Process Management
&
Scheduling Policies

Shamjith

Concept

- Process – a program in execution; process execution must progress in sequential fashion
- A process includes:
  - program counter
  - stack
  - data section
- Program is Text/Instructions without any dynamic behavior
- Process is also called a job, or task

Program (passive, on disk)

```
int global1 = 0;
int global2 = 0;

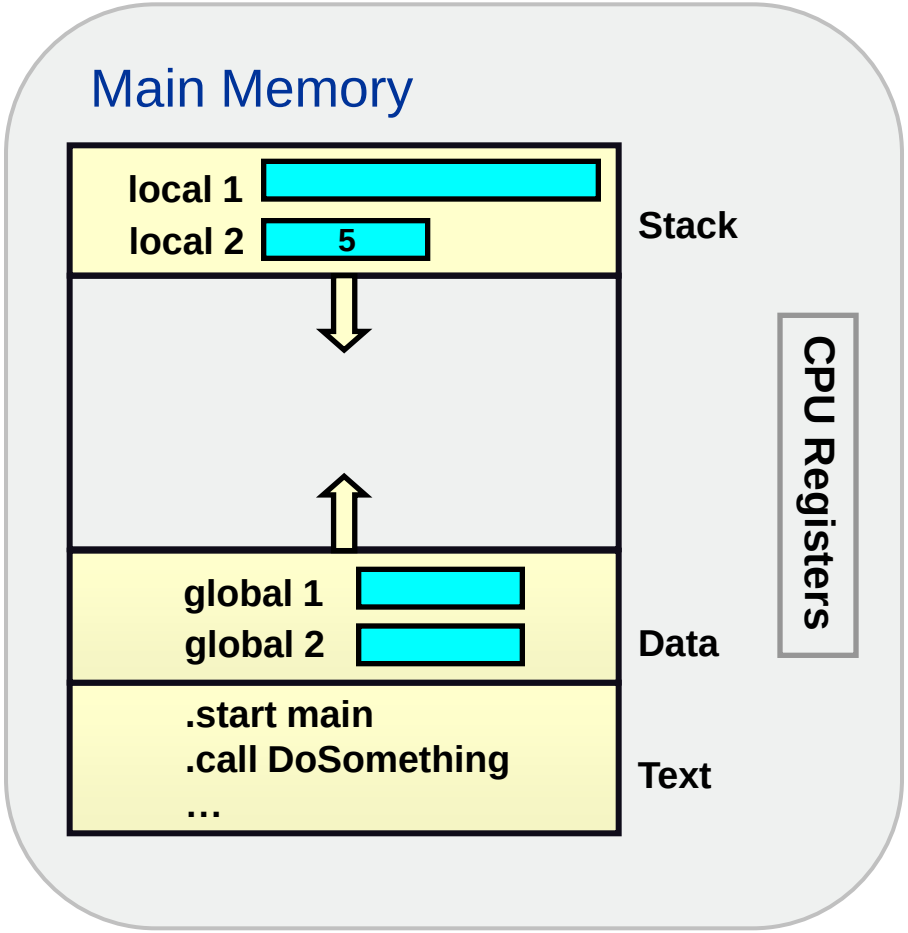
void DoSomething()
{
   int local2 = 5;

   local2 = local2 + 1;
   ...
 }

int main()
{
   char local1[10];

   DoSomething();
                ...
}
```

Process (active, in memory)



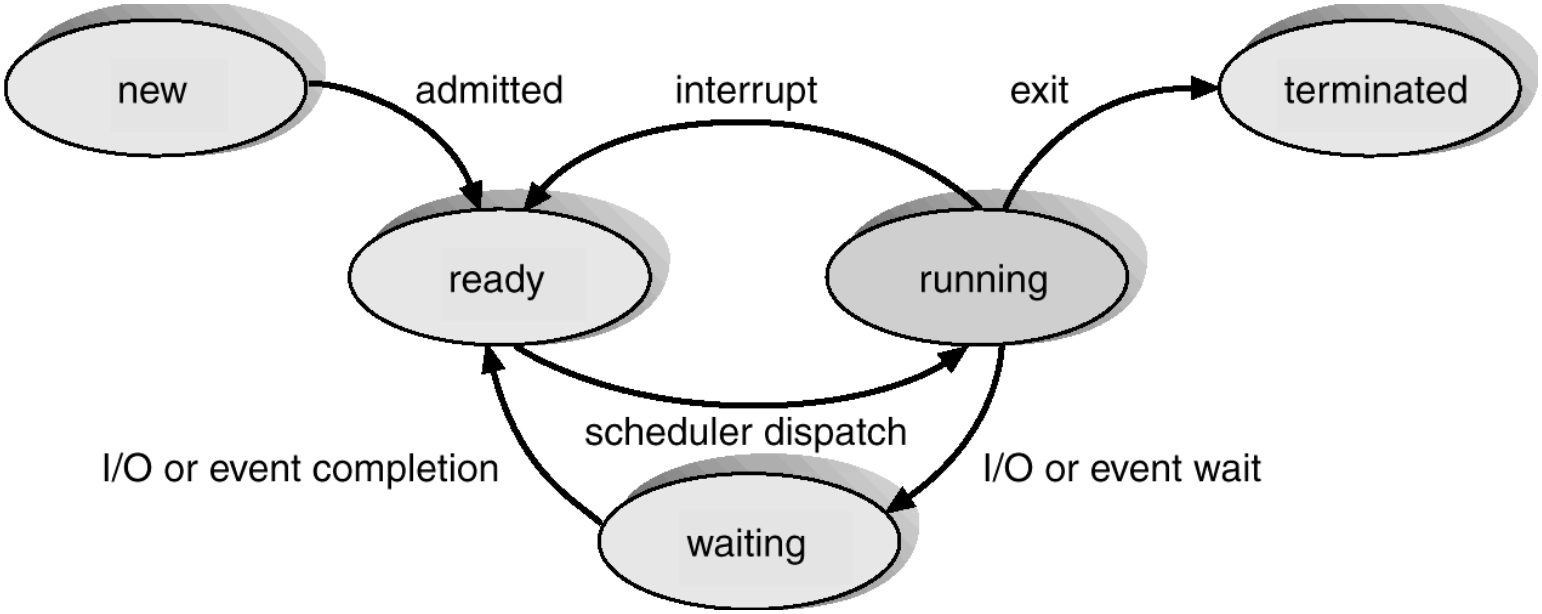
A program becomes a process when an executable file is loaded into memory.

- An instance of a Program/Application in Execution
  - Own address space
  - Process control block
  - Process ID
  - Each of these resources is exclusive to the process
    - the code for the running program
    - the data for the running program
    - and stack pointer (SP)
    - the program counter (PC)
    - a set of processor registers – general purpose and status
    - a set of system resources
      - files, network connections, privileges, …

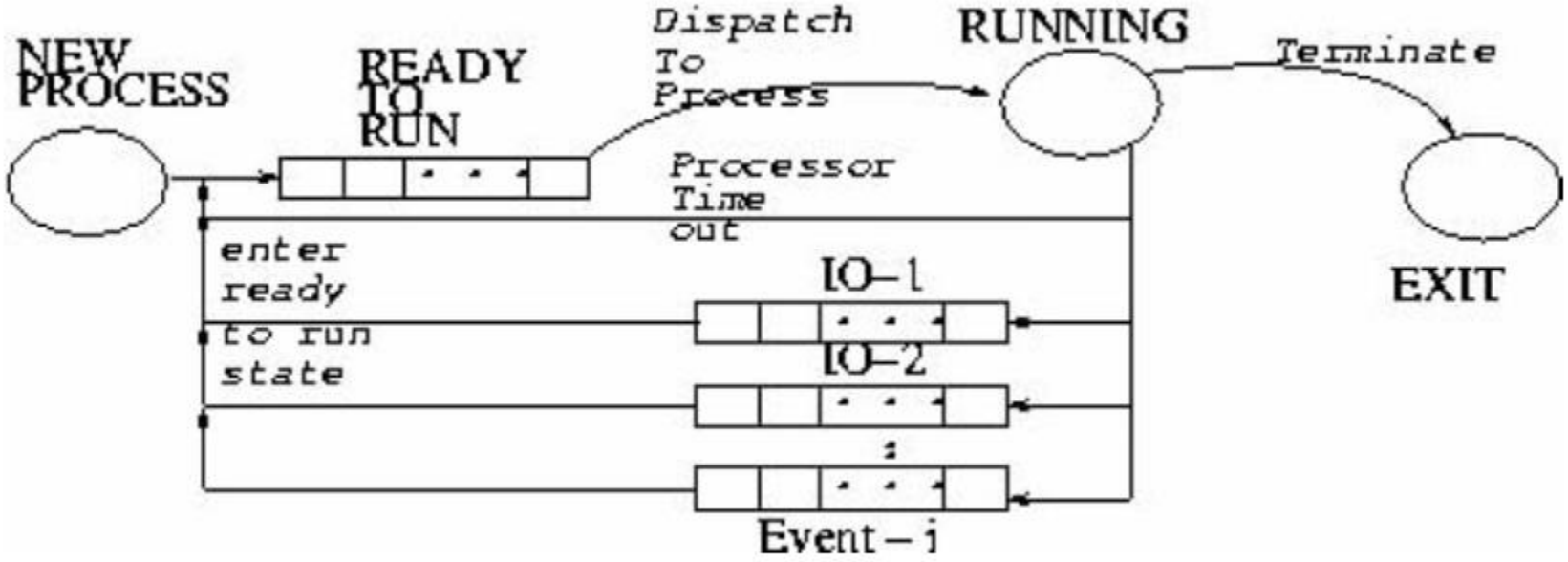
Components of Process

S.N.	Component & Description
1	**Object Program** Code to be executed.
2	**Data** Data to be used for executing the program.
3	**Resources** While executing the program, it may require some resources.
4	**Status** Verifies the status of the process execution. A process can run to completion only when all requested resources have been allocated to the process. Two or more processes could be executing the same program, each using their own data and resources.

Process State

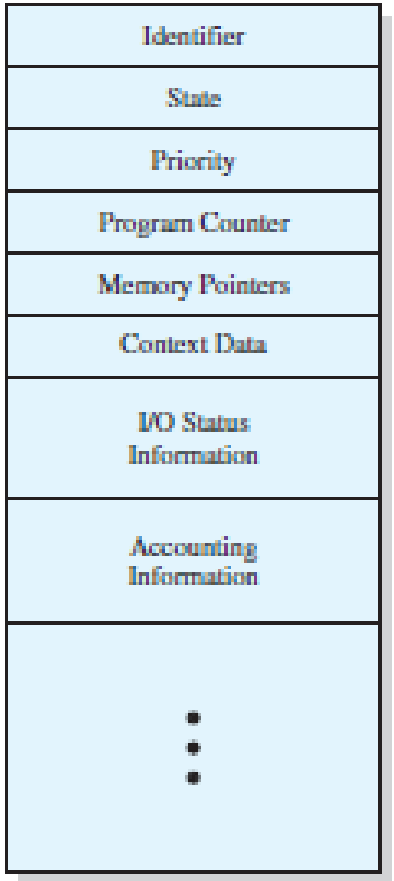


Queue based model



Process Control Block (PCB)

- Process can be uniquely characterized by a number of elements
  - Identifier
  - State
  - Priority
  - Program counter
  - Memory pointers:
    - pointers to code and data
    - memory blocks shared with other processes
  - Context data
  - I/O status information
    - outstanding I/O requests
    - I/O devices assigned to the process
    - list of files in use by processor …
  - Accounting information



top

top - currently happening on the system

%> top

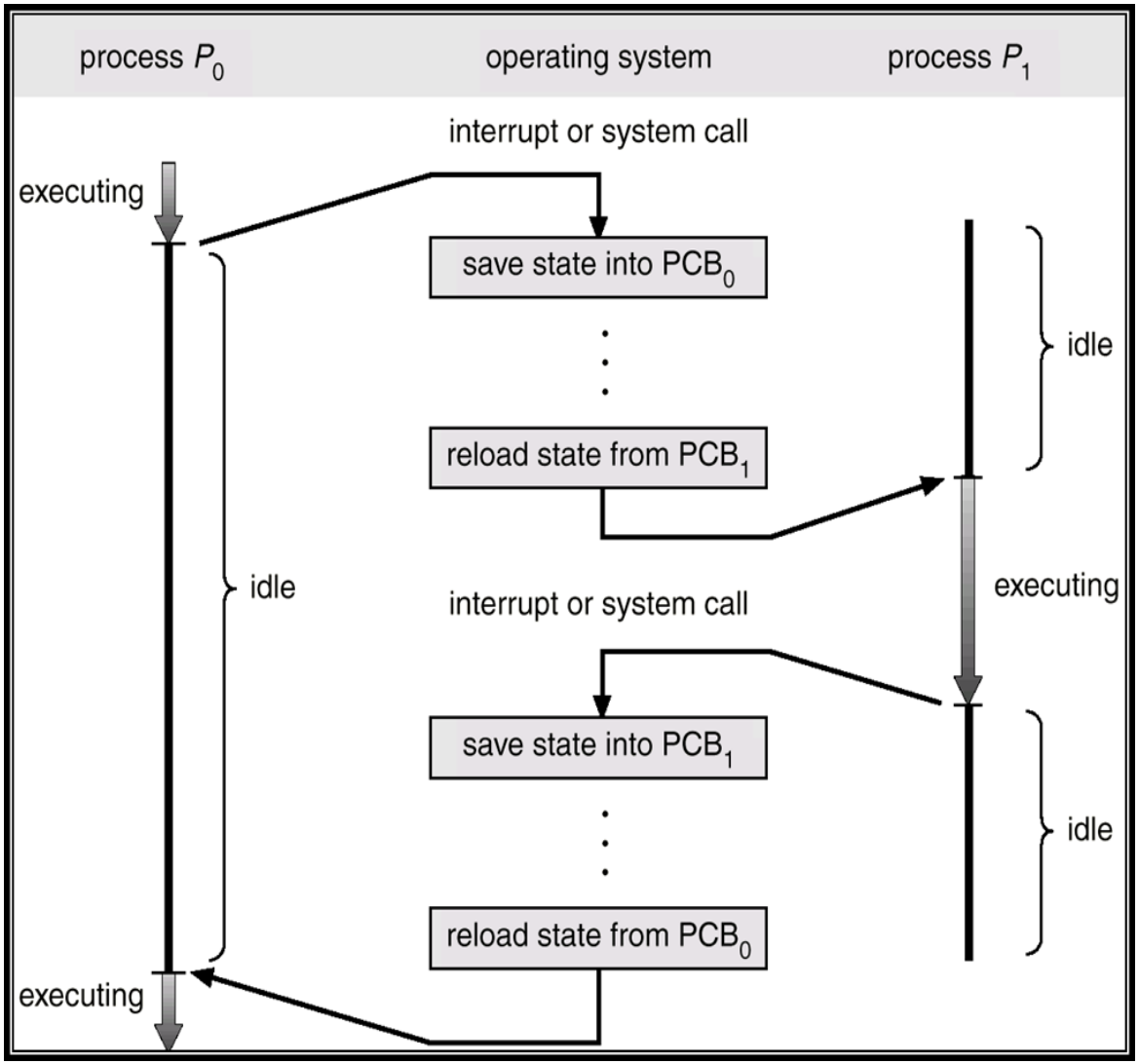
Tasks: 174 total, 3 running, 171 sleeping, 0 stopped
KiB Mem: 4050604 total, 3114428 used, 936176 free
Kib Swap: 2104476 total, 18132 used, 2086344 free

PID	USER	%CPU	%MEM	COMMAND
6978	ryan	3.0	21.2	firefox
11	root	0.3	0.0	rcu_preempt
6601	ryan	2.0	2.4	kwin

...

CPU Switch From Process to Process

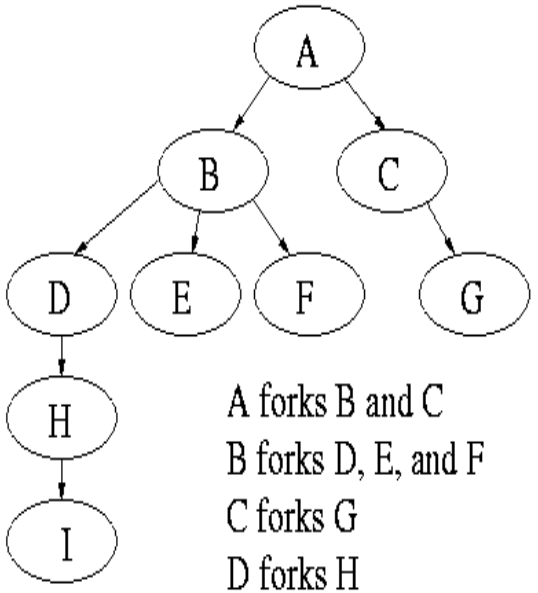
- Context Switch



Process Creation

- Modern general purpose operating systems permit a user to create and destroy processes.
- In Unix this is done by the fork system call and terminates using exit system call.

- **After a fork, both parent and child keep running, and each can fork off other processes.**
- **A process tree results. The root of the tree is a special process created by the OS during startup.**



PID and Parentage

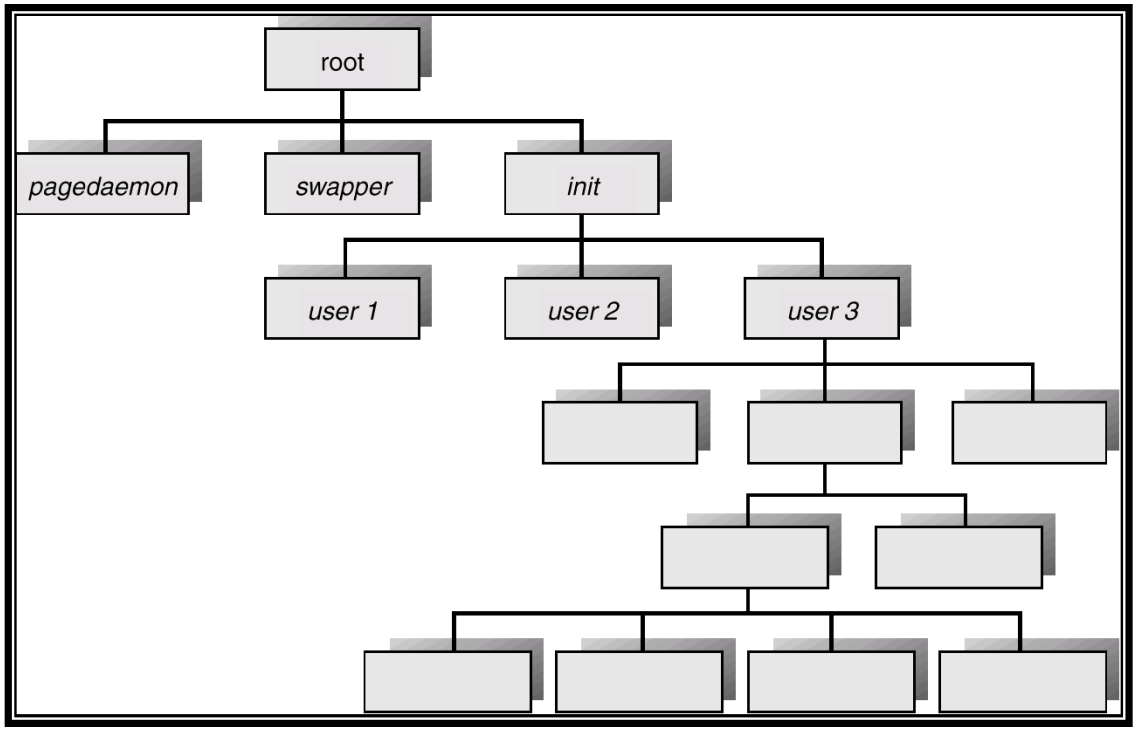
- A process ID or *pid* is a positive integer that uniquely identifies a running process, and is stored in a variable of type *pid_t*.

- You can get the process pid or parent's pid

```
#include <sys/types>
main()
   {
   pid_t pid, ppid;
   printf( "My  PID is:%d\n\n",(pid = getpid()) );
   printf( "Par PID is:%d\n\n",(ppid = getppid())
   );
   }
```

Process Creation

- Processes are identified using unique process identifier (pid) : integer number
- each process has one parent but zero, one, two, or more children



Process Creation (Cont…)

- Resource sharing
  - Parent and children share all resources.
  - Children share subset of parent's resources.
  - Parent and child share no resources.
- When a process is created, initialization data (input) may be passed along by the parent process to the child process (default)
- Execution : 2 possibilities
  - Parent and children execute concurrently.
  - Parent waits until some or all its children terminate.

Process Creation (Cont…)

- Address space : two possibilities
  - Child is a duplicate of parent. (program and data same)
  - Child has a new 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.
  - **ps** command can be used to list processes

Fork()

- #include <sys/types.h>
  #include <unistd.h>
  pid_t fork(void);
  - System call **fork()** is used to create processes
  - **fork()** creates a ***new*** process, which becomes the *child* process of the caller
  - Creates a child process by making a copy of the parent process --- an exact duplicate.
  - Implicitly specifies code, registers, stack, data, files
  - Both parent and child continue to execute following the ***fork()*** system call
  - Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of **fork()**

Process IDs

- pid = fork();
- In the child: pid == 0;

(**fork()** returns a zero to the newly created child process)

- In the parent: pid == the process ID of the child.

(**fork()** returns a positive value, the ***process ID*** of the child process, to the parent)

- If **fork()** returns a negative value, the creation of a child process was unsuccessful
- A program almost always uses this pid difference to do different things in the parent and child.

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

  int main()
  {
        pid_t pid;       /* could be int */
        int i;
        pid = fork();
        if( pid > 0 )
        {
        /* parent */
        for( i=0; i < 10; i++ )
                  printf("\t\t\tPARENT %d\n", i);
        }
        else
        {
        /* child */
       for( i=0; I < 10; i++ )
                  printf( "CHILD %d\n", i );
        }
    return 0;
  }
  ```
  ```

```
#include <unistd.h>
#include <sys/wait.h>
int main()
{
pid_t pid;
int status, died;
switch(pid=fork())
{
// fail to create child
case -1:
printf("can't fork\n“”);
exit(-1);
// this is the code the child runs
case 0:
sleep(2);
exit(3);
// this is the code the parent runs
default:
died= wait(&status);
}
}
```

Wait()

Used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed

Synopsis

– pid_t wait(int *status);

– pid_t waitpid(pid_t pid, int *status, int options);

Exit()

Causes normal process termination

Synopsis

– void exit(int status);

exec

- 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.
- When you use the shell to run a command (**ls**, say) then at some point the shell will execute a **fork()** call to get a new process running. Having done that, how does the shell then get **ls** to run in the child process instead of the duplicate copy of the shell
- The solution in this case is to use an **exec()** system call
- Family of functions for replacing process's program with the one inside the `exec()` call

Example

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv [])
{
int pid;
/* fork another process */
pid = fork();
if (pid < 0) { /* error occurred */
fprintf(stderr, "Fork Failed");
exit(-1);
}
else if (pid == 0) { /* child process */
execlp("/bin/ls","ls",NULL);
printf("CHILD PROC %d\n", getpid());
}
else { /* parent process */
/* parent will wait for the child to complete */
wait(NULL);
printf("Child Complete");
printf("PARENT PROC %d\n", getpid());
exit(0);
}
}
```

- Family of functions for replacing process's program with the one inside the `exec()` call.
  e.g.

```
#include <unistd.h>
int execlp(char *file, char *arg0,
           char *arg1, ..., (char *)0);

execlp("/bin/sort", "sort", "-n",
                    "foobar", (char *)0);
```

```
Syntax:
Parameter1 = full path name
Following Parameters = options that are given on command line
Terminated by Null
```

E.g.

```
execl("/bin/ls", "ls", "-l", 0);
```

Exec() Family

- `int execl( const char *path, const char *arg, ... );`
- `int execlp( const char *file, const char *arg, ... );`
- `int execle( const char *path, const char *arg, ..., char *const envp[] );`
- `int execv( const char *path, char *const argv[] );`
- `int execvp( const char *file, char *const argv[] );`
- `int execve( const char *filename, char *const argv [], char *const envp[] );`

The argument vector and environment can be accessed by the called program's main function, when it is defined as **int main(int argc, char *argv[], char *envp[])**.

Reasons for Process Creation

- Events that cause processes to be created
  - System initialization.
  - Execution of a process creation system call by a running process
  - A user request to create a new process.
  - Initiation of a batch job.
- When the OS creates a process at the explicit request of another process, the action is referred to as process spawning

Process Termination

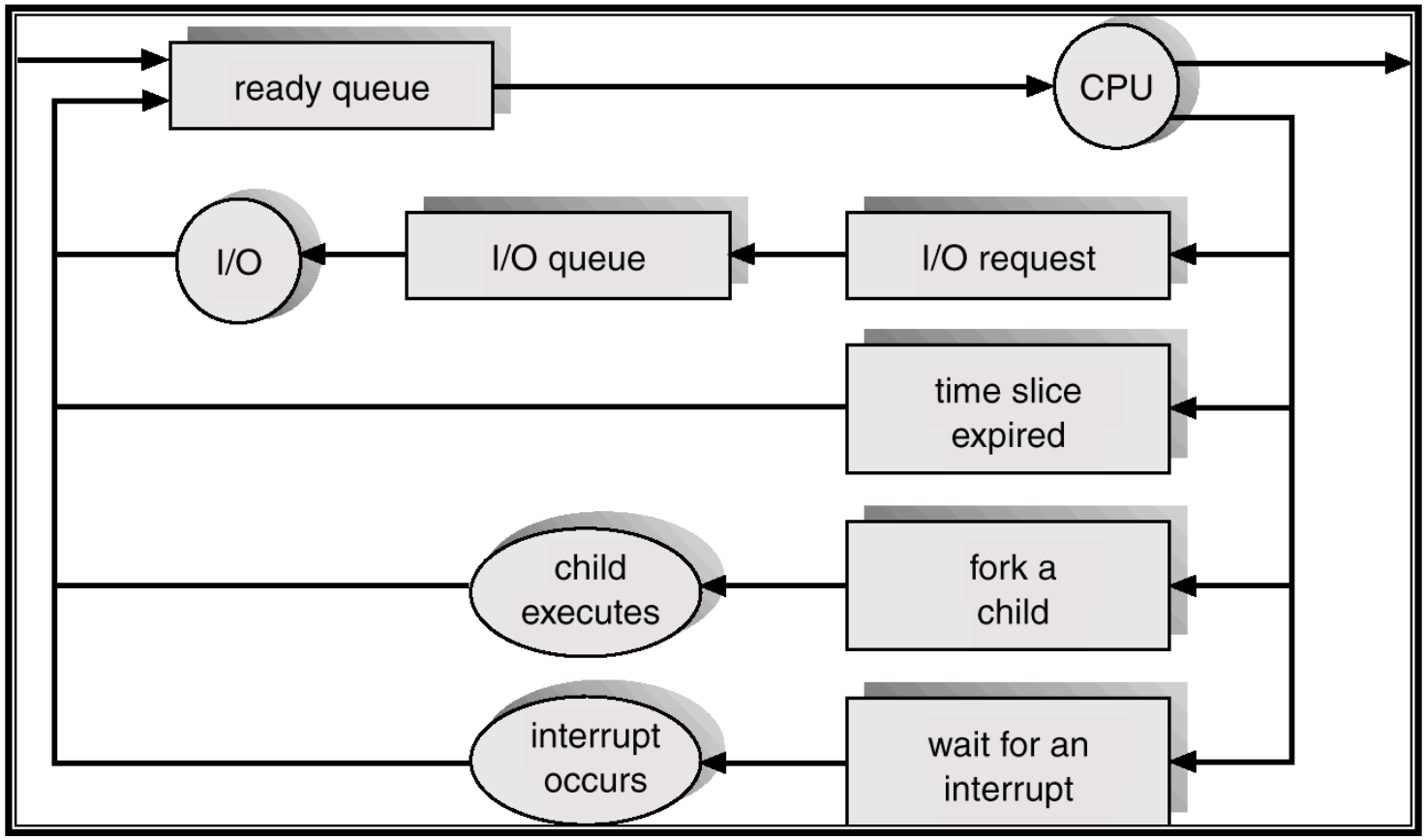
- Process executes last statement and asks the operating system to decide it (**exit**).
  - Output data from child to parent (via **wait**).
  - Process' resources are deallocated by operating system.
- Parent may terminate execution of children processes (**abort**).
  - Child has exceeded allocated resources.
  - Task assigned to child is no longer required.
  - Parent is exiting.
    - Operating system does not allow child to continue if its parent terminates.
    - Cascading termination.

Process Management

- Complete activity required for managing a process throughout its lifecycle
  - Allocate resources to processes
  - Enable processes to share and exchange information
  - Protect the resources of each process from other processes
  - Enable synchronization among processes.
  - Maintain a data structure for each process
  - Execution and Control of processes

Process 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.
- Process migration between the various queues.


ready queue
CPU
I/O
I/O queue
I/O request
time slice expired
child executes
fork a child
interrupt occurs
wait for an interrupt

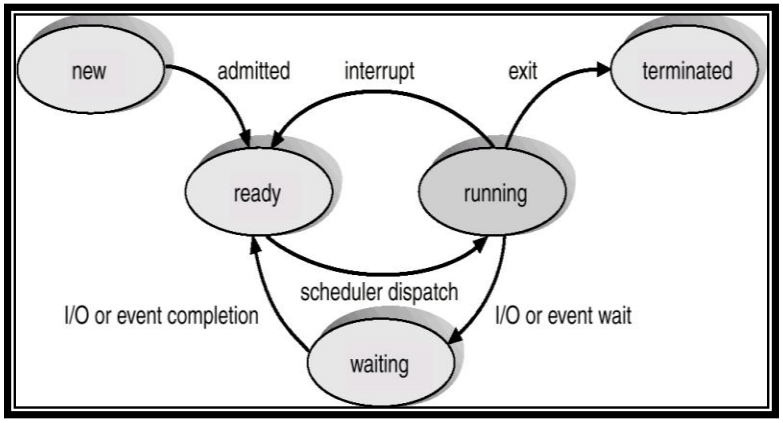
Schedulers

- Types:
  - Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue.
  - Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates CPU.
- Short term scheduler is invoked very frequently (milliseconds) $\Rightarrow$ must be fast
- The long-term scheduler controls the degree of multiprogramming (the number of processes in memory).
  - invoked only when a process leaves the system (seconds, minutes) can be slow

Process Scheduling

CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
  - Short term Scheduler or CPU scheduler
- CPU scheduling decisions may take place when a process:
  - 1.Switches from running to waiting state
  - 2.Switches from running to ready state
  - 3.Switches from waiting to ready
  - 4.Terminates



Preemptive and Non-preemptive scheduling

- Non-preemptive scheduling
  - A new process is selected to run either
    - when a process terminates or
    - when an explicit system request causes a wait state
      - (e.g., I/O or wait for child)
- Preemptive scheduling
  - New process selected to run also when
    - An interrupt occurs
    - When new processes become ready

Scheduling Criteria

- CPU utilization – keep the CPU as busy as possible
- Throughput – # of processes that complete their execution per time unit
- Turnaround time – amount of time to execute a particular process
- Waiting time – amount of time a process has been waiting in the ready queue
- Response time – amount of time it takes from when a request was submitted until the first response is produced, **not** output (for time-sharing environment)

Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

Scheduling Algorithms

- First-come, First-Served (FCFS)
  - Complete the jobs in order of arrival
- Shortest Job First (SJF)
  - Complete the job with shortest next CPU requirement (e.g., burst)
  - Provably optimal w.r.t. average waiting time
- Priority
  - Processes have a priority number
  - Allocate CPU to process with highest priority
- Round-Robin (RR)
  - Each process gets a small unit of time on CPU (time quantum or time slice)
  - For now, assume a FIFO queue of processes

FCFS: First – Come First - Serve

- Implement with a FIFO ready queue
- Major disadvantage can be long wait times
- Example:

Process	Burst time
P1	7
P2	3
P3	4
P4	6

- Draw Gantt chart

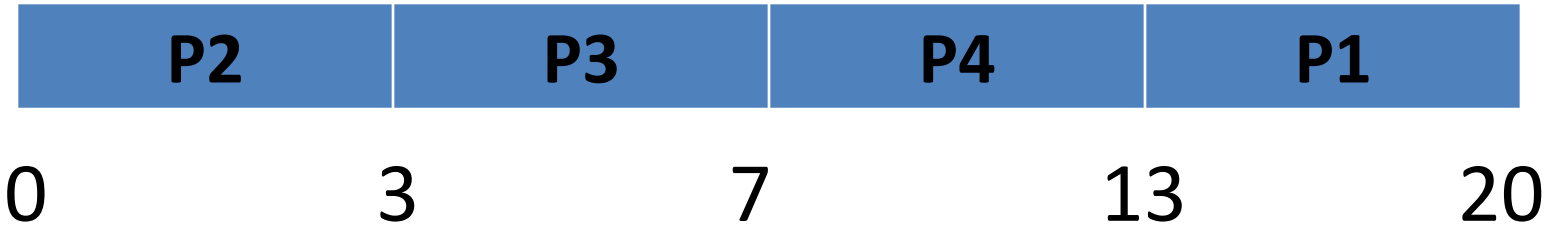
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
- Two schemes:
  - nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst
  - preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is known as the Shortest-Remaining-Time-First (SRTF)
- SJF is optimal – gives minimum average waiting time for a given set of processes

SJF: Shortest Job First

- The job with the shortest next CPU burst time is selected

Process	Burst time
P1	7
P2	3
P3	4
P4	6



Process	Waiting Time
P1	13
P2	0
P3	3
P4	7
Average	5.75

SJF…

- Shortest average wait time
- It’s unfair !!
- Continuous stream of short jobs will starve long job
- Need to know the execution time of a process

  May have an estimate, to predict next CPU burst
- Jobs are organized in order of estimated completion time

  “Recent history is a good indicator of the near future”

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem ≡ Starvation – low priority processes may never execute
- Solution ≡ Aging – as time progresses increase the priority of the process

Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.
- Performance
  - *q* large $\Rightarrow$ FIFO
  - *q* small $\Rightarrow$ *q* must be large with respect to context switch, otherwise overhead is too high

Problem

Process	Arrival Time	Burst time	Priority
P1	10	10	2
P2	15	4	1
P3	20	12	4
P4	5	8	3
P5	25	6	5

- FCFS, Round Robin, SJF (Non- Preemptive), SJF (Preemptive)

Assignment

- Write a program to create a child process and print the contents of a file in the child process. Parent process must use waitpid() to collect the termination status of the child process and print message accordingly to the user.

THANK YOU