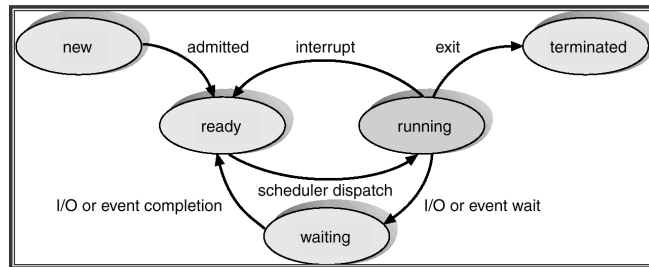


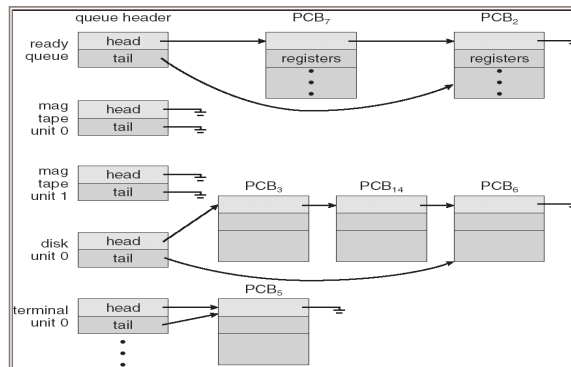
Process States

- As a process executes, it changes its *state*:
 - new: The process is being created.
 - running: Instructions are being executed.
 - waiting (blocked): The process is waiting for some event to occur.
 - ready: The process is waiting to be assigned to a CPU.
 - terminated: The process has finished execution.

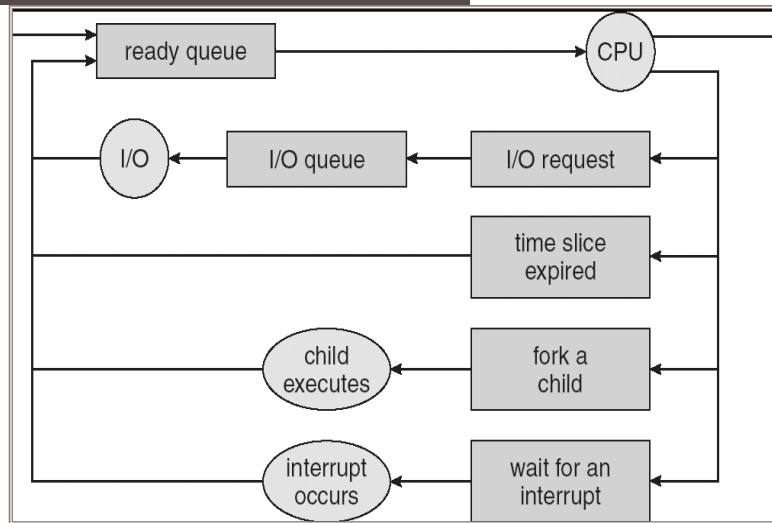


Process Scheduling Queues

- **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
- **Processes migrate among the various queues**



Process Scheduling



CSE660: Introduction to Operating Systems

3

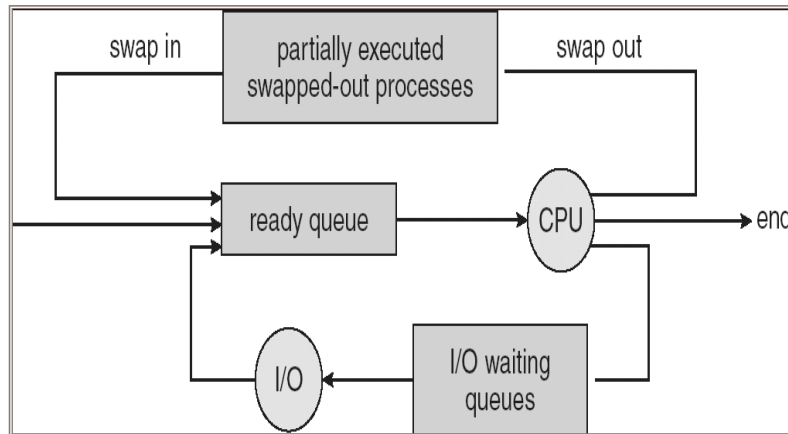
Schedulers

- **Long-term scheduler** (or job scheduler)
 - selects which processes should be brought into the ready queue
 - invoked very infrequently (seconds, minutes) ⇒ (may be slow)
 - controls the *degree of multiprogramming*
 - Should balance different types of processes:
 - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
 - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- **Short-term scheduler** (or CPU scheduler)
 - selects which process should be executed next and allocates CPU
 - invoked very frequently (milliseconds) ⇒ (must be fast)

CSE660: Introduction to Operating Systems

4

Addition of Medium Term Scheduling



CSE660: Introduction to Operating Systems

5

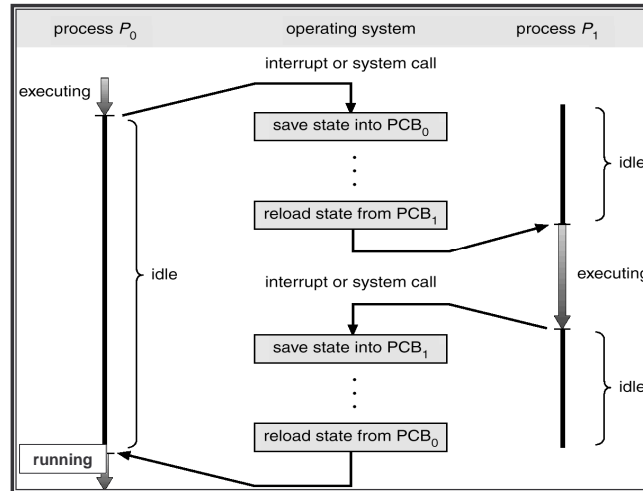
Context Switch

- Switch CPU from one process to another
- Performed by scheduler (**dispatcher**)
- It includes:
 - save PCB state of the old process;
 - load PCB state of the new process;
 - Flush memory cache;
 - Change memory mapping (TLB);
- Context switch is expensive(1-1000 microseconds)
 - No useful work is done (pure overhead)
 - Can become a bottleneck

CSE660: Introduction to Operating Systems

6

CPU Switch From Process to Process



CSE660: Introduction to Operating Systems

7

Process Creation

- System initialization
 - reboot
- Process creation
 - system call: `fork()`
- Users request to create a new process
 - Command line or click an icon
- Initiation of a batch job
 - Cron
 - Try "Crontab" or "at"

CSE660: Introduction to Operating Systems

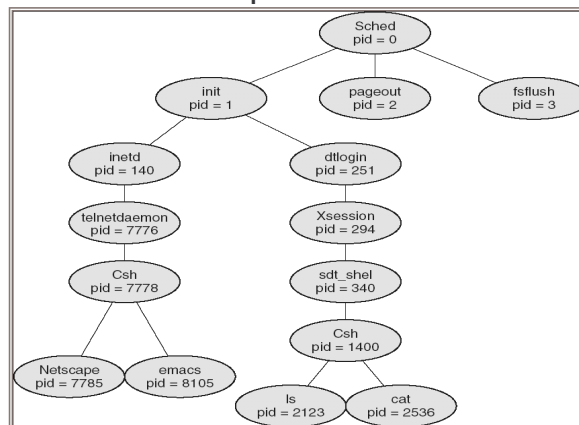
8

Process Termination

- Normal exit (voluntary)
 - End of main()
- Error exit (voluntary)
 - exit(2)
- Fatal error (involuntary)
 - Divide by 0, core dump
- Killed by another process (involuntary)
 - Kill procID, end task

Process Hierarchies

- Parent creates a child process, a child process can create its own processes

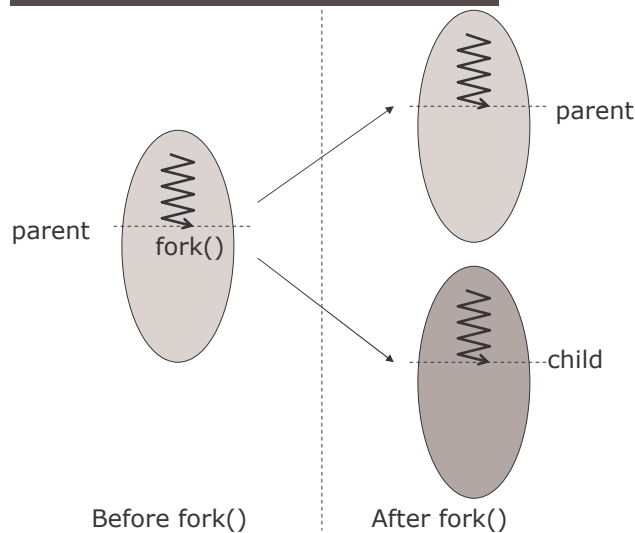


Unix Process Related System Calls

int fork(void);

- A new process is created that is an exact copy of the process making the system call, i.e. a copy of the calling process is made and it runs as a new process. The copy is of the memory image of the calling process at the moment of the fork system call, not the program the calling process was started from.
- The new process does not start at the beginning but at the exact point of the fork system call.
- Thus, right after the fork there are two processes (the issuing process and newly created process) with identical memory images
- The return value to the parent is the process identifier of the new process. The return value to the child process is 0.

Processes after fork()



Unix Process Related System Calls

int execv(char **programName*, char **argv*[]);

The program *programName* is loaded in the calling process address space. Thus, the old program in the calling process is overwritten and will no longer exist. The arguments are in the argument vector *argv*, which is an array of strings, that is, an array of pointers to characters. (There are 6 versions of exec system call)

void exit(int *returnCode*);

This system call causes a process to exit. The *returnCode* is returned to the parent process if it waits for its child process to terminate.

int wait(int * *returnCode*);

This system call will cause the calling process to wait until any process created by the calling process exits. The return value is the process identifier of the process that exited. The return code is stored in *returnCode*.

Unix Processes: Example 1

What's the functionality of the following code snippet?

```
ProgA
{
  int x; char *arg[1]={0};
  x=fork();
  if (x==0) execv("ProgB",arg);
  exit(3)
}
```

Unix Processes: Example 2

Write C code for a process A which creates another process with code from ProgB file, then the process A waits for its child process to terminate, before it exits.

```
ProgA
{
  int x,y,z; char *arg[1]={0};
  x=fork();
  if (x==0) execv("ProgB",arg);
  y=wait(&z);
  exit(3)
}
```

```
ProgB
{
  -
  exit(5); /*point A*/
  -
  exit(1); /*point B*/
}
```

If the child process exits at point A, then z=5, while if it exits at point B then z=1.

Unix Processes: Example 3

This is the example from Figure 3.10 in the textbook and it illustrates how Unix commands (in particular ls command) can be issued from a program.

```
void main (int argc, char *argv [ ])
{
  int pid;
  pid = fork (); /* fork another process*/

  if (pid <0) { /* error*/
    fprintf (stderr, "Fork failed");
    exit(-1);
  } else if (pid == 0) { /*child process*/
    execlp ("/bin/ls", "ls", NULL);
  } else { /*parent process*/
    wait(NULL) ; /*waits for child to terminate*/
    printf ("Child completed");
    exit(0);
  }
}
```

Unix Processes: Example 4

Write C code for a process A which creates another process to execute code from file "gcc (a compiler)" with a list of 3 parameters passed. Then the process A waits for its child process to terminate, before it exits.

```
ProgA
{
  int x,y,z;
  char *argv[4]={ "gcc", "-o", "NameFileToCompile", 0};

  x=fork();
  if (x==0) execv("gcc",argv);
  y=wait(&z);
  exit(0)
}
```