Operating Systems (CSE531)
Lecture # 06

Manish Shrivastava

LTRC, IIIT Hyderabad

Outline

- Process Concept
- Process Control Block (PCB)
- Process Scheduling
- Operations on Processes
- Cooperating Processes
- Next Class: Inter-process Communication

Process Concept

- Construction site
    - Many worker working simultaneously
    - Question: How to organize them effectively to improve the performance ?
    - Sitting/Idle vs Working
    - Working worker is similar to process
        - Uses tools
        - Cooperation and synchronization among them is required.
    - The manager should have some information about the working worker.
        - What kind of tools S/he is using.
        - The nature of the job, and status of the job.

- In computer system
    - Operating system is similar to manager
    - The sitting workers are similar to the programs reside on the disk.
    - The working workers are processes
    - CPU is an expensive tool which should not be kept idle. It is OK if some workers wait for expensive tool.

Process Concept

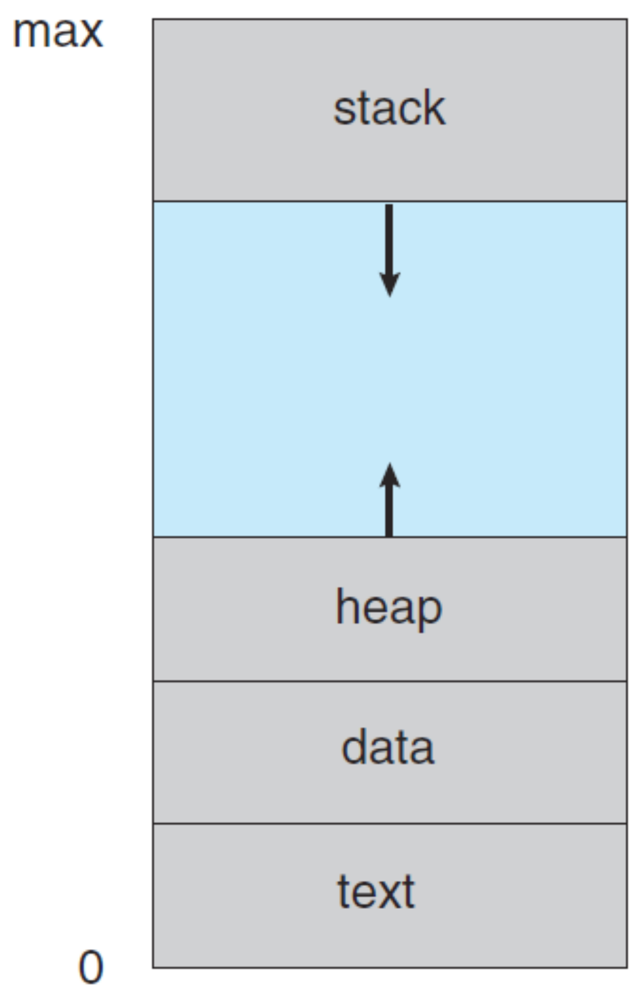
- Early OS: One program at a time with complete control.
- Modern OS: allow multiple programs to be loaded in to memory and to be executed concurrently.
- This requires firm control over execution of programs.
- The notion of process emerged to control the execution of programs.
- A process
  - Unit of work
  - Program in execution
- OS consists of a collection of processes
  - OS processes executes system code.
  - User processes executes user code.
- By switching CPU between processes, the OS can make the computer more productive.

Process Concept

- Process (task or job) includes the current activity.–
    - a program in execution;
- process execution must progress in sequential fashion.
- The components of a process are
    - The program to be executed
    - The data on which the program will execute
    - The resources required by the program– such as memory and file (s)
    - The status of execution: a) Program counter b) Stack
- Multiple processes can be associated with the same program.
- For CPU, all processes are similar
    - Batch Jobs and user programs/tasks
    - Word file, Internet browser
    - System call
    - Scheduler
    - ….

Process Concept

- Process in Memory

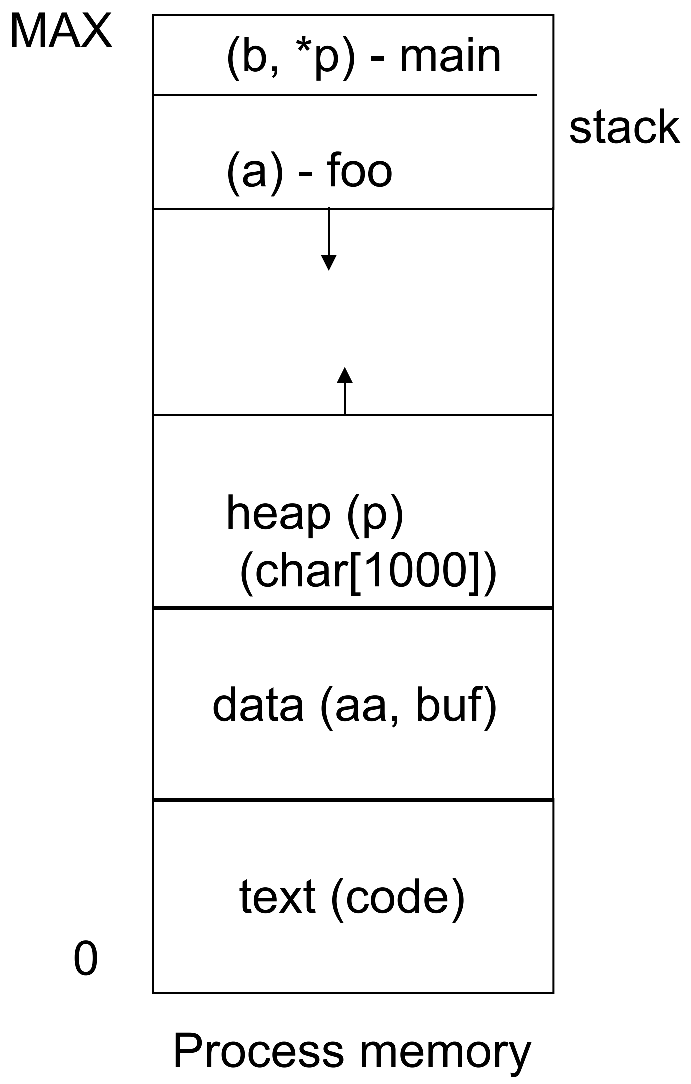


Process memory

Process Concept

- Example: User level context

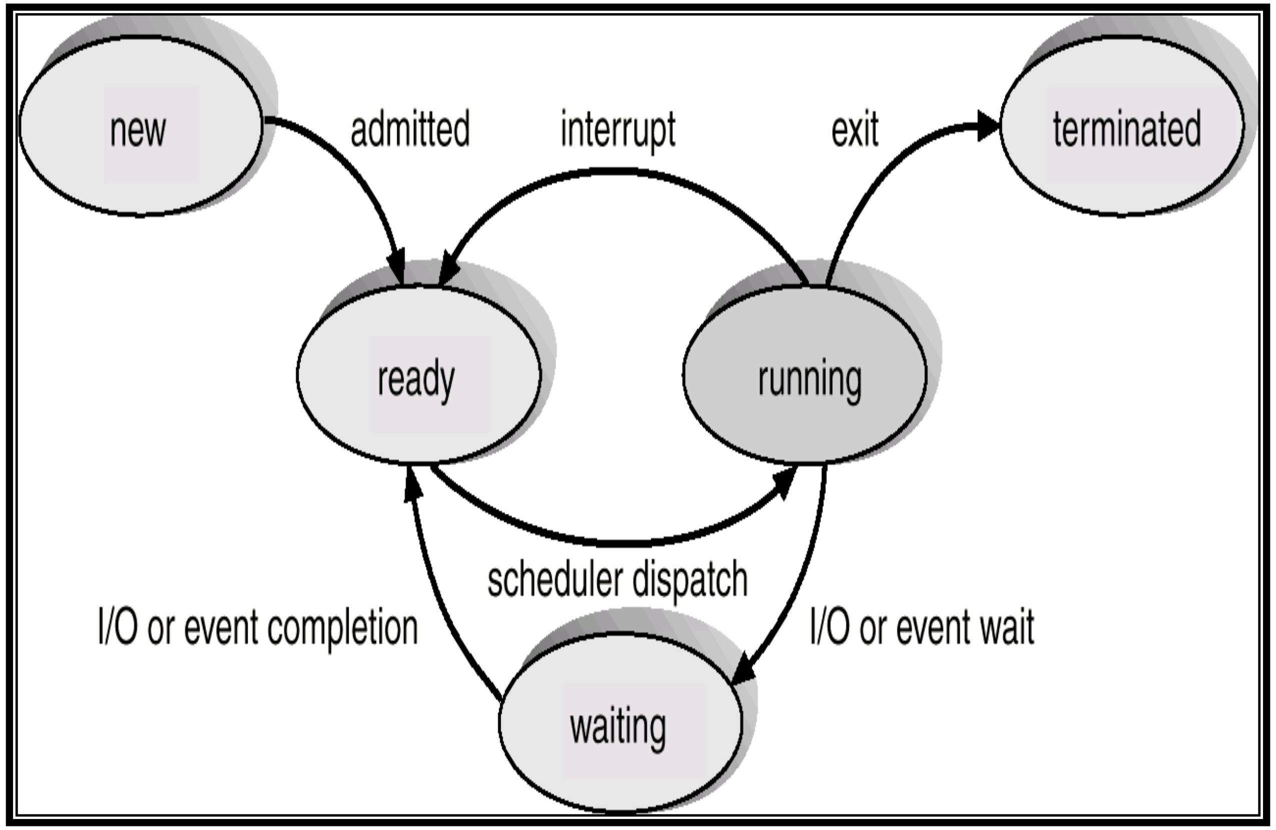
```
…
int aa;
char buf[1000];
void foo() {
  int a;
   …
}
main() {
  int b;
  char *p;
  p = new char[1000];
  foo();
}
```



Process State

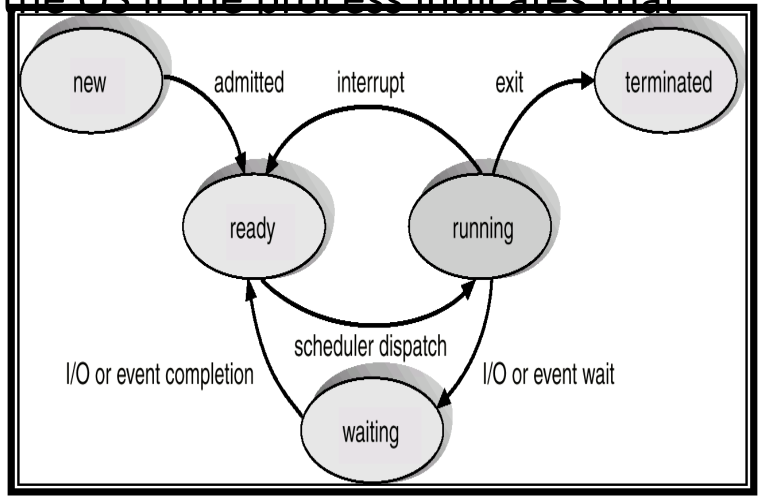
- As a process executes, it changes *state*
- *Each process can be in one of*
  - **new**: The process is being created.
  - **running**: Instructions are being executed.
  - **waiting**: The process is waiting for some event to occur.
  - **ready**: The process is waiting to be assigned to a process.
  - **terminated**: The process has finished execution.
- The names may differ between OSs.

Process State



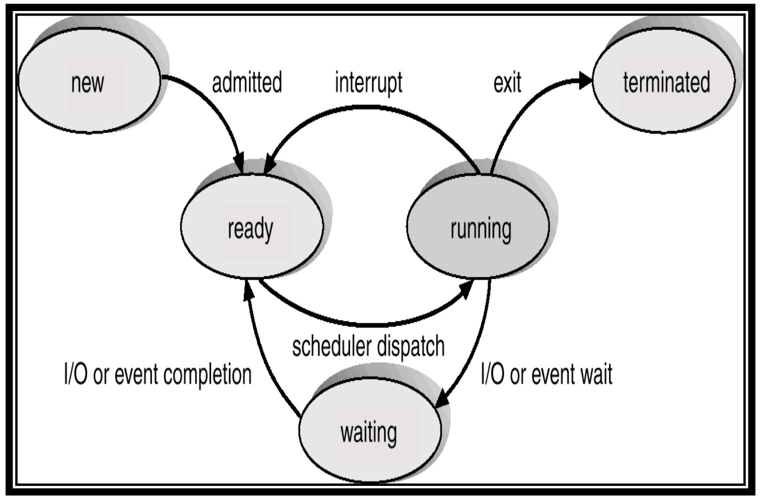
State Change

- **Null → New :** a new process is created to execute the program
  - New batch job, log on
  - Created by OS to provide the service
- **New → ready**: OS will move a process from prepared to ready state when it is prepared to take additional process.
- **Ready → Running**: when it is a time to select a new process to run, the OS selects one of the process in the ready state.
- **Running → terminated:** The currently running process is terminated by the OS if the process indicates that it has completed, or if it aborts.



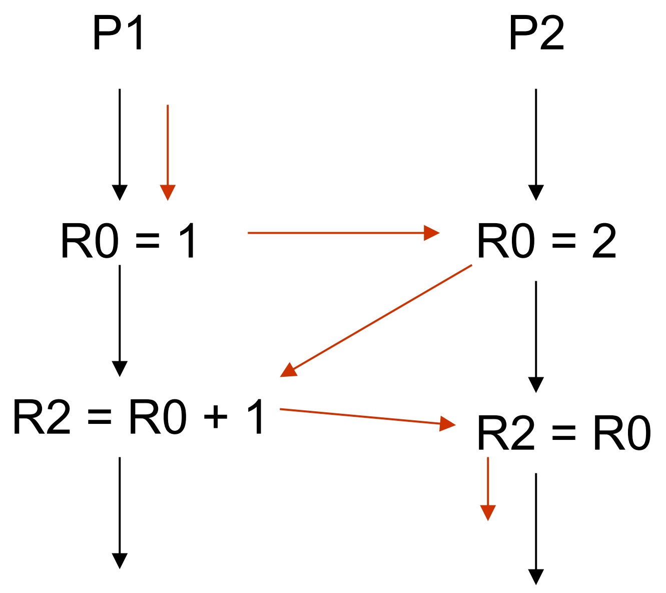
State Change

- **Running → Ready**: The process has reached the maximum allowable time or interrupt.
- **Running → Waiting:** A process is put in the waiting state, if it requests something for which it must wait.
  - Example: System call request.
- **Waiting → Ready**: A process in the waiting state is moved to the ready state, when the event for which it has been waiting occurs.
- **Ready → Terminated:** If a parent terminates, child process should be terminated
- **Waiting → Terminated:** If a parent terminates, child process should be terminated



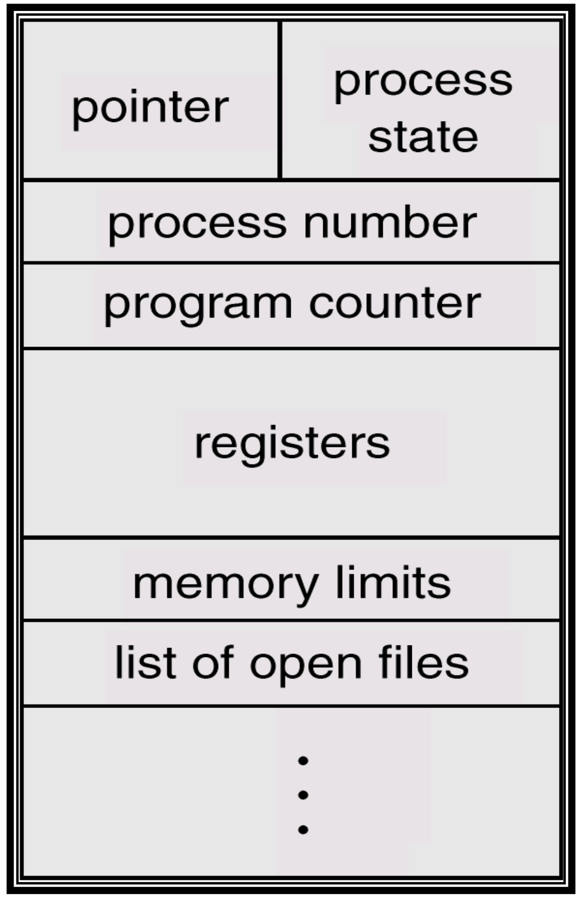
Process Context

- Contains all states necessary to run a program
  - Is the user level context sufficient?
    - Only if the system runs through one program at a time
    - The system typically needs to switch back and forth between programs.



- R2 in P1 is wrong. How to make It correct?
  - Save R0 in P1 before switching
  - Restore R0 in P1 when switching from P2 to P1.
- Registers should be a part of process context: the **register context**!

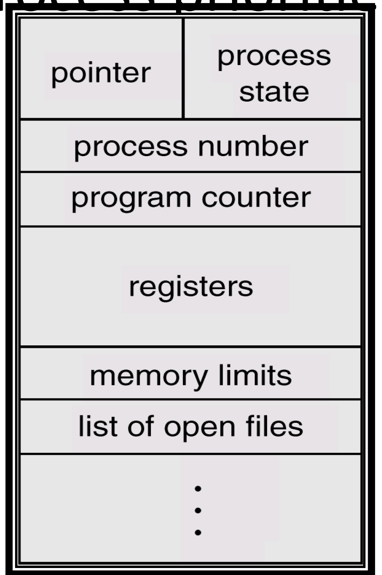
Process Control Block (PCB)



Process Control Block (PCB)

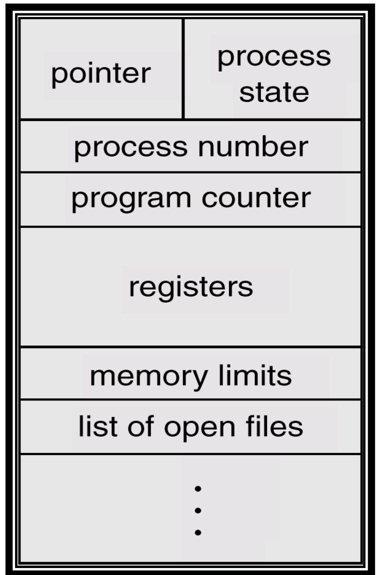
Information associated with each process.

- **Process state**: new, ready, running,…
- **Program Counter (PC)**: address of the next instruction to execute
- **CPU registers**: data registers, stacks, condition-code information, etc.
- **CPU scheduling information**: process priorities, pointers to scheduling queues, etc.



Process Control Block (PCB)

- **Memory-management information**: locations including value of base and limit registers, page tables and other virtual memory information.
- **Accounting information**: the amount of CPU and real time used, time limits, account numbers, job or process numbers etc.
- **I/O status information**: List of I/O devices allocated to this process, a list of open files, and so on

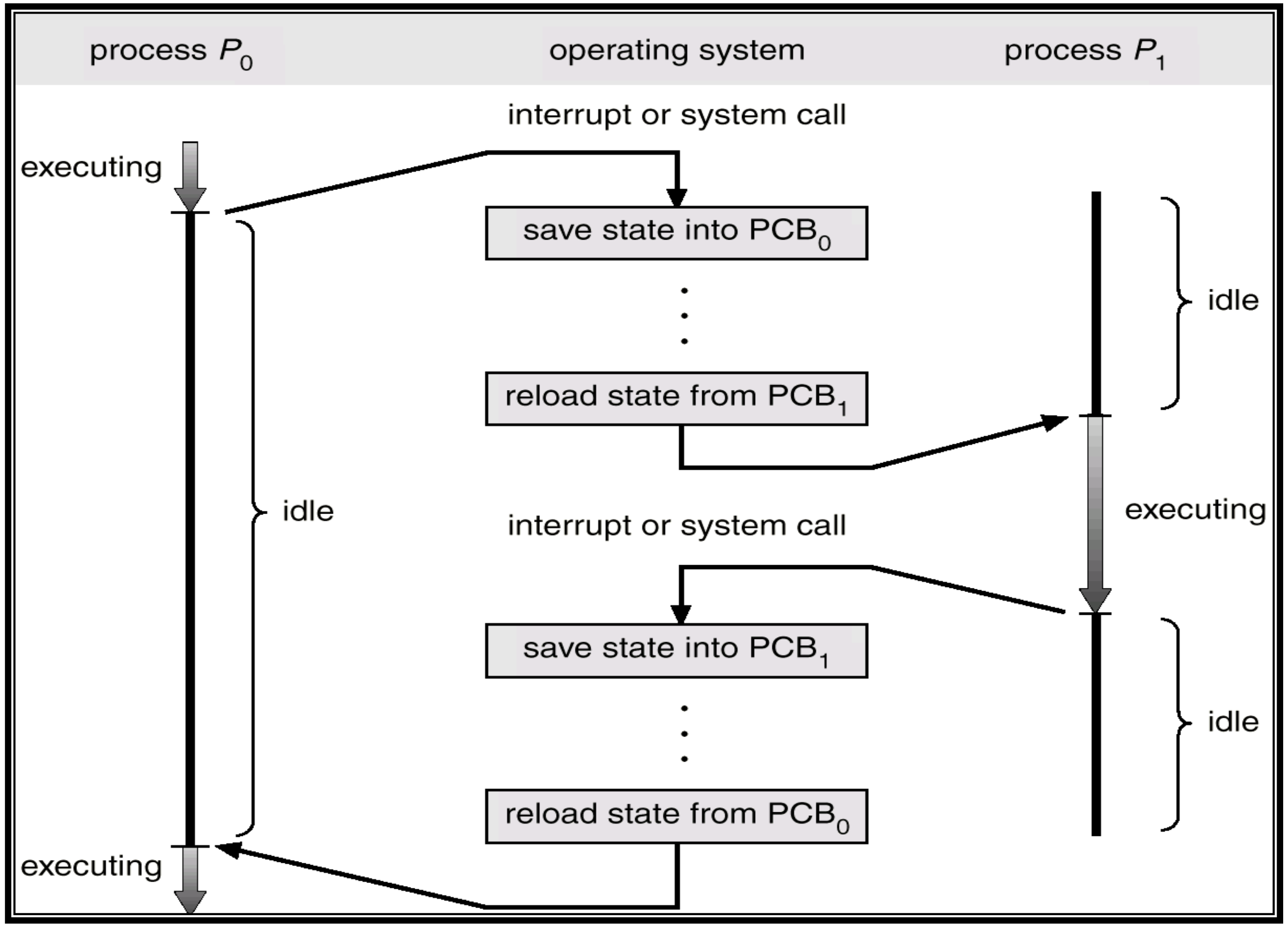


Process Control Block (PCB)

- The process control block in the Linux operating system is represented by the C structure task struct, which is found in the <linux/sched.h> include file in the kernel source-code directory.

```
long state; /* state of the process */
struct sched_entity se; /* 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 */
```

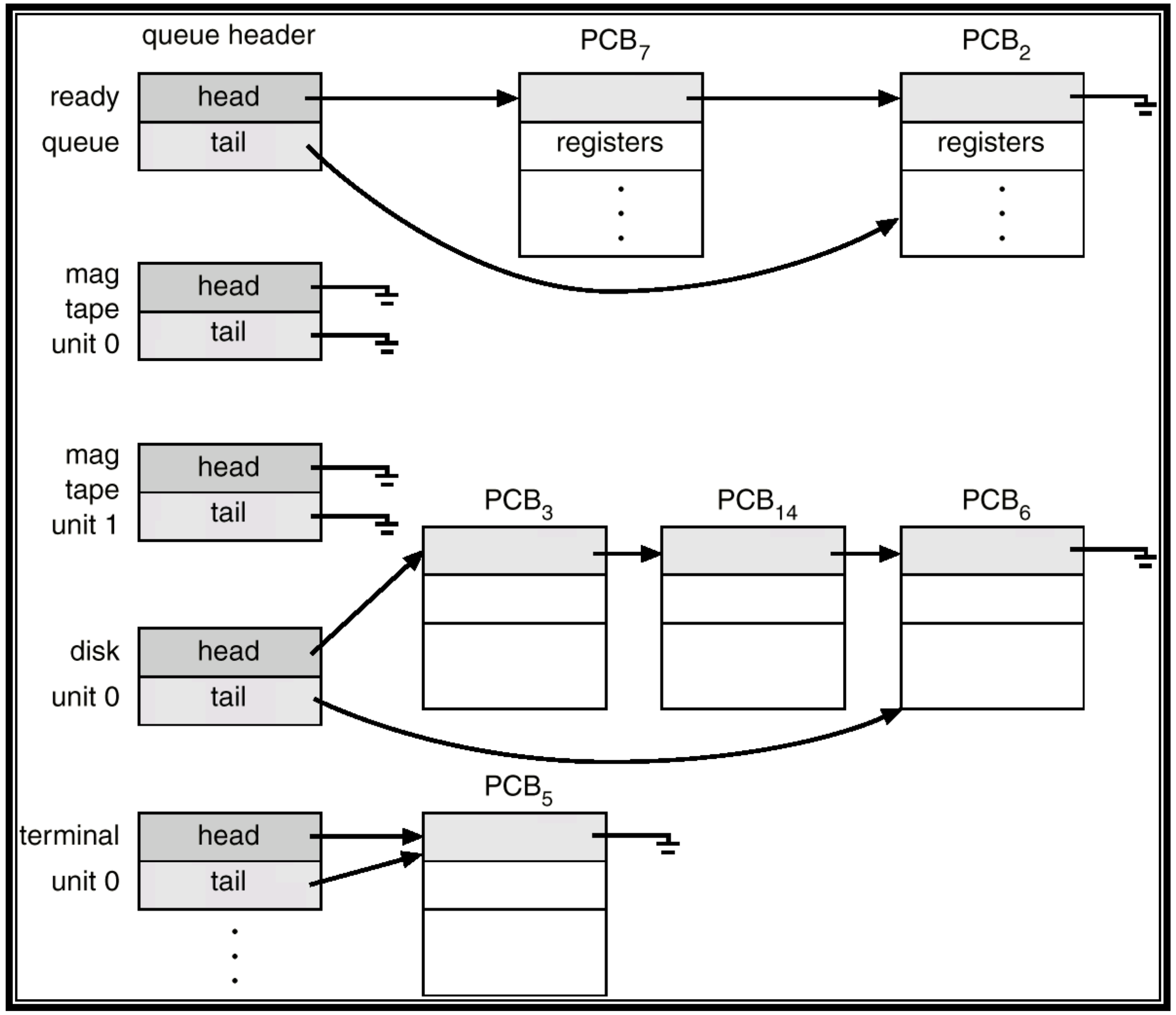
CPU Switch From Process to Process



Process Scheduling Queues

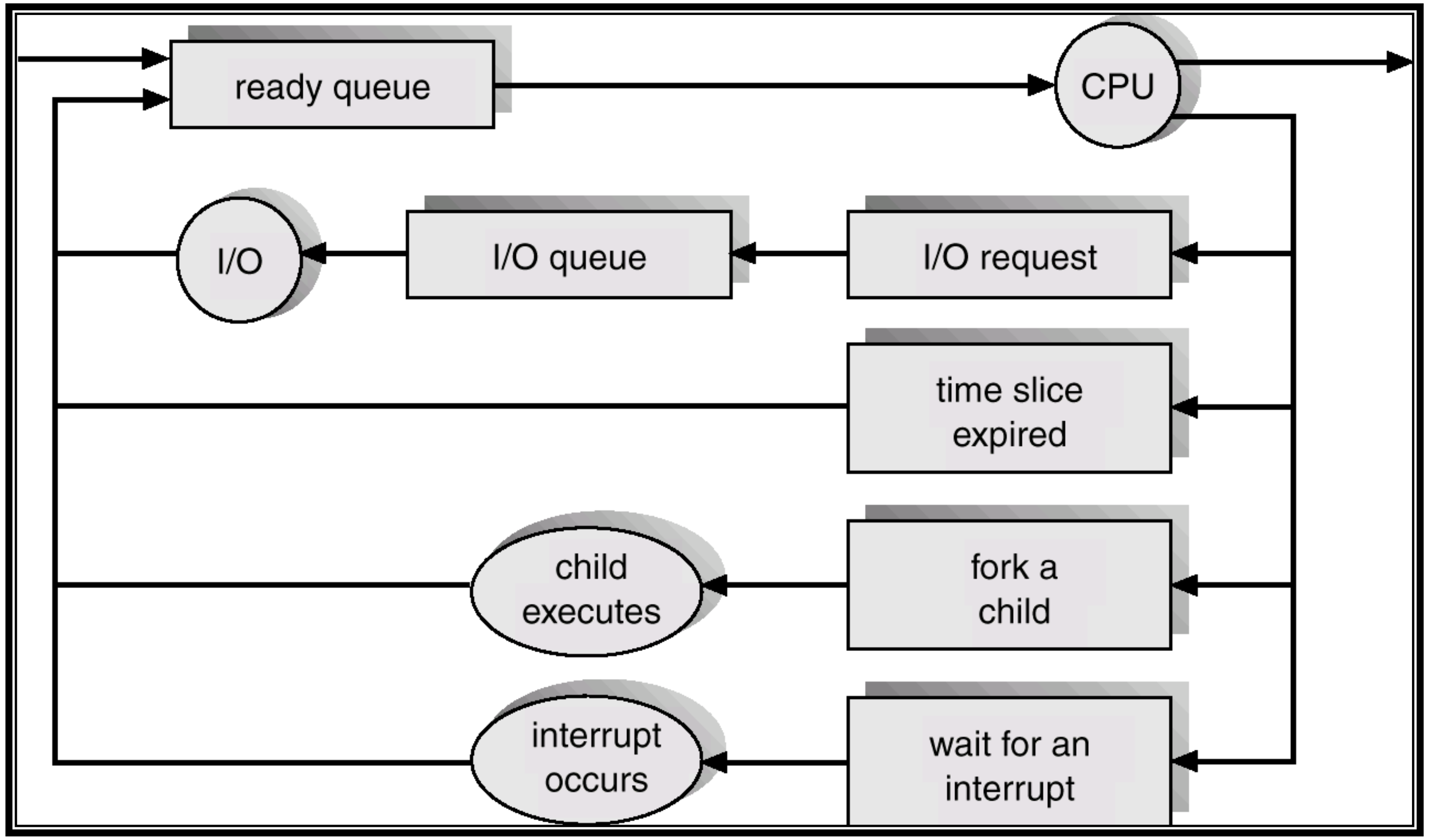
- Scheduling is to decide which process to execute and when
- The objective of multi-program
    - To have some process running at all times.
- **Timesharing**: Switch the CPU frequently that users can interact can interact with the program while it is running.
- If there are many processes, scheduling is used to decide which process to execute and when.
- Scheduling queues:
    - 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.
        - Each device has its own queue.
- Process migrates between the various queues during its life time.

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling

- Queuing Diagram



Representation of Process Scheduling

- A new process is initially put in the ready queue
- Once a process is allocated CPU, the following events may occur
  - A process could issue an I/O request
  - A process could create a new process
  - The process could be removed forcibly from CPU, as a result of an interrupt.
- When process terminates, it is removed from all queues. PCB and its other resources are de-allocated.

Process Schedulers

- A process migrates between the various scheduling queues throughout its lifetime.
- The OS must select a process from the different process queues in some fashion. The selection process is carried out by a scheduler.
- In a batch system the processes are spooled to mass-storage device.
- **Long-term (LT) scheduler (or job scheduler)** – selects which processes should be brought into the ready queue.
- The LT scheduler controls degree of multiprogramming (i.e., the number of processes active in the system)
- DoM is stable: average rate of process creation == average departure rate of processes.

Process Schedulers

- The LT scheduler should make a careful selection.
- Most processes are either I/O bound or CPU bound.
  - **I/O bound process** spends more time doing I/O than it spends doing computation.
  - **CPU bound process** spends most of the time doing computation.
- The LT scheduler should select a good mix of I/O-bound and CPU-bound processes.
- Example:
  - If all the processes are I/O bound, the ready queue will be empty
  - If all the processes are CPU bound, the I/O queue will be empty, the devices will go unutilized and the system will be imbalanced.

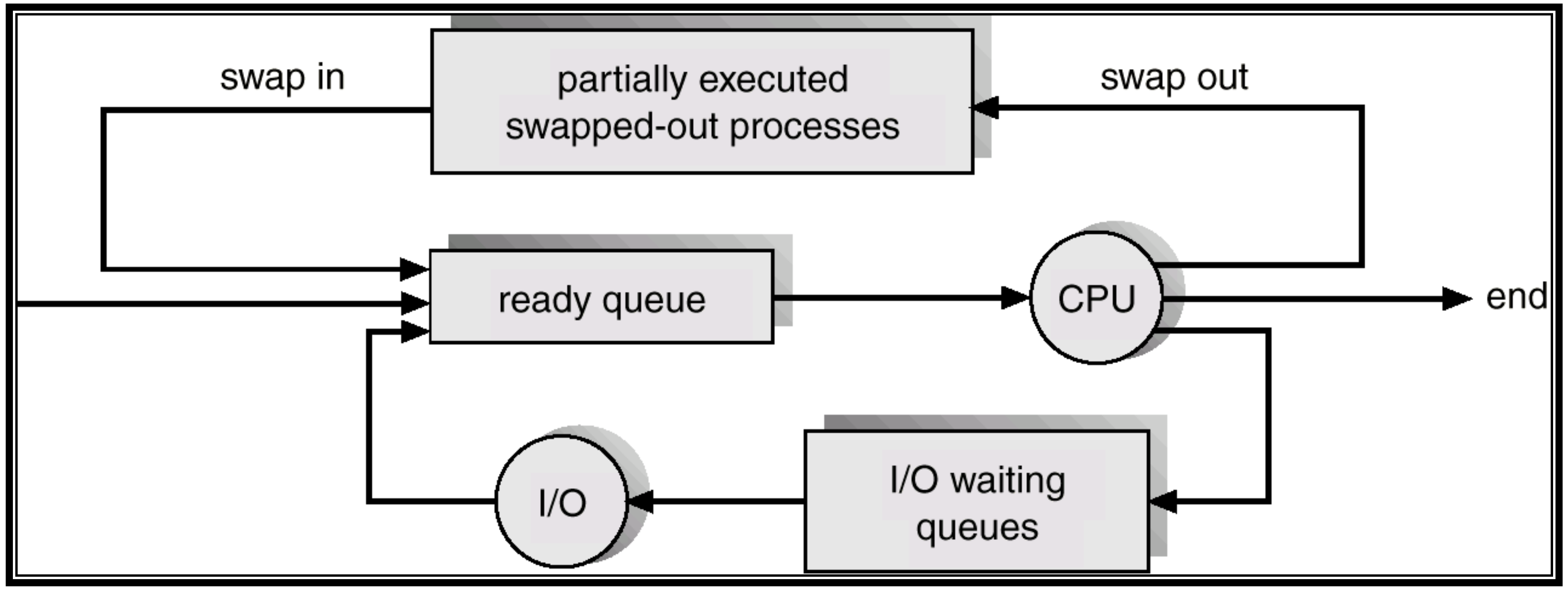
Process Schedulers

- **Short-term (ST) scheduler**
  - selects which process should be executed next and allocates CPU.
  - It is executed at least once every 100 ms.
  - If 10 ms is used for selection, then 9 % of CPU is used (or wasted)
- The long-term scheduler executes less frequently.

Process Schedulers

- Some OSs introduced a **Medium-Term** scheduler using swapping.
  - Advantageous to remove the processes from the memory and reduce the multiprogramming.
- **Swapping:** removal of process from main memory to disk to improve the performance. At some later time, the process can be reintroduced into main memory and its execution can be continued when it left off.
- Swapping improves the process mix (I/O and CPU), when main memory is unavailable.

Addition of Medium Term Scheduling



Context Switch

- Context switch is a task of switching the CPU to another process by saving the state of old process and loading the saved state for the new process
- Context of old process is saved in PCB and loads the saved context of old process.
- Context-switch time is **overhead**; the system does no useful work while switching.
- New structures threads were incorporated.
- Time dependent on hardware support.
  - 1 to 1000 microseconds

Operations on Processes

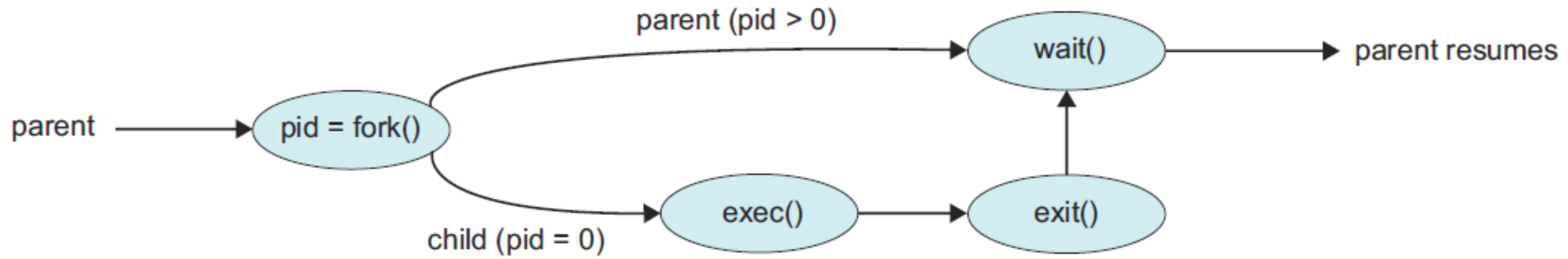
- Process Creation
- Process Termination

Process Creation

- A system call is used to create process.
  - Assigns unique id
  - Space
  - PCB is initialized.
- The creating process is called parent process.
- Parent process creates children processes, which, in turn create other processes, forming a tree of processes.
  - In UNIX **pagedaemon, swapper,** and **init** children are root process. Users are children of **init** process.
- A process needs certain resources to accomplish its task.
  - CPU time, memory, files, I/O devices.

Process Creation

- When a process creates a new process,
  - Resource sharing possibilities.
    - Parent and children share all resources.
    - Children share subset of parent's resources.
    - Parent and child share no resources.
- Execution possibilities
  - Parent and children execute concurrently.
  - Parent waits until children terminate.
- Address space
  - Child duplicate of parent.
  - Child has a program loaded into it.



Process Creation

- 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.
  - The new process is a copy of the original process.
  - The exec system call is used after a fork by one of the two processes to replace the process memory space with a new program.
- WINDOWS NT supports both models:
  - Parent address space can be duplicated or
  - parent can specify the name of a program for the OS to load into the address space of the new process.

UNIX: fork() system call

- fork() is used to create processes. It takes no arguments and returns a process ID.
- fork() creates a new process which becomes the child process of the caller.
- After a new process is created, both processes will execute the next instruction following the fork() system call.
- The checking the return value, we have to distinguish the parent from the child.
- fork()
    - If returns a negative value, the creation is unsuccessful.
    - Returns 0 to the newly created child process.
    - Returns positive value to the parent.
- Process ID is of type pit_t defined in sys/types.h
- getpid() can be used to retrieve the process ID.
- The new process consists of a copy of address space of the original process.

UNIX: fork() system call

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#define  MAX_COUNT 200
#define  BUF_SIZE 100
void main(void)
    {
         pid_t pid;
         int i;
      char buf[BUF_SIZE];
         fork();
       pid=getpid();
         for(i=1; i<=MAX_COUNT;i++)
       {
             sprintf(buf,”This line is from pid %d, value=%d\n”,pid,i);
         write(1,buf,strlen(buf));
       }
}
```

UNIX: fork() system call

- If the fork() is executed successfully, Unix will
  - Make two identical copies of address spaces; one for the parent and one for the child.
  - Both processes start their execution at the next statement after the fork().

Parent	**Child**
main()	main()
{	{
fork();	fork();
pid=…;	pid=…
}	}

UNIX: fork() system call

```
#include <stdio.h>
#include <sys/types.h>
#define  MAX_COUNT 200
void  ChildProcess(void);
void ParentProcess(void);
#define  BUF_SIZE 100
void main(void)
     {
         pid_t pid;
         pid=fork();
        if (pid==0)
                         ChildProcess();
      else
                         ParentProcess();
}
}
```

```
Void ChildProcess(void)
     {
        int i;
        for(i=1;i<=MAX_COUNT;i++)
        {
           printf(buf,"This line is from child, value=%d\n",i);
        Printf(" *** Child Process is done ***\n");
        }
}

Void ParentProcess(void)
     {
        int i;
        for(i=1;i<=MAX_COUNT;i++)
        {
         printf(buf,"This line is from parent, value=%d\n",i);
        printf(" *** Parent Process is done ***\n");
        }
}
```

UNIX: fork() system call

Parent

PID=3456

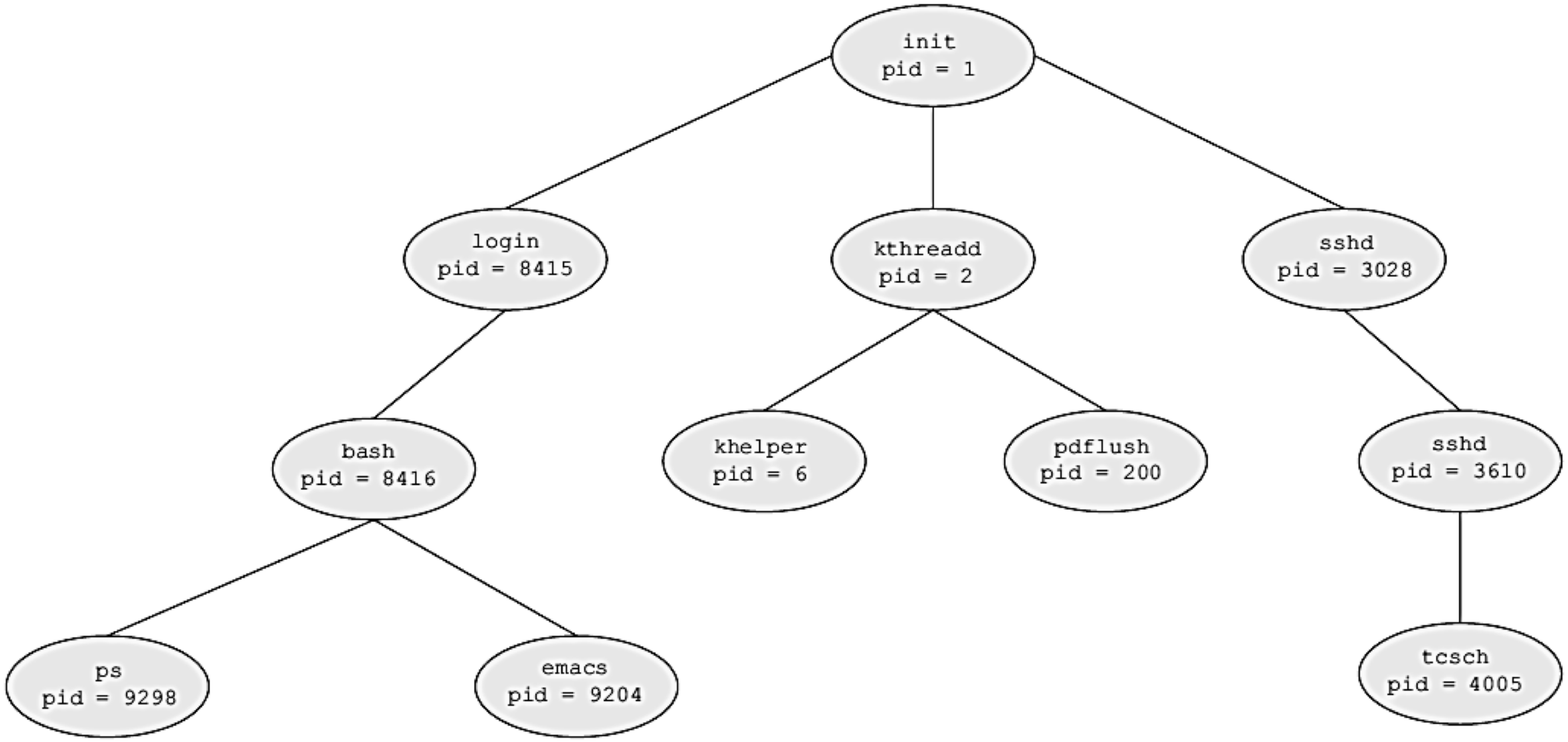
```
void main(void)
    {
      pid=fork();
     if (pid==0)
                    ChildProcess();
    else
                    ParentProcess();
  }
}
Void ChildProcess(void)
    {
      }

Void ParentProcess(void)
    {

}
```

Child

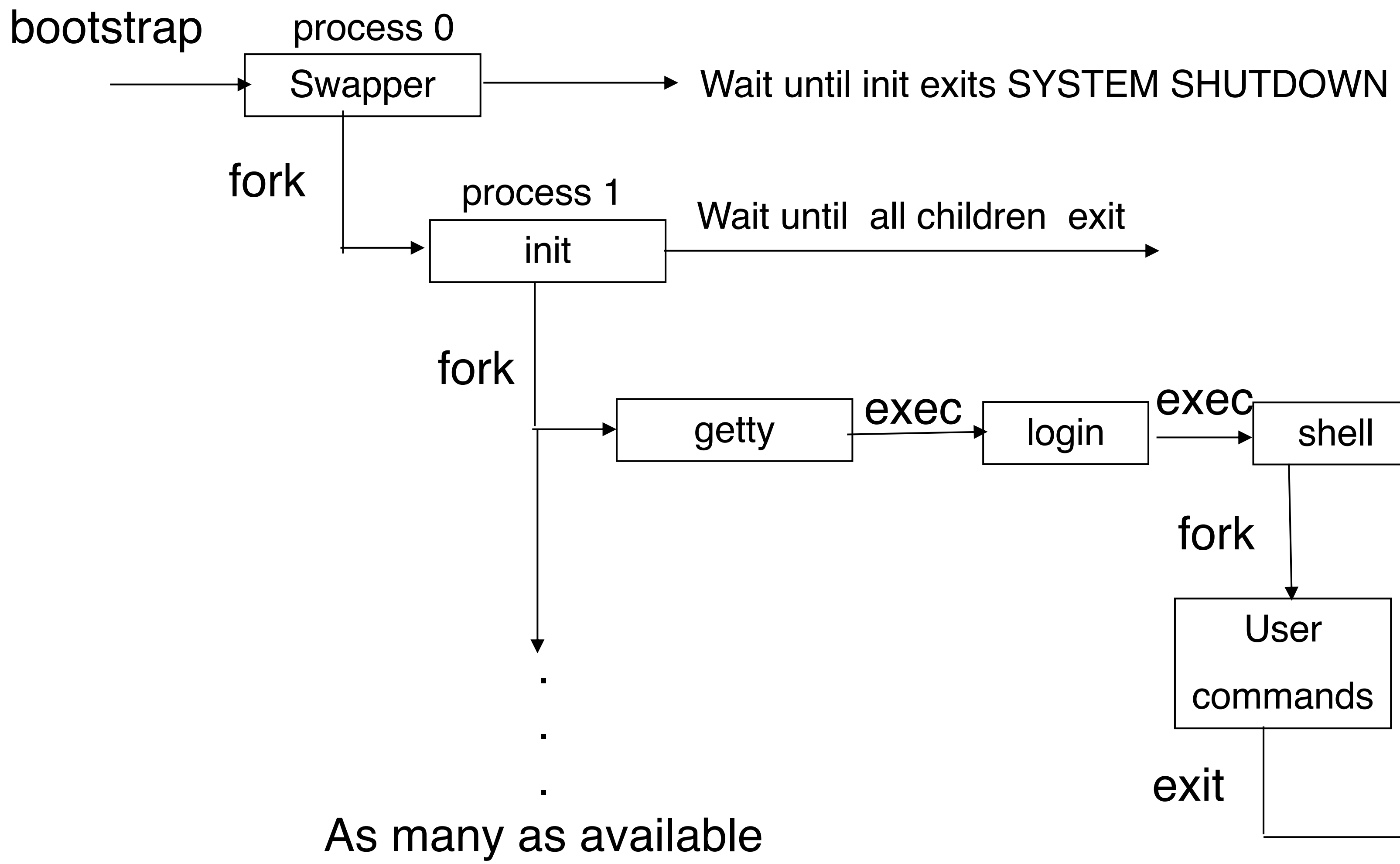
PID=0

```
void main(void)
    {
      pid=fork();
     if (pid==0)
                    ChildProcess();
    else
                    ParentProcess();
  }
}
Void ChildProcess(void)
    {
      }

Void ParentProcess(void)
    {

}
```

Processes Tree on a UNIX System



UNIX system initialization



Process Termination

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

- Parent may terminate the execution of children processes (**abort**).
  - Child has exceeded allocated resources.
  - Task assigned to child is no longer required.
  - Parent is exiting.
  - Many operating system does not allow child to continue if its parent terminates leading to phenomenon of **Cascading termination**.

Process Termination

- A process that has terminated, but whose parent has not yet called wait(), is known as a **zombie** process.
  - When a process terminates, its resources are deallocated by the operating system. However, its entry in the process table must remain there until the parent calls wait(), because the process table contains the process's exit status.

```
pid_t pid;
int status;

pid = wait(&status);
```

- If a parent did not invoke wait() and instead terminated, thereby leaving its child processes as **orphans**.
- UNIX address this scenario by assigning the *init* process as the new parent to orphan processes.

Cooperating Processes

- The processes can be independent or cooperating processes.
- *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
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up: Break into several subtasks and run in parallel
  - Modularity: Constructing the system in modular fashion.
  - Convenience: User will have many tasks to work in parallel
    - Editing, compiling, printing

THANK YOU