Process Management

Introduction

- Process is a program in execution.
- Processes carry out tasks in a system
- A process includes program counter (PC), CPU registers and process stacks, which contains temporary data.
- Unix is a multiprocessing system
- The UNIX kernel is reentrant

Processes : Introduction

- A process uses many resources like memory space, CPU, files, etc., during its lifetime.
- Kernel should keep track of the processes and the usage of system resources.
- Kernel should distribute resources among processes fairly.
- Most important resource is CPU. In a multiprocessing environment, to attain an ideal performance of a system, the CPU utilization should be maximum.

Mode and space

- In order to run Unix, the computer hardware must provide two modes of execution
  - kernel mode
  - user mode
- Some computers have more than two execution modes
  - eg: Intel processor. It has four modes of execution.
- Each process has virtual address space, references to virtual memory are translated to physical memory locations using set of address translation maps.

Context Switch

- Execution control is changing from one process to another.
- When a current process either completes its execution or is waiting for a certain event to occur, the kernel saves the process context and removes the process from the running state.
- Kernel loads next runnable process's registers with pointers for execution.
- Kernel space: a fixed part of virtual address space of each process. It maps the kernel text and data structures.

Process structure

- Every process is represented by a task_struct data structure.
- This structure is quite large and complex.
- When ever a new process is created a new taskstruct structure is created by the kernel and the complete process information is maintained by the structure.
- When a process is terminated, the corresponding structure is removed.
- kernel maintains all information about the process using these structures in **Process control Block** or **Task Control Block**

Identifiers

- Every process in the system has a process identifier (pid)
- Every process is a child of another process so will also have a ppid (parent pid)
- Each process also has User and Group identifiers, these are used to control the process access to the files and devices in the system

  eg: ppid, pid, uid, gid, euid, egid

Process System calls

fork()

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

pid_t fork(void);
```

- When a fork() system call is made, the operating system generates a copy of the parent process which becomes the child process.
- If the fork system call is successful, the process ID of the child process is returned in the parent process and a 0 is returned in the child process.
- If the *fork* system call fails, it will return a -1.

Process System calls

- The *fork* system call does not take an argument.
- The operating system will pass to the child process most of the parent's process information. However, some information is unique to the child process:
  - The child has its own process ID (PID)
  - The child will have a different PPID than its parent

example of fork()

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(void)
        {
        printf("Hello \n");
        fork();
        printf("bye\n");
        return 0;
        }
```

example of fork()

- Hello - is printed once by parent process
  bye - is printed twice, once by the parent and once by the child

  If the *fork* system call is successful a child process is produced that continues execution at the point where it was called by the parent process.

  After the *fork* system call, both the parent and child processes are running and continue their execution at the next statement in the parent process.

  A summary of fork() return values follows:
- if *(fork()==-1)* then fork() failed and there is no child
- if *(fork()>0)* then this is the parent
- if *(fork()==0)* then this is the child

Process System calls

- **wait()**

  #include <sys/types.h>
  #include <sys/wait.h>

pid_t wait(int *status);

- A parent process usually needs to synchronize its actions by waiting until the child process has either stopped or terminated its actions.

  The *wait()* system call allows the parent process to suspend its activities until one of these actions has occurred.

Process System calls

- A few additional notes about *fork()*:
  - an orphan is a child process that continues to execute after its parent has finished execution (or died)
  - to avoid this problem, the parent should execute: *wait(&return_code);*
- The *wait()* system call accepts a single argument, which is a pointer to an integer and returns a value defined as type pid_t.
- If the calling process does not have any child associated with it, wait will return immediately with a value of -1.
- If any child processes are still active, the calling process will suspend its activity until a child process terminates.

Example of wait()

```
int status;
int pid;
pid = fork();
if (pid == 0)           /* child process */
        {
        printf("\n I'm the child!");
        exit(0);
        }
else                    /* parent process */
        {
        wait(&status);
        printf("\n I'm the parent!")
        printf("\n Child returned: %d\n", status)
        }
```

Process System calls

- **getpid()**

  #include <sys/types.h>

  #include <unistd.h>

  pid_t getpid(void);

  pid_t getppid(void);

- *getpid()* returns the process id of the current process. The process ID is a unique positive integer identification number given to the process when it begins executing.
- *getppid()* returns the process id of the parent of the current process. The parent process forked the current child process.