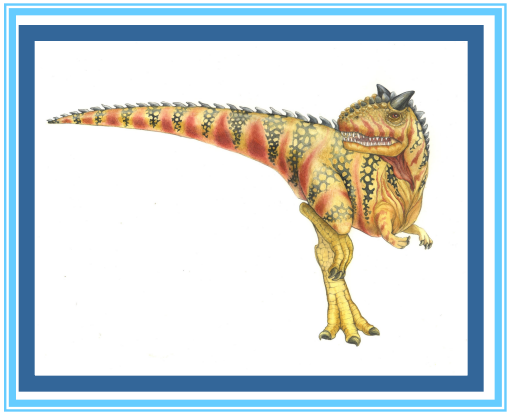
Section 3: Processes



Chapter 3: Processes

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

Objectives

- To introduce the notion of a process -- a program in execution, which forms the basis of all computation
- To describe the various features of processes, including scheduling, creation and termination, and communication

Process/Program

- A process is a program in execution
- What's a program: source code (init.c)

```
int mybigarray[50000] = {1,2,3,4};
int main (void) {
        mybigarray[0] = 3;
        return 0;
}
```

- Once compiled we get an object file (init.o) which has among several sections the following two:
  - Code segment: the executable code
  - Data segment: initialized static variables
- See the output of deassembling init.o using objdump –d and then –s –d

Part of disassembling init.o

Disassembly of section .text:

0000000000004f0 <_start>:

```
4f0:    31 ed                   xor    %ebp,%ebp
4f2:    49 89 d1                mov    %rdx,%r9
4f5:    5e                      pop    %rsi
4f6:    48 89 e2                mov    %rsp,%rdx
4f9:    48 83 e4 f0             and    $0xfffffffffffffff0,%rsp
4fd:    50                      push   %rax
4fe:    54                      push   %rsp
```

Process/Program

- Notice what happens with this second program not-init.c:

```
int mybigarray[50000];
int main (void) {
        mybigarray[0] = 3;
        return 0;
}
```

- The size of the object file not-init.o is 8216 bytes while init.o is 208216 bytes.
- The array mybigarray is not initialized in not-init.c, therefore the memory space is not allocated in the data section of the object file not-init.o
- Only once not-init.o is loaded in main memory will the array be initialized with all 0's

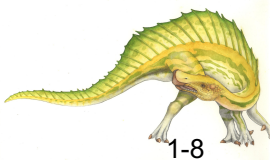
Object file

- So the code (text section) and the static data (data section) are separated in the object file.
- How about local variables, function parameters and variables allocated dynamically?

```
 int Sum(int n);
 int main(){
      int num = 2;
      Sum(num);
      return 0;
}
int Sum(int n){
      int s = 0;
      while( n > 0 )          {
            s += n; n--;
      }
      return s;
}
```

Dynamic data

- No storage is allocated for local variables and function parameters in the object code (binary)
  - Only the name of these variables exist in the text segment
- If a variable is assigned a value in the program, the value is only represented as text, i.e. as ASCII characters
- Data for local and function parameters are stored in the process memory addresses called “run time stack”
- Finally, variables for which storage is allocated dynamically at run time, such as those allocated using an instruction like “malloc” in C, their storage come from the process memory addresses called “heap”



Compiled code representation

Static Data

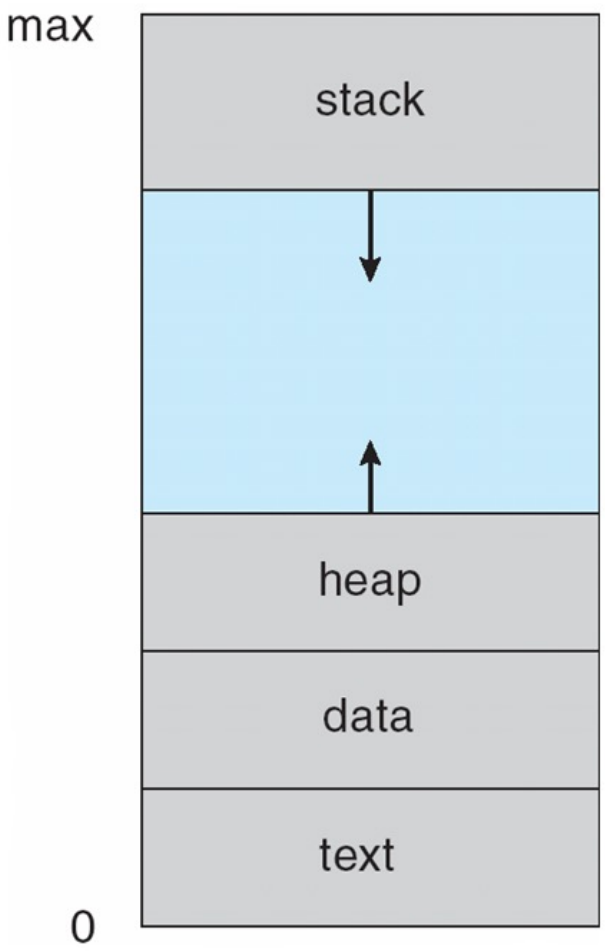
int big-array[5000]; has memory reserved for it if initialized in the text segment

Code, i.e. text, ascii characters

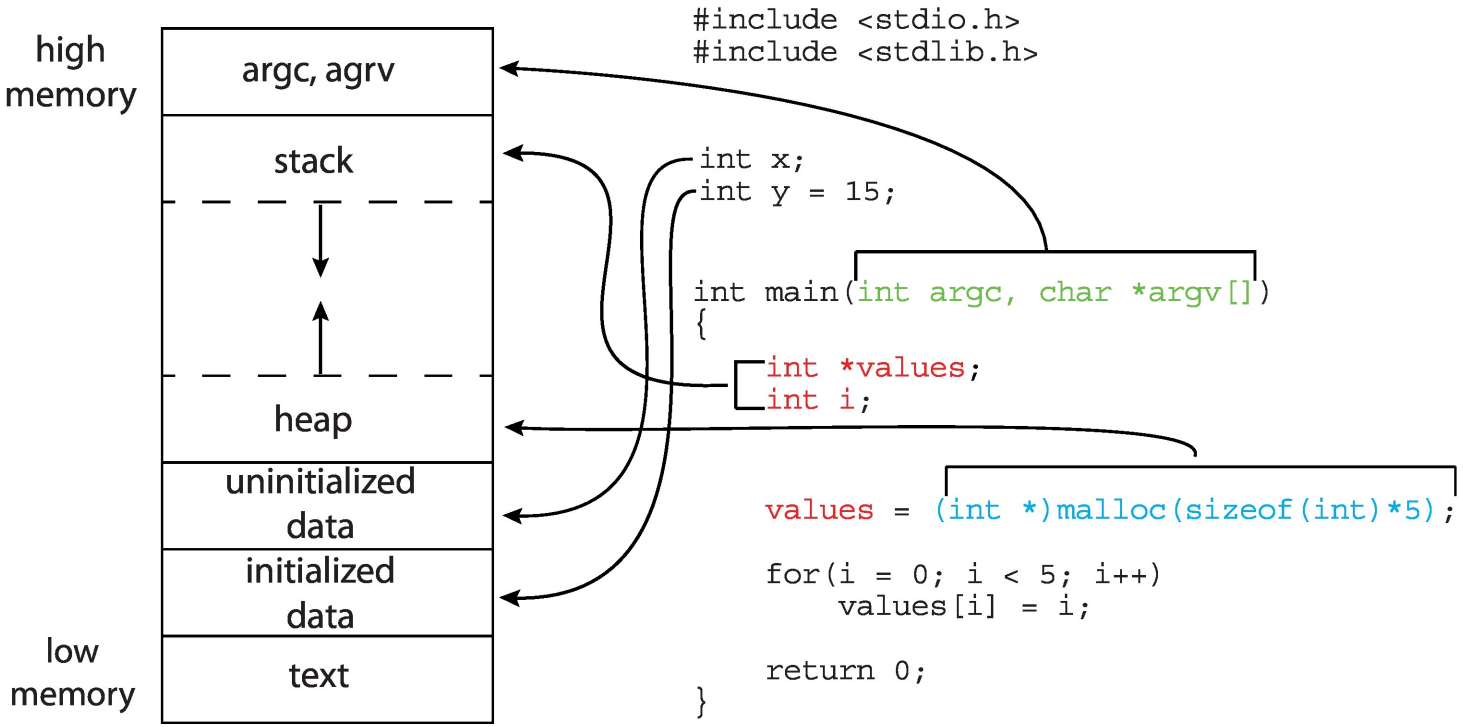
even int i = 6; float j[26]; is all text, no memory is created to store these variables

From program to process

- The object file of a program is stored permanently on the file system
- In order to execute it must first be copied in the main memory.
- As the program executes, it generates data, so part of the main memory must be allocated for storing these data
  - The run time stack
  - The heap
- It is impossible to know before execution how much memory is needed for the stack and the heap
  - A fix amount of main memory is allocated to a program to grow stack and heap



Process memory layout of a C Program



Process ID

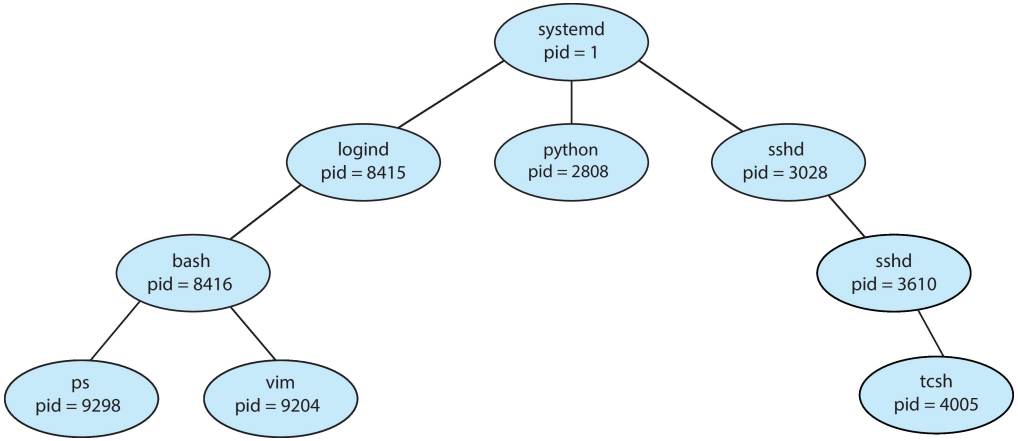
- Each process has a Process ID (PID), which is associated to a process at its creation

- The following program output the PID of the process running that program and its parent process ID. See code PID1.c

```
#include <stdio.h>
#include <unistd.h>
int main (void) {
printf("I am process %ld\n", (long)getpid());
printf("My parent is %ld\n", (long)getppid());
return 0;
}
```

Unix process creation

- All processes are created by other processes except for the first one (process 0) which is created when the system boots.
- Except process 0, all the other processes have a parent process, i.e. the process that has created them
- Most processes have also child processes, i.e. processes they have created
- All together processes form a tree which can be displayed using the command pstree -p



Unix process creation

- In Unix OSs, a process can create a new process by executing the command fork().
- The fork() command creates a child process, the process calling fork() becomes the *parent.* See fork1 and ps –f

```
#include <stdio.h>  #include <unistd.h>  #include <sys/types.h>
int main(void) {
    pid_t childpid;
    childpid = fork();
    if (childpid == 0) /* child code */
        printf("I am child %ld\n", (long)getpid());
     else /* parent code */
         printf("I am parent %ld\n", (long)getpid());
    return 0;
}
```

Implementation of process creation

- As we can see, the function fork() has no argument, how can it knows from which program it fills the text and data sections of the child process???
- Answer: it is filled with the text and data section of the parent process
- Actually, the OS copies the parent's memory image in the child process, the child process is a copy of its parent.

```
#include <stdio.h>  #include <unistd.h>  #include <sys/types.h>
int main(void) {
     pid_t childpid;
     childpid = fork();
     if (childpid == 0) /* child code */
          printf("I am child %ld\n", (long)getpid());
      else /* parent code */
           printf("I am parent %ld\n", (long)getpid());
     return 0;
}
```

Implementation of process creation

- After a fork() there are two completely identical processes (except for the PID)
- Both processes continue execution at the instruction after the fork() statement
- However, parent and child can tell each other apart because fork() returns two different values dependent on whether a process is the parent or the child
- fork() returns 0 to the child process and return the PID of the child process to the parent process: (see code fork2.c)

```
pid_t childpid; pid_t mypid; mypid = getpid();
childpid = fork();
if (childpid == 0) /* child code */
        printf("I am child with PID = %ld, My parent PID is = %ld\n", (long)getpid(),
        (long)mypid);
else /* parent code */
        printf("I am parent with PID = %ld, My child PID is = %ld\n", (long)getpid(),
        (long)childpid);
return 0;
```

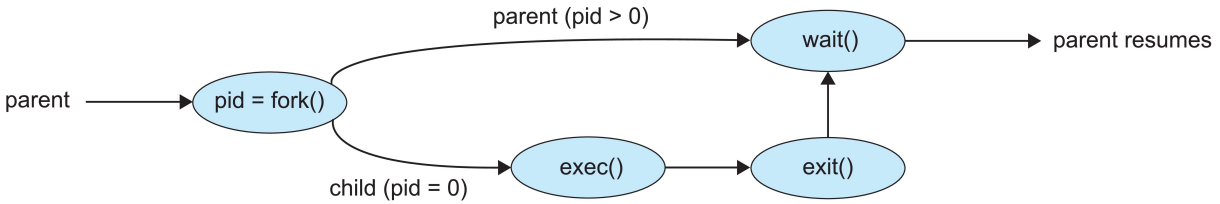
Parent's address space copied in child

- This example illustrates the fact that the fork() creates a child process that has the memory space of its parent process. The child starts execution after the fork()
- The variable mypid is initialized by the parent process. When the child print the value of mypid it is the same as for the parent. The child get a copy of all the parent variables (see fork3-memory)

```
#include <stdio.h> #include <unistd.h> #include <sys/types.h>
int main(void) {
    pid_t childpid;
    pid_t mypid;
    mypid = getpid();
    childpid = fork();
    if (childpid == 0) /* child code */
            printf("I am child %ld, ID = %ld\n", (long)getpid(), (long)mypid);
    else /* parent code */
            printf("I am parent %ld, ID = %ld\n", (long)getpid(), (long)mypid);
    return 0;
}
```

Making child and parent different

- Creation of two completely identical processes is not very useful
- There are two ways we can have parent and child execute different codes
  - One is to have code written for the child in the parent process and require that only the child execute it (like in the previous example)
  - The second one is as in the figure below, the child process executes the function **`exec()`** after the **`fork()`** which load a new binary in the child process.
- Parent process calls **`wait()`** waiting for the child to terminate



Parent and child: same computation

- In this example, before the fork(), one instance of this program is running (is a process) which executes x = 0.
- After the fork() there are two instances of this program running, the parent and child processes, both execute the assigment x = 1. (See fork-same.c)

```
#include <stdio.h>
#include <unistd.h>
int main(void) {
    int x = 0;
    fork();
    x = 1;
    printf("I am process %ld and my x is %d\n", (long)getpid(), x);
    return 0;
}
```

Parent and child: different computation

- In this example, parent and child processes choose which part of the code to execute base on the value returned by fork() (see fork-different.c)

```
#include <stdio.h>  #include <unistd.h>  #include <sys/types.h>
int main(void) {
     pid_t childpid;
     childpid = fork();
     if (childpid == 0) /* child code */
          printf("I am child %ld\n", (long)getpid());
      else /* parent code */
           printf("I am parent %ld\n", (long)getpid());
     return 0;
}
```

Different parent-child using exec()

- This code shows how to use exec() such that the child process loads a different program to execute it (See fork-exec.c)
- The child executes the linux command “ls -l”

```
#include <stdio.h> #include <stdlib.h> #include <unistd.h> #include <sys/wait.h>
int main(void) {
        pid_t childpid;
        childpid = fork();
        if (childpid == 0) { /* child code */
                execl("/bin/ls", "ls", "-l", NULL);
                perror("Child failed to exec ls");
                return 1;
        }
        if (childpid != wait(NULL)) { /* parent code */
                perror("Parent failed to wait due to signal or error");
                return 1;
        }
```

Different parent-child using exec()

- In this code, the child first execute the code it has inherited from his parent. Then replace the code of his parent by another one. The code after exec(), printf(), is not executed as it is no longer the code of the child

```
#include <stdio.h> #include <stdlib.h> #include <unistd.h> #include
<sys/wait.h>
int main(void) {
        pid_t childpid; pid_t mypid = getpid();
        printf("I am parent with PID = %ld\n", (long)mypid);
        childpid = fork();
        if (childpid == 0) { /* child code */
          printf("I am child with PID = %ld\n", (long)getpid());
          execl("/bin/ls", "../ls","..", NULL);
          printf("I am child with PID = %ld,  my parent PID is = %ld\n",
(long)getpid(),(long)mypid);
        }
        if (childpid != wait(NULL)) { /* parent code */
                perror("Parent failed to wait due to signal or error");
                return 1;
        }
        return 0;
}
```

The exec() command

- The `exec` family of functions provides a facility to replace the executable of the parent process by a new code
- There are six variations of the `exec` function `execl`, `execlp`, `execle`, `execv`, `execvp` and `execve` which differ in the way command-line arguments are passed.
- `int execl(const char *path, const char *arg0, ... /*, char *(0) */);`
  - The first parameter is the path to the binary which is to be executed
  - The other parameters are pointers to the arguments of the program to execute
- For example in execl("/bin/ls", "ls", "-l", NULL); "/bin/ls", is the path, “ls” is the command, “-l” is an argument of the command “ls”,
- excel() commands always terminates with NULL

Chain of processes

- Each process terminates after the execution of one iteration of the loop
- Each child process continues the loop with *i* equal to the increment *i*++ from the parent process (See fork-chain.c)

```
#include <stdio.h> #include <stdlib.h> #include <unistd.h>#include<sys/wait.h>
int main (int argc, char *argv[]) {
        pid_t childpid = 0;
        int i, n;
        n = atoi(argv[1]);
        for (i = 1; i < n; i++)
                if (childpid = fork())
                        break;
        fprintf(stderr, "i:%d process ID:%ld parent ID:%ld child ID:%ld\n",
        i, (long)getpid(), (long)childpid, (long)getppid());
        wait(NULL);
        return 0;
}
```

One level tree of processes

- In this example, child processes exit the loop immediately after creation.
- Only the parent process executes all the iterations of the loop calling fork(). This creates a tree of processes with parent as the root and the children as leaves (See fork-fan.c)

```
#include <stdio.h> #include <stdlib.h> #include <unistd.h>
int main (int argc, char *argv[]) {
        pid_t childpid = 0;
        int i, n;
        n = atoi(argv[1]);
        for (i = 1; i < n; i++)
                if ((childpid = fork()) <= 0)
                        break;
        fprintf(stderr, "i:%d process ID:%ld parent ID:%ld child ID:%ld\n",
        i, (long)getpid(), (long)childpid, (long)getppid());
        return 0;
}
```

Multilevel tree of processes

- The following program creates a multilevel tree of processes. Using the PID of the processes created you can construct the tree relationship of parent versus child processes (See fork-multilevel.c)

```
#include <stdio.h> #include <stdlib.h> #include <unistd.h>
int main (int argc, char *argv[]) {
        pid_t childpid = 0;
        int i,j, n;
        n = atoi(argv[1]);
         for (i = 1; i < n; i++){
                childpid = fork();
                if (childpid != 0)fprintf(stderr, "i:%d process ID:%ld parent ID:%ld child
ID:%ld\n",  i, (long)getpid(), (long)getppid(), (long)childpid);
         }
        sleep(15);
        return 0;
}
```

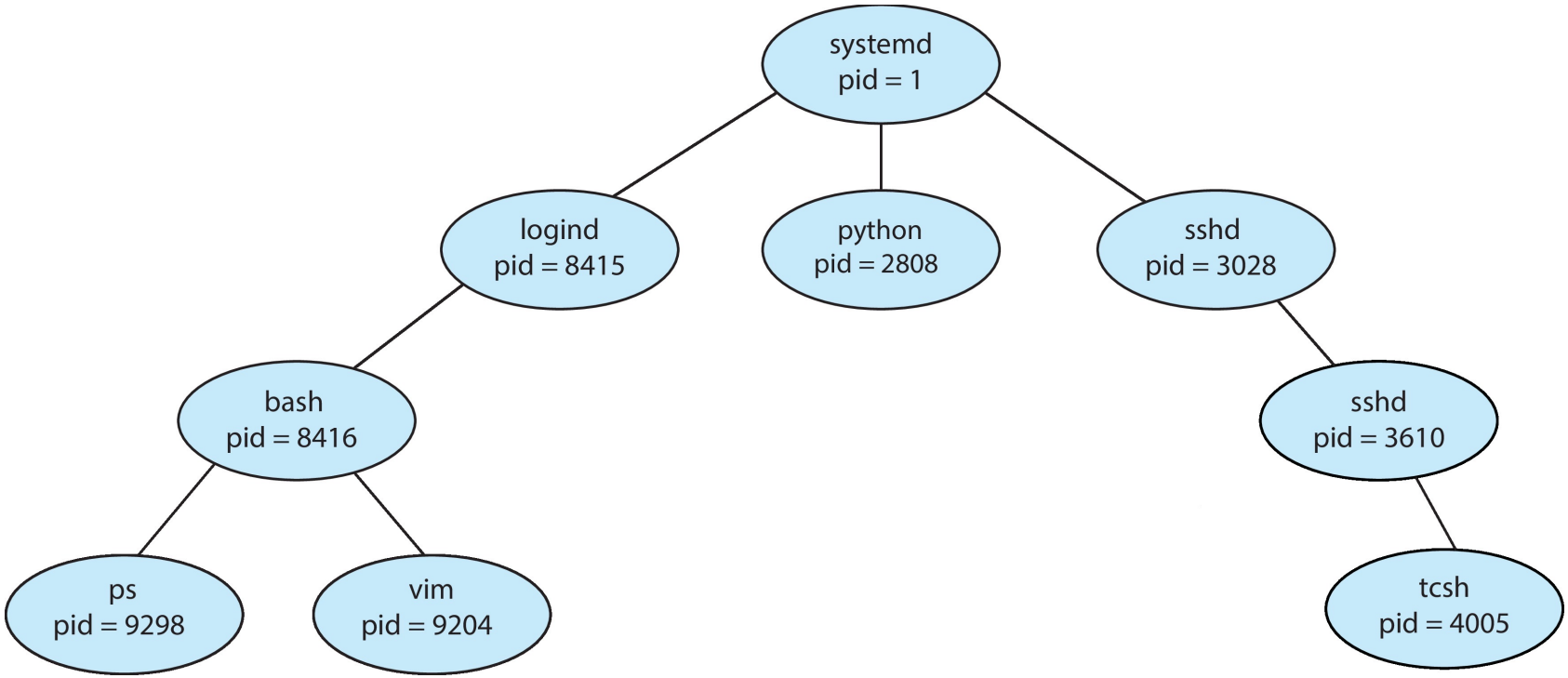
Multilevel tree of processes

- The root process p0-0 forks n-1 processes p0-1, p0-2, .., p0-n-1
- Process p0-1 forks n-2 processes p1-2, p1-3, .., p1-n-1
- Process p0-2 forks n-3 processes p2-3, p2-4, .., p2-n-1

- Process p1-2 forks n-3 processes p2-3, p2-4, .., p2-n-1

```
#include <stdio.h> #include <stdlib.h> #include <unistd.h>
int main (int argc, char *argv[]) {
        pid_t childpid = 0;
        int i,j, n;
        n = atoi(argv[1]);
         for (i = 1; i < n; i++){
                childpid = fork();
                if (childpid != 0)fprintf(stderr, "i:%d process ID:%ld parent ID:%ld child ID:%ld\n",  i,
(long)getpid(), (long)getppid(), (long)childpid);
         }
        sleep(15);
        return 0;
}
```

Tree of processes in a Linux version



wait() command

- The wait() command executed by a parent process causes the caller to block until a child finishes.
- The `wait(&status)` function causes the caller to suspend execution until a child's status becomes available or until the caller receives a signal
  - for example, the child executes exit(), which returns a signal to the parent, the signal is captured by wait())
- wait() returns the pid of the process that interrupt the wait. If something else causes the wait to end it returns -1
- wait(NULL) means wait for any changes in the status of the child process
- The waitpid() command causes the parent to wait for a specific child, waitpid(pid,&status,options)
- See wait4.c for wait() and fork-wait.c for waitpid()

wait() example

```
#include <stdio.h> #include <unistd.h> #include <sys/types.h>
#include <sys/wait.h>
int main (void) {
        pid_t childpid;
        childpid = fork();
        if (childpid == 0)
                fprintf(stderr, "I am child %ld\n", (long)getpid());
        else if (wait(NULL) != childpid)
                fprintf(stderr, "A signal must have interrupted the wait!\n");
        else
                fprintf(stderr, "I am parent %ld with child %ld\n", (long)getpid(),
                        (long)childpid);
        return 0;
}
```

Process Termination: exit() command

- A called process executes last statement and then asks the operating system to delete it using the **`exit(exit_statut)`** system call (See exit1.c)
- Returns status = 0 from child to parent (via **`wait()`**) when exit success
- Process' resources are deallocated by operating system
- Child process may also exit by executing return 0, where 0 means exit normally

Exit() command example

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

int main(void) {
pid_t childpid, thechild;
childpid = fork();
if (childpid == 0){ /* child code */
   printf("I am child, I wait 5 seconds %ld\n", (long)getpid());
   sleep(5);
   exit(0);
 }
 else {/* parent code */
   printf("I am parent I will wait for child to exit %ld\n", (long)getpid());
  thechild = wait(NULL);
  printf("Child exit %ld\n",(long) thechild);
   return 0;
 }
}
```

Process Termination: kill() command

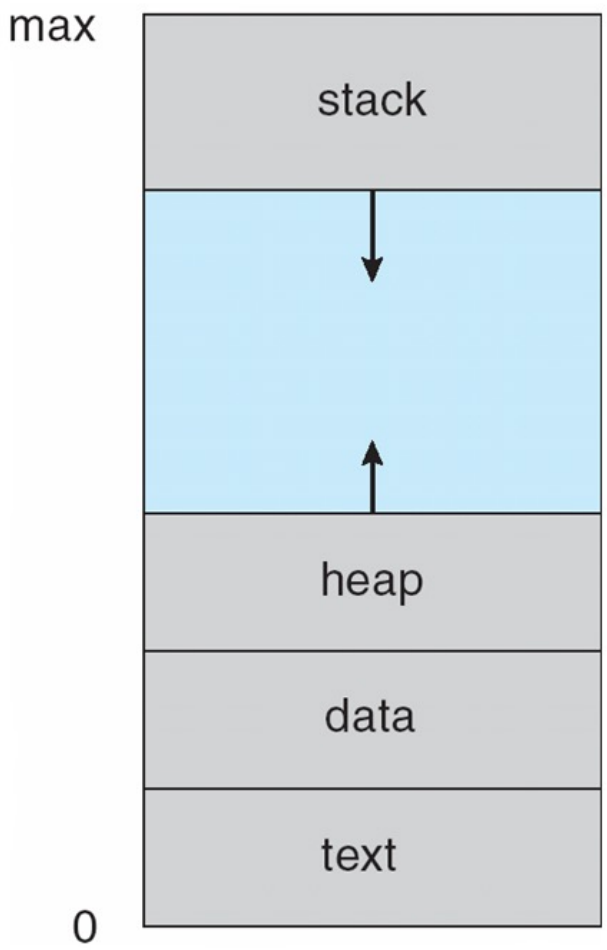
- Parent may terminate the execution of children processes using the **`kill(pid,signal)`** system call (see kill1.c).
- Signal is usually SIGKILL

Example of kill() command

```
#include <stdio.h>#include <unistd.h>#include <sys/types.h>#include
#include <stdlib.h>#include <signal.h>
int i = 0;
int main(void) {
pid_t childpid, mypid;mypid = getpid();
childpid = fork();
 if (childpid == 0){ /* child code */
   while (i++ <10){
     printf("I am child with PID = %ld, loop index is  = %d\n", (long)getpid(), i);
     sleep(1);}
 }
 else {/* parent code */
   printf("I am parent with PID = %ld, My child PID is = %ld\n", (long)getpid(),
(long)childpid);
   while (i++ < 7) {
     printf("I am parent in the loop PID = %ld, My loop index is is = %d\n", (long)getpid(),
i);
     sleep(1);}
   kill(childpid,SIGKILL);
   printf("I am parent with PID = %ld, My child PID is = %ld, I have killed my child
process\n", (long)getpid(), (long)childpid);
 }return 0;
}
```

Other components of a process

- Memory allocated to a process is the most visible part.
- Also, in the kernel of the OS, there is a data structure called process table that hold quite a bit of information needed to manage processes.
- Further, associated to each process, is the process control block (PCB) which contains detailed information about its process



Process table

- Information includes in the process table about each process:
  - PID process id
  - UID process owner
  - PRI process priority
  - ST current process state (s = sleep)
  - PPID the parent process
  - PGRP process group

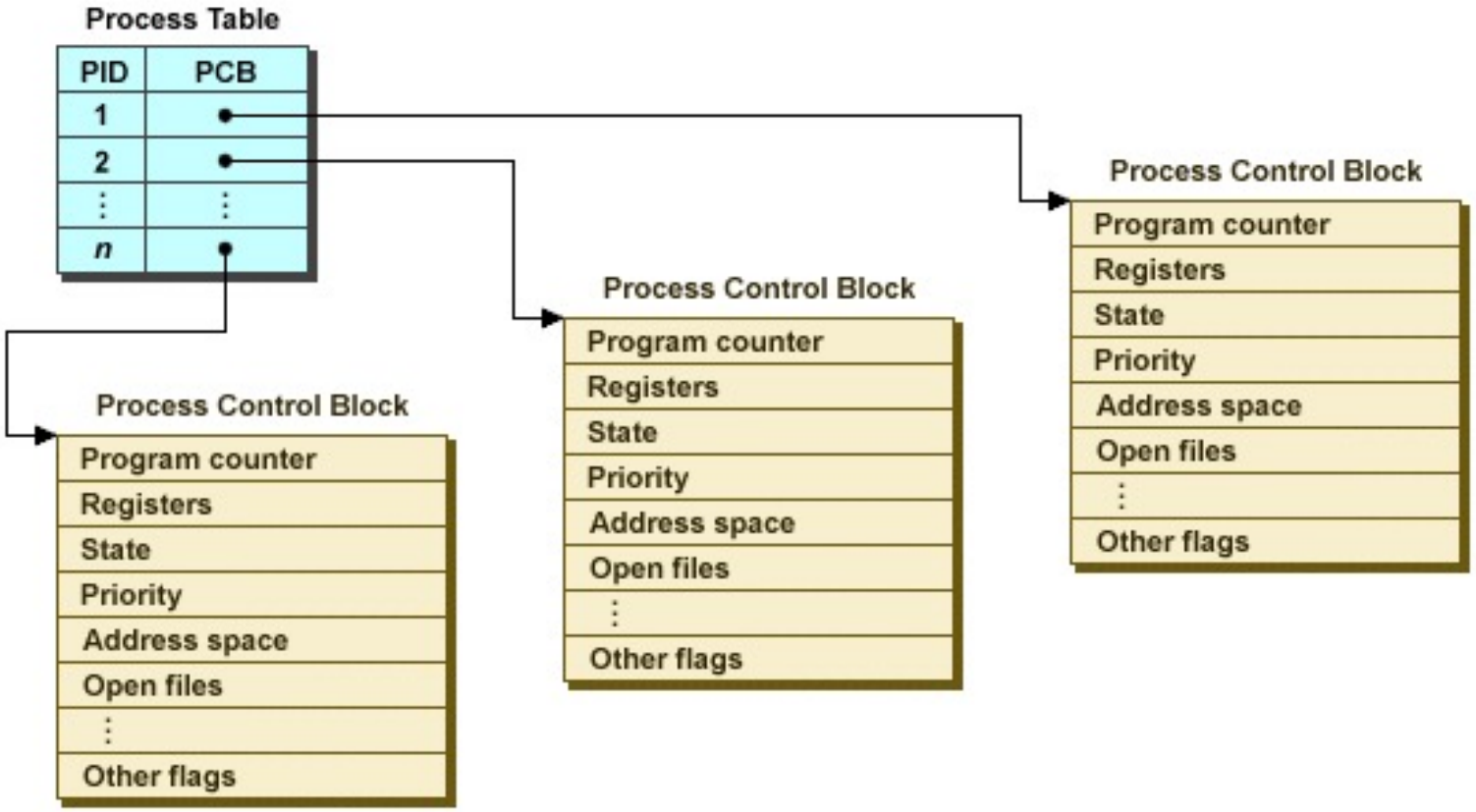
```
PROC TABLE SIZE = 47
SLOT ST PID PPID PGRP UID PRI CPU   NAME
  0   s   0   0    0    0   95   0  sched
  1   s   1   0    0    0   66   1  init
  2   s   2   0    0    0   95   0  vhand
  3   s   3   0    0    0   81   0  bdflush
  4   s   4   1    1    0   95   0  kmadaemon
  5   s   5   1   11    0   95   0  htepi_daemon
  6   s   6   1   16    0   95  22   strd
  7   s 289   1  289    0   73   0  ksh
```

Windows and Linux process tables

- PID 0 is a special process that is created when the system boots
- The table on the previous slide and the table displayed on Ubuntu using the command ps –ef are process tables of unix like OSs (ps displays information about processes associated with the user. The -a option displays information for processes associated with terminals. The -A option displays information for all processes)
- In this class our laptops have two OSs, Ubuntu and Windows, therefore there are two process tables
- To display the process table of Windows, go to the Windows command line and enter the command tasklist
- Tasklist displays the Windows processes, they are a different set of processes compared to the one displays in Ubuntu

Process table and Process Control Block

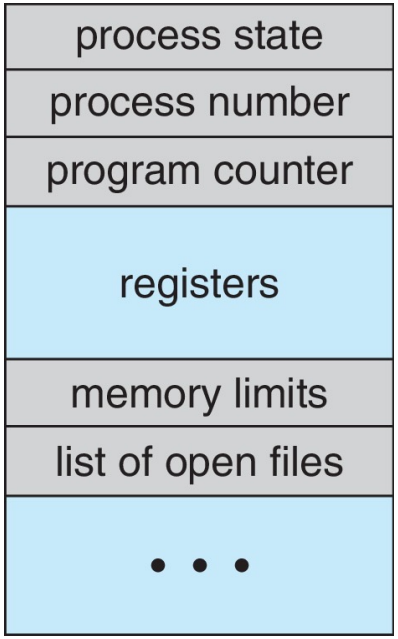
- Each process is associated with a Process Control Block (PCB)
- There is one process table in the OS but there is one PCB for each process



Process Control Block (PCB)

The PCB contains the following information about its process:

- Process state – running, waiting, etc.
- Program counter – location of instruction to be executed next
- 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



Process State

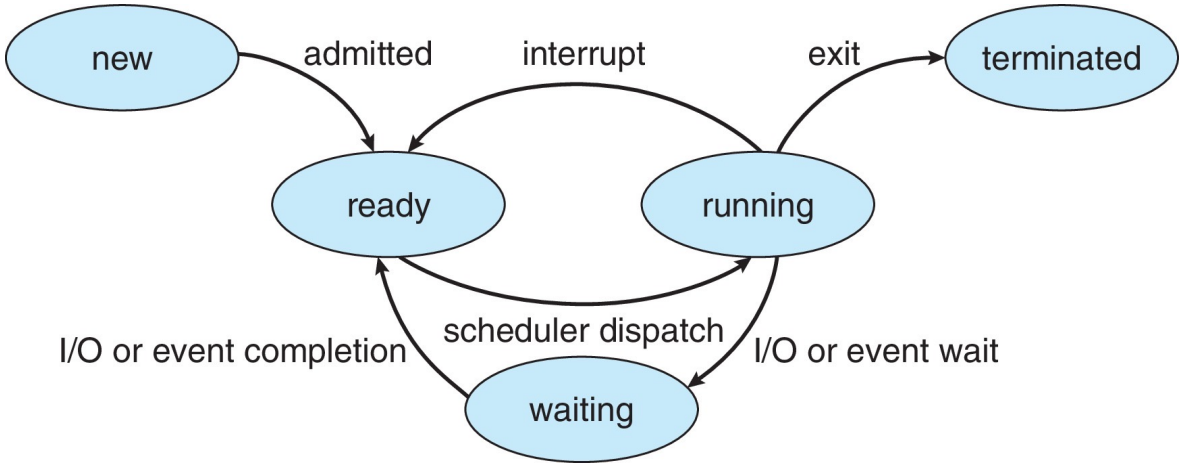
- The *state* of a process indicates its status at a particular time.
- As a process executes, it changes state
  - **New**: The process is being created
  - **Running**: Instructions are being executed
  - **Waiting**: The process is waiting for some event to occur
  - **Ready**: The process is waiting to be assigned to a processor
  - **Terminated**: The process has finished execution

Diagram of Process State

- A *state diagram* is a graphical representation of the allowed states of a process and the allowed transitions between states

- The nodes of the graph in the diagram represent the possible states, and the edges represent possible transitions

- The labels on the arcs specify the conditions that cause the transitions between states to occur



Process states explained

- While a program is on the way to be transformed into an active process, it is said to be in the *new* state.
- Once the transformation is completed, the OS puts the process in a queue of processes that are ready to run. The process is then in the *ready* or *runnable* state.
- Eventually the component of the OS called the process scheduler selects a process to run. The process is in the *running* state when it is actually executing on the CPU
- A process is in a *blocked* state if waiting for an event and is then not eligible to be picked for execution. Typically a process moves to the *blocked* state when it performs an I/O request

Steps in a Process Creation

1. Allocate a slot in the process table for the new process.
2. Assign a unique process ID to the child process.
3. Copy of process image of the parent, with the exception of any shared memory.
4. Increment the counters for any files owned by the parent, to reflect that an additional process now also owns those files.
5. Assign the child process to the Ready to Run state.
6. Because both processes (parent and child) may perform different tasks, we should be able to identify them. The return value is:
   - Child's process ID (in parent process)
   - Zero (in the child process)

- Child and parent processes execute different parts of the same program or the child process execute another program

Zombie processes

- When a child exits, the parent process must wait() on it to get its exit code.
- That exit code is stored in the process table until the parent execute wait()
- Between the time a child exits and wait() is executed, the child is called a zombie
- A zombie process has been erased from the main memory, it does not occupied space and does not used cpu
- However, a zombie process has still his entry in the process table where its exit status is kept
- (see zombie.c)

Example of zombie process

```
#include <stdio.h>#include <stdlib.h> #include <sys/types.h>
#include <unistd.h>#include <sys/wait.h>
/*the child will become a zombie*/
int main()
{
   pid_t child_pid = fork();
   if (child_pid > 0) {
     printf("I am parent, will execute wait in 20 seconds, my pid is %ld\n", (long) getpid());
            sleep(20);
            wait(NULL);
            printf("I am parent exiting \n");
   }
   else // Child process
      if (child_pid == 0) {
             printf("I am child, will exit in 5 seconds, my pid is %ld \n", (long) getpid());
         sleep(5);
         printf("I am child exiting \n");
               exit(0);
      }
            return 0;
}
```

Orphan processes

- If a parent process exits when its children are still running (and doesn't kill its children), those children are orphans.
- Orphan processes are immediately adopted by another process, in our case it is the bash process
- An orphan is a normal process, when it exit the adopting parent will take care that it does not become a zombie
- (see orphan.c)

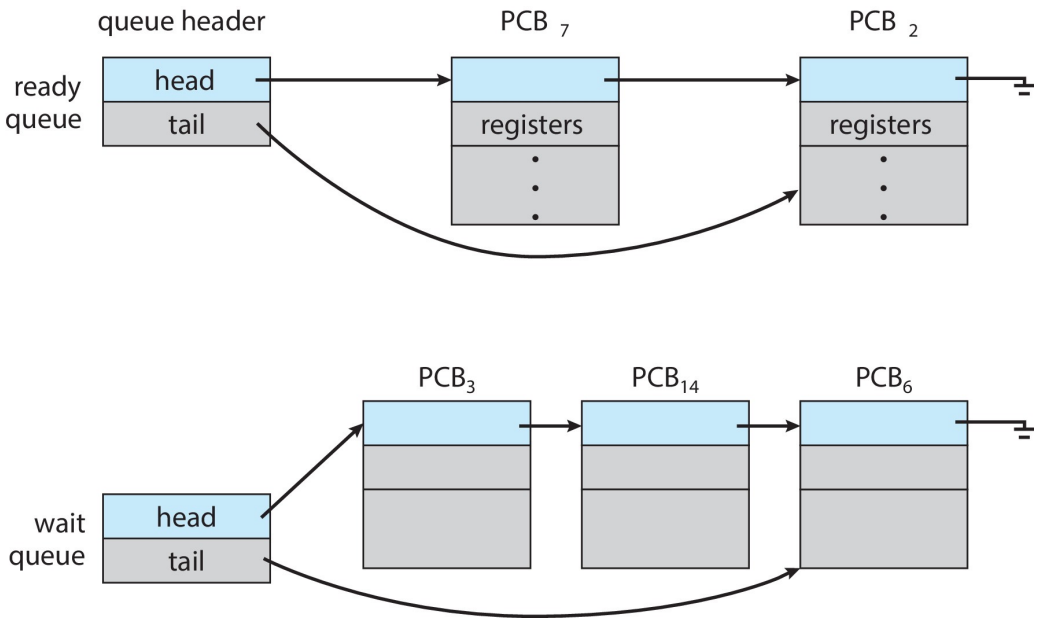
Example of orphan process

```
#include <stdio.h>#include <stdlib.h> #include <sys/types.h>
#include <unistd.h>
/*the child will become an orphan*/
int main()
{
    pid_t child_pid = fork();
    if (child_pid > 0) {
      printf("I am parent, will finish execution in 5 seconds, before my child
%ld\n", (long) getpid());
            sleep(5);
    }
    else // Child process
        if (child_pid == 0) {
              printf("I am child, will sleep 15 seconds, before exiting %ld \n", (long)
getpid());
           sleep(15);
           printf("I am child exiting \n");
        }
            return 0;
}
```

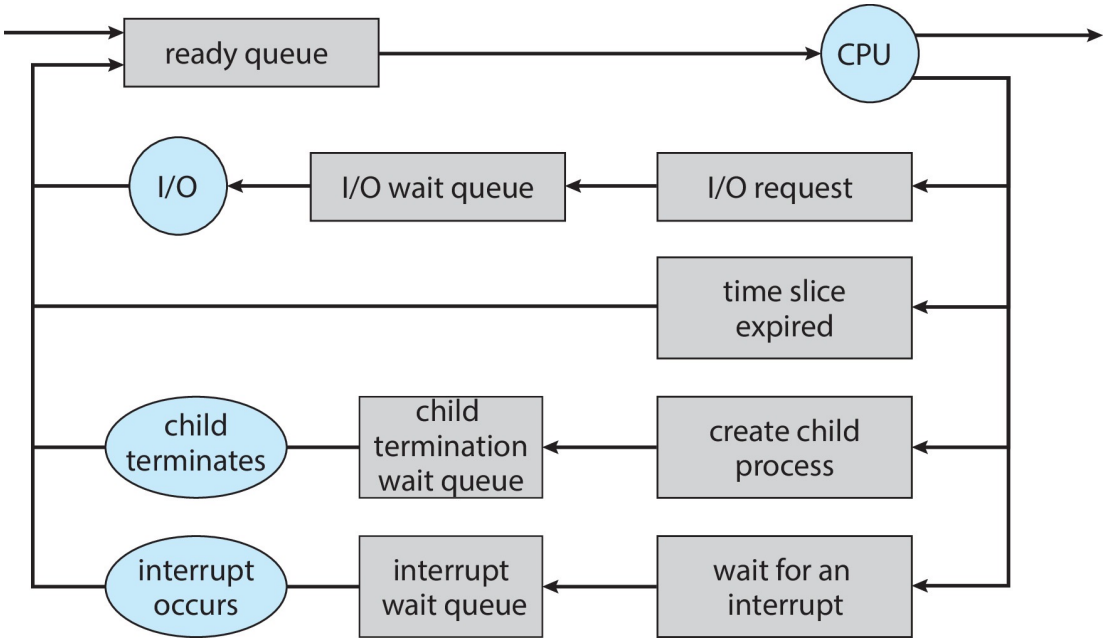
Process Scheduling

- There are always a lot of processes active in same time but only one or few CPUs
- The action of assigning a process to a CPU (moving from ready state to running state) is called **scheduling**
- The **process scheduler** selects among available processes for next execution on CPU core
- Goal -- Maximize CPU use, quickly switch processes onto CPU core
- Maintains **scheduling queues** of processes
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Wait queues** – set of processes waiting for an event (i.e., I/O)
  - **Device queues** – set of processes waiting for an I/O device
  - Processes migrate among the various queues

Ready and Wait Queues



Representation of Process Scheduling

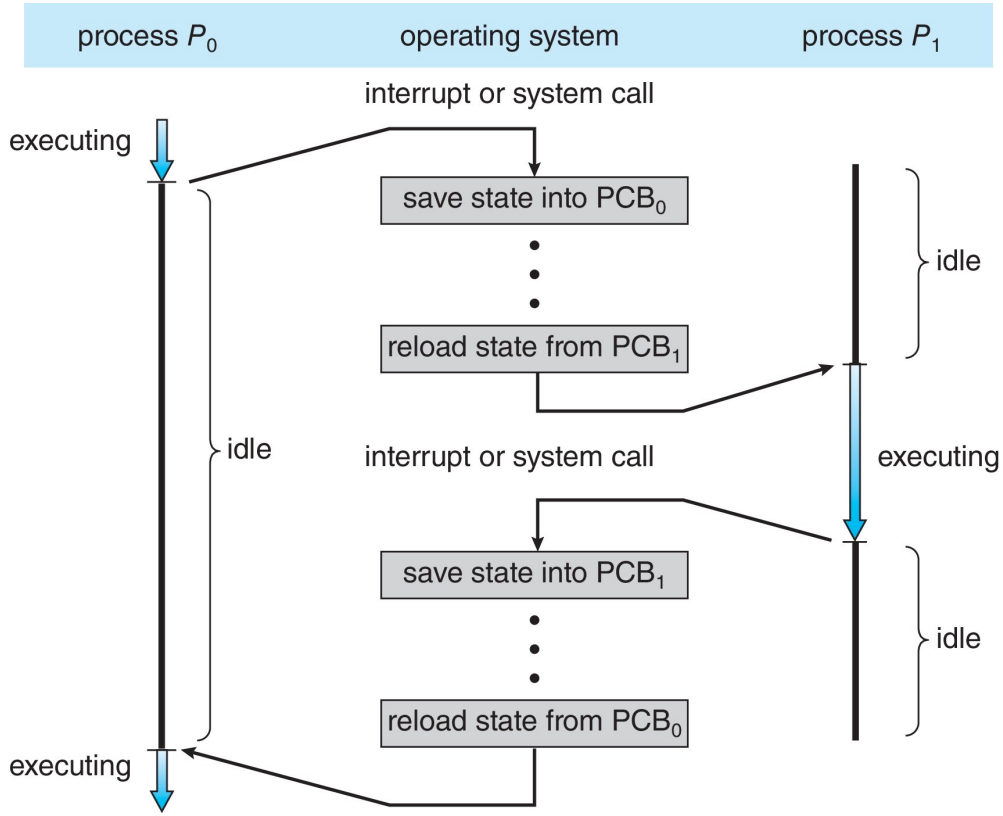


Process context

- A *context switch* is the act of removing one process from the *running* state and replacing it by another.
- The *process context* is the information that the operating systems needs about the process to restart it after a context switch.
- For example, when a process is interrupted
  - the current value of the program counter and processor registers (context data) need to be saved
  - the state of the process is changed to “blocked” or “ready”
- This information is stored in the process control block (PCB).

Context Switch from Process to Process

- A **context switch** occurs when the scheduler removes a process from CPU to replace it by another process.
- System **save the state** of the old process into its PCB and load the **saved state of** the new process from its PCB.



Steps in a Context Switch

- The steps in a context switch are:
  1. Save context of processor including program counter and other registers
  2. Update the process control block of the process that is currently in the Running state
  3. Move process control block to appropriate queue – ready; blocked; ready/suspend
  4. Select another process for execution
  5. Update the process control block of the process selected
  6. Update memory-management data structures
  7. Restore context of the selected process

Context Switch

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

End chapter 3