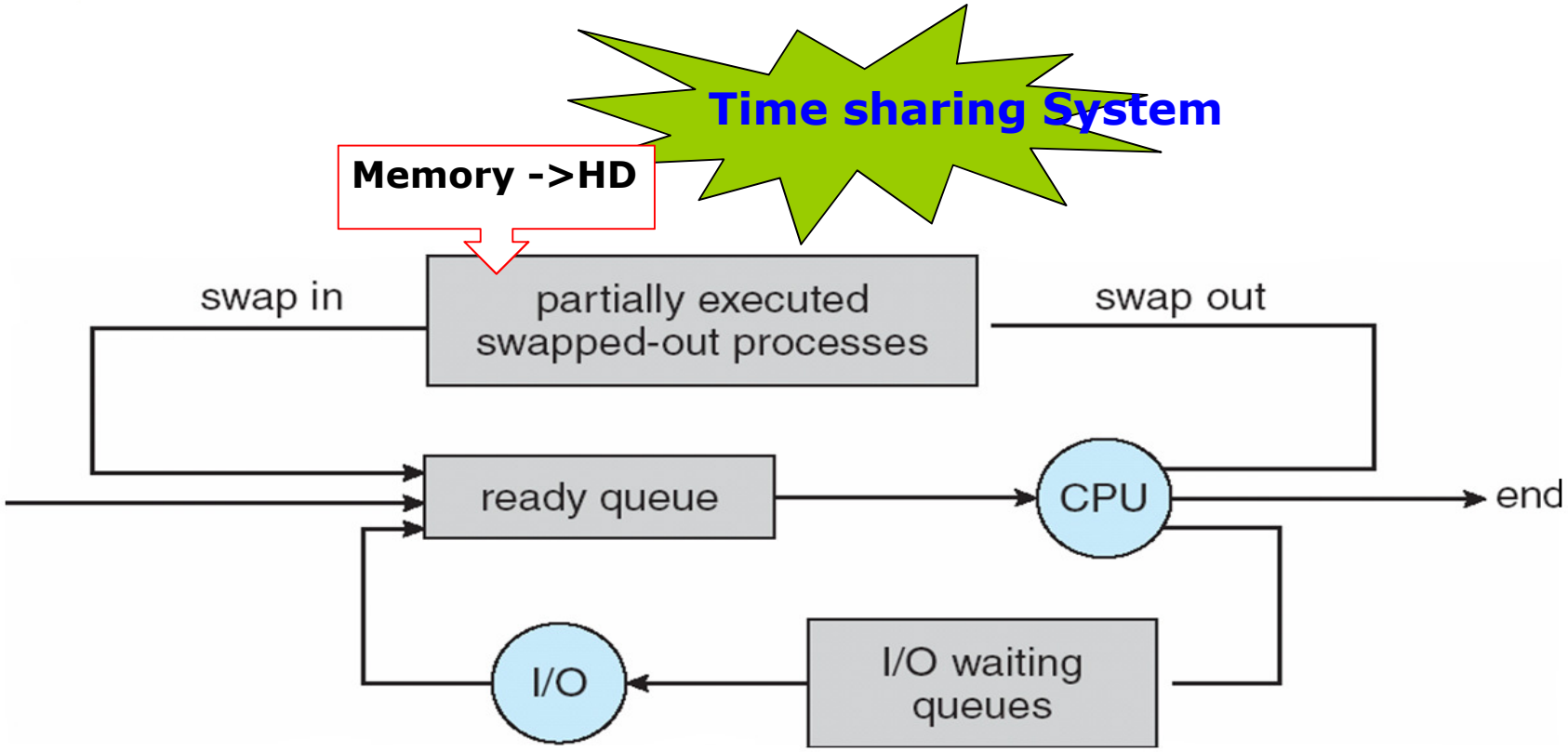
- which process should be executed next and allocates CPU
- invoked very frequently (milliseconds) ⇒ (must be fast)

Balanced

I/O-bound process

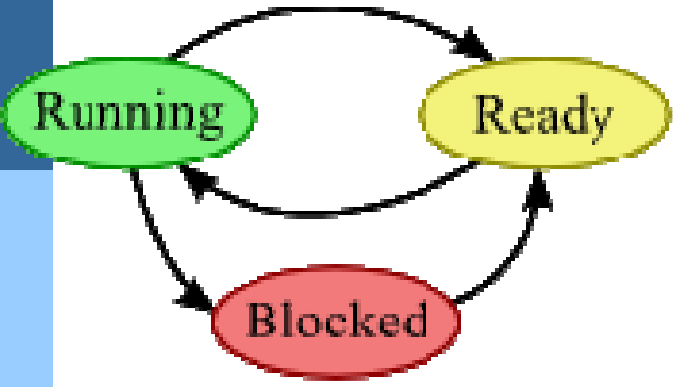
CPU-bound process

Addition of Medium Term Scheduling



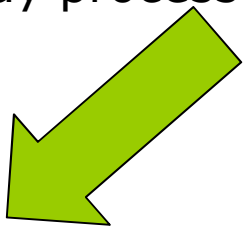
Context Switch

Three States: during a process's life



Context switch

Operating system moves the process (for I/O) from the running state to the blocked state, and promote some ready process into the running state instead.

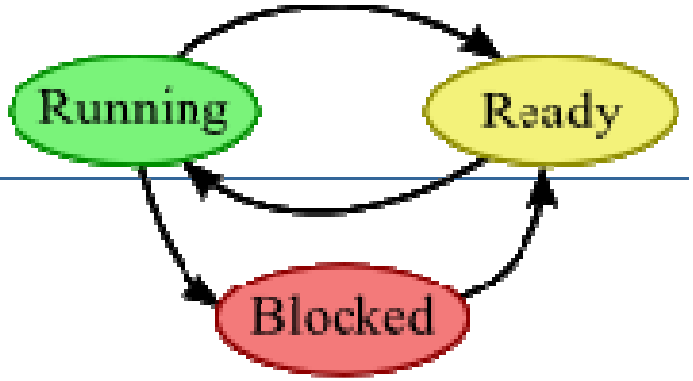


Running -> Blocked->Ready

1. The running process sends a system call via an interrupt.
2. The CPU jumps into the interrupt handler, which is part of the OS.

3. The OS saves all the registers of the running process into that process's entry of the process table.
   - If another process is already waiting for the device to respond, the OS places the process into a waiting queue for that device. Otherwise, the process's request is sent to the device.
4. The OS selects the next process to execute from the queue of those processes in the ready state. This is called the **ready queue**.
5. The OS restores the registers to the values saved in the next process's entry of the process table.
6. The OS returns to the program counter value stored in the next process's entry of the process table.

Blocked->Ready-> Running



- Much later, when the device has found the requested data, it will send a hardware interrupt to the CPU. This interrupts whichever process is currently running, and the CPU begins instead executing the OS's interrupt handler. The handler proceeds as follows.
  - The OS saves the device's response in memory for the requesting process to use when it gets the CPU again.
  - The OS moves the blocked process into the ready queue.
  - If there are processes waiting for the device, the OS sends the next request to the device.
  - The OS returns back to the process that was running at the time the interrupt occurred.

Process Creation

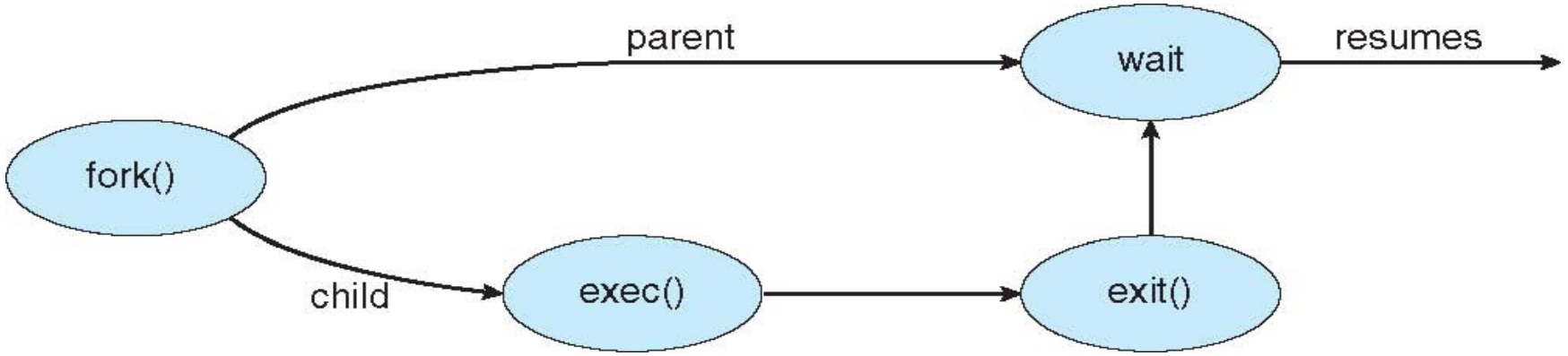
- **Parent** process creates **children** processes, which, in turn create other processes, forming a tree of processes.
- **Process identifier** (**pid**): process identified and managed.
- **Resource sharing**
  - Parent and children share all resources
  - Children share subset of parent's resources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

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
    - As a new process is not created, the process identifier (PID) does not change, but the machine code, data, heap, and stack of the process are replaced by those of the new program.

Process Creation



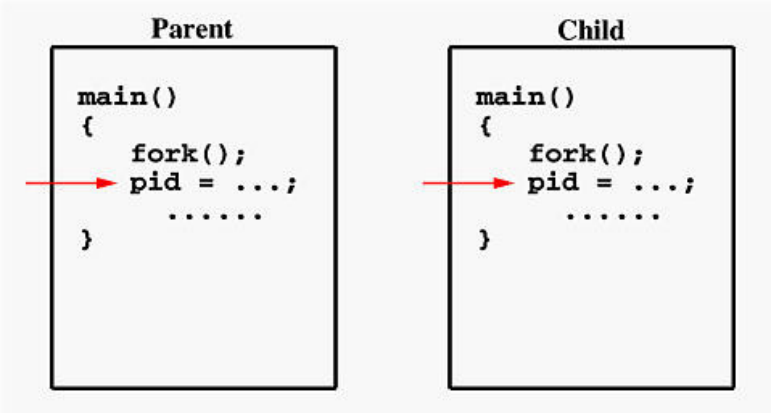
The FORK() System Call

Processes in UNIX are created with the fork() system call.

```
#include <sys/types.h>
#include <unistd.h>

void main(void)
{
    int pid;
    pid = fork();
    if (pid == -1) {
        printf("error in process creation\n")
        exit(1);
    }
    else if (pid == 0) child_code();
    else parent_code();
}
```

Unix will make an exact copy of the parent's address space and give it to the child. Therefore, the parent and child processes have separate address spaces.



The FORK() System Call

```
#include <sys/types.h>
#include <unistd.h>

#define PROCESS 10

void main(void)
{
    int pid, j;

    for (j=0; j < PROCCESS; j++) {
        pid = fork();
        if (pid == -1) {
            printf("error in creation of process %d\n", j);
            exit(1);
        }
        else if (pid == 0)  child_code(j);
    }
     for (j = 0; j < PROCESS; j++) wait(0);
}
```

```
void child_code(int id)
{
        pid_t myid, pid;
        myid = getpid();
        pid = getppid();

        printf("My pid is %d and my parent's id is %d",
                myid, pid);

        printf("My virtual id is %d\n", id);
        exit(0);
}
```

If we are interested to wait for a particular child, we can use instead of wait(), **waitpid**() (see man pages for more information).

C Program Forking Separate Process

```
#include <sys/types.h>
#include <studio.h>
#include <unistd.h>
int main()
{
pid_t  pid;
      /* fork another 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 */
            wait (NULL);
            printf ("Child Complete");
      }
      return 0;
}
```

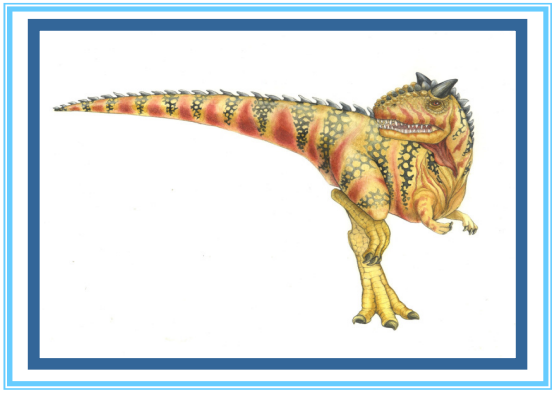
Process Termination

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

No Wait: Zombie process
Specially handle by OS

- Parent may terminate execution of children processes (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is terminated
    - some operating systems do not allow child to continue if its parent terminates
      - All children then also terminated – called **cascading termination**

End of Chapter 3



Silberschatz, Galvin and Gagne ©2009