Inter-Process Communication

IPC in unix systems
1] Exit codes (Child Parent Communication)

Scope of Labs

- Get familiar with Unix environment
- Know the importance of IPC
- Know how to implement and use different IPC techniques

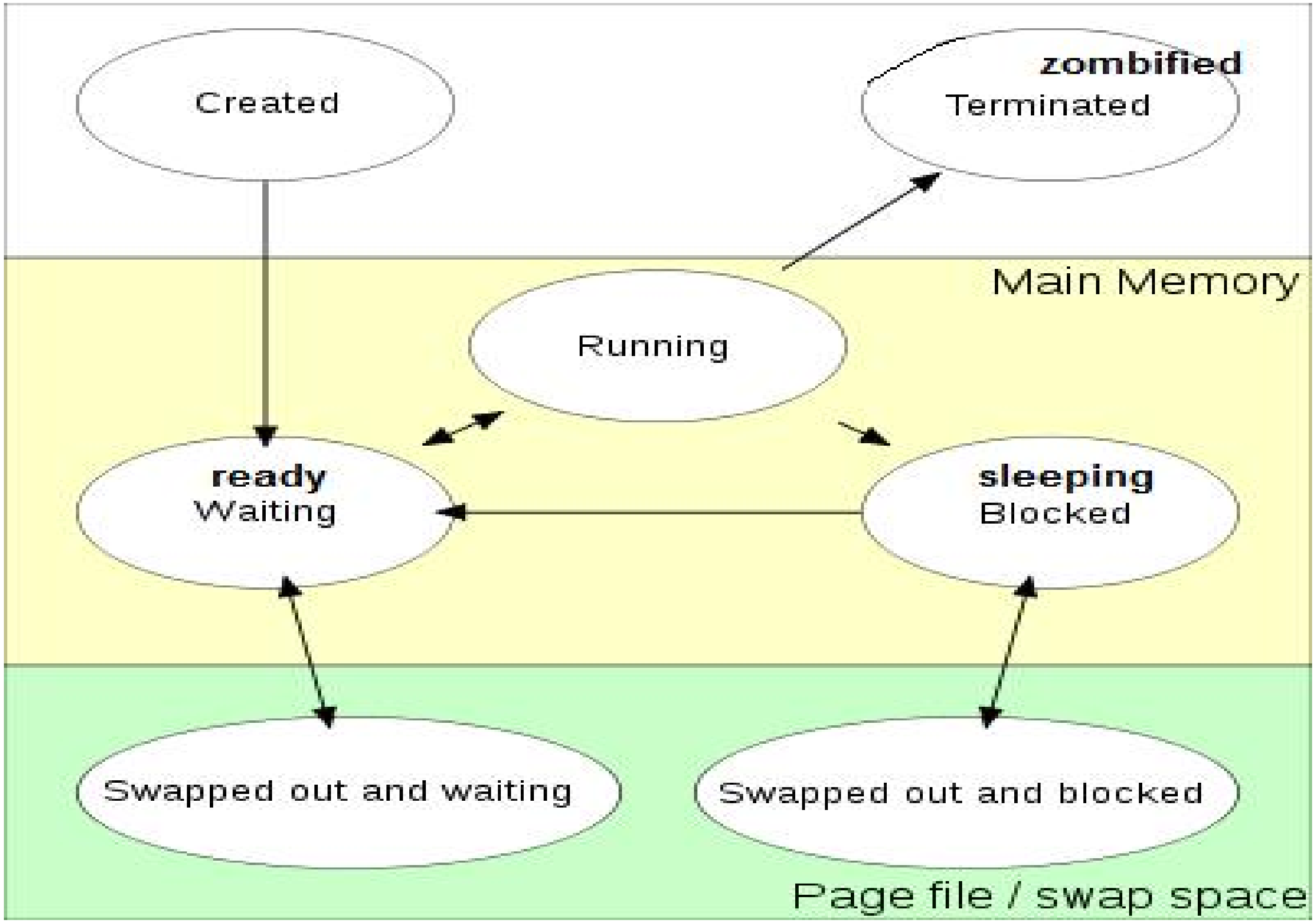
Today's Goals

- Know unix file system structure & basic commands
- Learn about processes
- Learn the goal of IPC (inter-process communication)
- Learn how to Create new process
- Learn how to handle child parent communication

What is a Process

- A process is a running instance of a program , it consists of
  - ***Process code***, which contains the executable portion of the process.
  - ***Process data***, which contains the static data of the process.
  - ***Process stack***, which contains temporary data of the process.
  - ***User area***, which holds the information about signal handling, opened files, and another CPU info for the process.
  - ***Page tables***, which are used for memory management.

Process life cycle



Process Scheduling

- Process runs simultaneously by using a scheduling technique called Round Robin.
- Priority of the process determines how much quantum it takes.

We will talk about that more in section

Inter-Process Communication

- There are several reasons for providing an environment that allows process cooperation:
  - Information sharing
  - Computational Speedup
  - Modularity
  - Convenience
  - Privilege separation

- Two full glasses of water want to mix them
  - We need a third medium to handle the communication

BEFORE COMMUNICATION WE NEED TO CREATE THE PROCESSES FIRST ☺

How is Process Created ? (in Unix)

- When the system is turned on the first process is created, which in turn create the “init” process , the father of all process in the system.
- Each process created gets a unique identifier (PID)
- Init pid = 1
- Further processes are created by other process, the process which create them is called the parent, and the process created is the called the child.
- If a process parent died ….. ! Let’s see what’s happen

Fork()

- A process use the fork() command to create a child process of its own
- The child process takes almost an exact copy of the parent but it is separated from them “has a different address space and user area”
- The child start execution form the fork() statement

- Let’s take a look at some examples

Process01.c

```
main()
{   int pid, x = 3;  printf("\nmy pid = %d\n", getpid());

    pid = fork();

  if (pid == -1) perror("error in fork");

  else if (pid == 0) { //child
    x=7;
    printf("\n PPID:%d, PID:%d, X:%d",getppid(),getpid(),x);
     }

  else {  //parent
        x=19;
    printf("\n PPID:%d, PID:%d, X:%d",getppid(),getpid(),x);
      }

printf("\n Finish: PID:%d, X:%d", getpid(),x);

}
```

Process01.c

```
main()
{   int pid, x = 3;  printf("\nmy pid = %d\n", getpid());

    pid = fork();

 if (pid == -1) perror("error in fork");

 else if (pid == 0) { //child
    x=7;
    printf("\n PPID:%d, PID:%d, X:%d",getppid(),getpid(),x);
     }

 else {  //parent
        x=19;
    printf("\n PPID:%d, PID:%d, X:%d",getppid(),getpid(),x);
      }

printf("\n Finish: PID:%d, X:%d", getpid(),x);

}
```

Process01.c

```
main()
{   int pid, x = 3; printf("\nmy pid = %d\n", getpid());

     pid = fork();

 if (pid == -1) perror("error in fork");

 else if (pid == 0) { //child
     x=7;
     printf("\n PPID:%d, PID:%d, X:%d",getppid(),getpid(),x);
      }

 else {  //parent
         x=19;
     printf("\n PPID:%d, PID:%d, X:%d",getppid(),getpid(),x);
       }

printf("\n Finish: PID:%d, X:%d", getpid(),x);

}
```

Process01.c

```
main()
{   int pid, x = 3;  printf("\nmy pid = %d\n", getpid());

    pid = fork();

 if (pid == -1) perror("error in fork");

 else if (pid == 0) { //child
    x=7;
    printf("\n PPID:%d, PID:%d, X:%d",getppid(),getpid(),x);
     }

 else {  //parent
       x=19;
    printf("\n PPID:%d, PID:%d, X:%d",getppid(),getpid(),x);
      }

printf("\n Finish: PID:%d, X:%d", getpid(),x);

}
```

Process01.c

```
main()
{   int pid, x = 3; printf("\nmy pid = %d\n", getpid());

      pid = fork();

  if (pid == -1) perror("error in fork");

  else if (pid == 0) { //child
      x=7;
      printf("\n PPID:%d, PID:%d, X:%d",getppid(),getpid(),x);
       }

  else {  //parent
          x=19;
      printf("\n PPID:%d, PID:%d, X:%d",getppid(),getpid(),x);
        }

printf("\n Finish: PID:%d, X:%d", getpid(),x);

}
```

Compile and run

- Change the directory to where the code exist using “cd”
- Compile : gcc filename -o outputfilename
- Run : ./ outputfilename

Process02.c (orphan)

```
main()
{ int pid;  printf("\nmy pid = %d\n", getpid());
   pid = fork();
if (pid == -1) perror("error in fork");
else if (pid == 0)  {  //child
        printf("\nI am the child, my pid = %d and my parent's pid = %d\n\n", getpid(), getppid());
        sleep(3);
        printf("\nAgain I am now an orphan child, my pid = %d and my parent's pid = %d\n\n", getpid(), getppid());
     }
 else {  //parent
        printf("\nI am the parent, my pid = %d and my parent's pid = %d\n\n", getpid(), getppid());
        sleep(2);  //give chance to child to get started
     }
printf("\nPID %d terminated\n\n", getpid());  }
```

COMPILE & RUN

Process03.c (wait for my child)

```
main()
{ int pid, stat_loc;
   printf("\nmy pid = %d\n", getpid());
    pid = fork();
if (pid == -1) perror("error in fork");
else if (pid == 0)  //child
      {
        printf("\nI am the child, my pid = %d and my parent's pid = %d\n\n",
getpid(), getppid());
         sleep(3);
       }
```

Process03.c (wait for my child)

```
main()
{ int pid, stat_loc;
   printf("\nmy pid = %d\n", getpid());
    pid = fork();
if (pid == -1) perror("error in fork");
else if (pid == 0)  //child
      {
      printf("\nI am the child, my pid = %d and my parent's pid = %d\n\n", getpid(), getppid());
       sleep(3);
        }
```

Process03.c (wait for my child)

```
main()
{ int pid, stat_loc;
   printf("\nmy pid = %d\n", getpid());
    pid = fork();
if (pid == -1) perror("error in fork");
else if (pid == 0)  //child
      {
      printf("\nI am the child, my pid = %d and my parent's pid = %d\n\n", getpid(), getppid());
       sleep(3);
       }
else  {  //parent
   printf("\nI am the parent, my pid = %d and my parent's pid = %d\n\n", getpid(),
 getppid());
   pid = wait(&stat_loc);
    if(!(stat_loc & 0x00FF))
         printf("\nA child with pid %d terminated with exit code %d\n", pid,
 stat_loc>>8);

    if(WIFEXITED(stat_loc))
          printf("\nChild terminated normally with status %d",
 WEXITSTATUS(stat_loc));

    }
printf("\nPID %d terminated\n\n", getpid());
}
```

```
else  {  //parent
   printf("\nI am the parent, my pid = %d and my parent's pid = %d\n\n", getpid(),
 getppid());
   pid = wait(&stat_loc);
    if(!(stat_loc & 0x00FF))
         printf("\nA child with pid %d terminated with exit code %d\n", pid,
 stat_loc>>8);
    }
printf("\nPID %d terminated\n\n", getpid());
}
```

```
var1 = wait(&var2);
```

var1 : is the return of wait function, it contains the pid of the terminated child.

var2 : is an integer variable passed by reference.

First 3 bytes	Last byte
Exit Code	“0” if exit normally

use man waitpid

```
else  {  //parent
   printf("\nI am the parent, my pid = %d and my parent's pid = %d\n\n", getpid(),
 getppid());
   pid = wait(&stat_loc);
    if(!(stat_loc & 0x00FF))
        printf("\nA child with pid %d terminated with exit code %d\n", pid,
 stat_loc>>8);
     }
printf("\nPID %d terminated\n\n", getpid());
}
```

COMPILE & RUN

Process04.c (IPC)

```c
main()
{ int pid, stat_loc;
   printf("\nmy pid = %d\n", getpid());
    pid = fork();
if (pid == -1) perror("error in fork");
else if (pid == 0) { //child
       printf("\nI am the child, my pid = %d and my parent's pid = %d\n\n",
getpid(), getppid());
          exit(42);
}
```

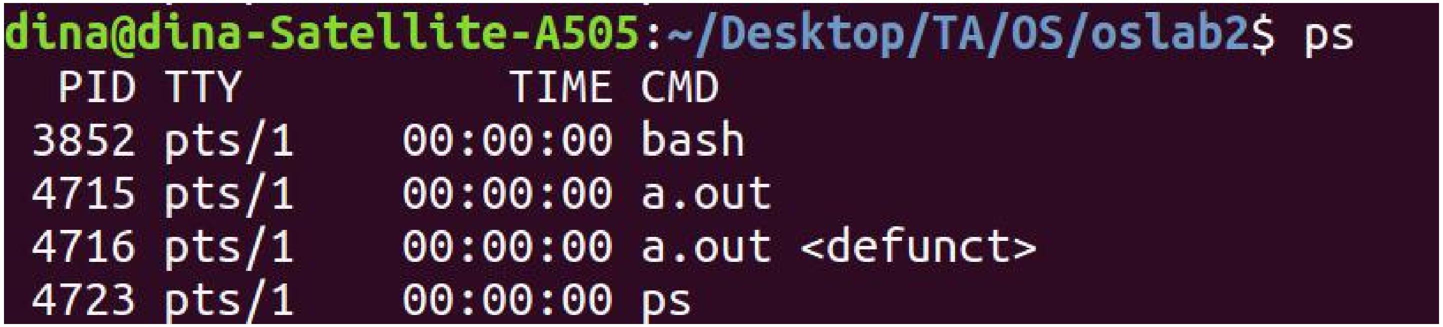
```
 else {  //parent
   printf("\nI am the parent, my pid = %d and my parent's pid = %d\n\n", getpid(),
getppid());

   pid = wait(&stat_loc);

    if(!(stat_loc & 0x00FF))
         printf("\nA child with pid %d terminated with exit code %d\n", pid,
stat_loc>>8);
     }

printf("\nPID %d terminated\n\n", getpid());
}
```

COMPILE & RUN

Process05.c (zombified)

- Change the directory to where the code exist using “cd”
- Compile : gcc filename -o outputfilename
- Run the process in the bg : ./ outputfilename &
- Check the process status : ps

- // defunct :-> represents zombified childs
- The init process take cares of removing them



```
dina@dina-Satellite-A505:~/Desktop/TA/OS/oslab2$ ps
  PID TTY          TIME CMD
 3852 pts/1    00:00:00 bash
 4715 pts/1    00:00:00 a.out
 4716 pts/1    00:00:00 a.out <defunct>
 4723 pts/1    00:00:00 ps
```

Process06.c (change image)

- Execl command is used to change the current process to execute another one using the same ID
- execl("/bin/ps", "ps", "-f", NULL);

use man execl

COMPILE & RUN

Process07.c (kill them all!)

- Run the process in background
- Use kill command to kill the parent check status
- then kill the child and check the status of the processes

Process08.c (being nice)

- Lower priority value means higher priority
- Nice() :→ function used to make the current process more nice to the others (allow increasing priority value if possible so it decreases priority)

Summarize

- Each process has its own space, no other process can access it
- Each process has a parent
- We use forking to create children
- Exit code are the simplest form of communication