COSC1112/1114: Operating Systems Principles

Tutorial 02 (week 03)

1. Describe the differences among short-term, medium-term and long-term scheduling.

Answer:

- Short-term (CPU scheduler)—selects from jobs in memory those jobs that are ready to execute and allocates the CPU to them.
- Medium-term—used especially with time-sharing systems as an intermediate scheduling level. A swapping scheme is implemented to remove partially run programs from memory and reinstate them later to continue where they left off.
- Long-term (job scheduler)—determines which jobs are brought into memory for processing.

The primary difference is in the frequency of their execution. The short-term must select a new process quite often. Long-term is used much less often since it handles placing jobs in the system and may wait a while for a job to finish before it admits another one.

2. Describe the actions taken by a kernel to context-switch between processes.

Answer:

In general, the operating system must save the state of the currently running process and restore the state of the process scheduled to be run next. Saving the state of a process typically includes the values of all the CPU registers in addition to memory allocation. Context switches must also perform many architecture-specific operations, including flushing data and instruction caches.

3. Including the initial parent process, how many processes are created by the following program:

```
#include <stdio.h>
#include <unistd . h>
#include <stdlib.h>

int main ( ) {
        /* for a child process */
        fork ( ) ;
        /* fork another child process */
        fork ( ) ;
        /* and fork another */
        fork ( ) ;
        return EXIT_SUCCESS;
}
```

Answer:

8 processes are created. Add printf() statements to better understand how many processes have been created. After every fork both the client and the parent execute all the remaining forks.

4. Explain the output of line A of the following program:

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

int value = 5;

int main ( ) {
        pid_t pid ;
        /* for a child process */
        pid = fork ( ) ;
        if (pid == 0) { /* chi ld */
                value += 15;
                return EXIT_SUCCESS;
        } else if ( pid >0) { /* parent process */
                wait (NULL) ;
                printf ( "PARENT: value = %d\n" , value ) ; /* line A */
                return EXIT_SUCCESS;
        }
}
```

Answer:

The result is still 5 as the child updates its copy of value. When control returns to the parent, its value remains at 5.

5. Explain the role of the init process on UNIX and Linux systems in regards to process termination.

Answer:

When a process is terminated, it briefly moves to the zombie state and remains in that state until the parent invokes a call to wait(). When this occurs, the process id as well as entry in the process table are both released. However, if a parent does not invoke wait(), the child process remains a zombie as long as the parent remains alive. Once the parent process terminates, the init process becomes the new parent of the zombie. Periodically, the init process calls wait() which ultimately releases the pid and entry in the process table of the zombie process.

6. Explain the circumstances when the line of code marked printf("LINE J") is reached.

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

int main(){
        pid_t pid;
        /* fork a child process */
        pid = fork();

        if (pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed");
                return 1;
        }
        else if (pid == 0) { /* child process */
                execlp("/bin/ls","ls",NULL);
                printf("LINE J");
        }
        else { /* parent process */
                /* parent will wait for the child to complete */
                wait(NULL);
                printf("Child Complete");
        }
        return 0;
}
```

Answer:

The call to execlp () replaces the address space of the process with the program specified as the parameter to execlp(). If the call to execlp () succeeds, the new program is now running and control from the call to execlp () never returns. In this scenario, the line printf("Line J"); would never be performed. However, if an error occurs in the call to execlp (), the function returns control and therefor the line printf("Line J"); would be performed.