Operating Systems
Assignment 1
Department of Mathematics and Computer Science,
Lehman College, the City University of New York, spring 2018

Due by March 22nd (Thursday), 2018 (submit answers as hard copy)

Preparation

For this assignment, you need to get familiar with a Linux system (Ubuntu, Fedora, CentOS, etc.). There are two options to run Linux on your Desktop (or Laptop) computer:

1. Run a copy of a virtual machine freely available on-line in your own PC environment using *Virtual Box*. Create your own Linux virtual machine *(Linux as a guest operating system)* in Windows. You may consider installing MS Windows as a guest OS in Linux host. OS-X can host Linux and/or Windows as guest Operating systems by installing Virtual Box.
2. Or on any hardware system that you can control, you need to install a Linux system (Fedora, CentOS Server, Ubuntu, Ubuntu Server).

Tasks to Be Performed

- **Study homework assignments 1, 2 and 3 (You do not need to submit homework assignments)**
- Write the programs listed (questions from Q1 to Q8), and answer the questions.
- Answer the questions (from Q9 to Q13)

Reference for the Assignment

Some useful shell commands for process control

Command	Descriptions
&	When placed at the end of a command, execute that command in the background.
CTRL + "Z"	Interrupt and stop the currently running process program. The program remains stopped and waiting in the background for you to resume it.
fg [%jobnum]	Bring a command in the background to the foreground or resume an interrupted program. If the job number is omitted, the most recently interrupted or background job is put in the foreground.
bg [%jobnum]	Place an interrupted command into the background. If the job number is omitted, the most recently interrupted job is put in the background.
jobs	List all background jobs.
ps	list all currently running processes including background jobs

kill %jobnum **kill processID**	Cancel and quit a job running in the background

[Examples]

$emacs & (to run emacs in the background)

$ps -f (to list all the processes related to you on that particular terminal with: UID,PID,PPID,C, STIME,TTY,TIME,CMD)

$ps -ef (to list all the processes running on the machine with the information given above)

$ps -fu username (to list all the processes running on the machine for the specified user)

$emacs test.txt

CTRL -Z (to suspend emacs)

$jobs (to list the jobs including the suspend emacs)
[1] + Suspended emacs test.txt

$bg %1 (to put emacs back in the foreground)
Instead, you could say

$kill %1 (to kill the emacs process)

System calls for process control

fork():

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

- The fork system call does not take an argument.
- The process that invokes the fork() is known as the **parent** and the new process is called the **child.**
- If the fork system call fails, it will return a -1.
- 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 newly created child process.
    - 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.
- When a fork system call is made, the operating system generates a copy of the parent process which becomes the child process.
- 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, for example
    - The child has its own process ID (PID)
    - The child will have a different PPID (Parent PID) than its parent

wait()

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

```
pid_t wait(int *status); //or NULL
```

- 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 until a child process terminates.
  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.
- Several macros are available to interpret the information. Two useful ones are:
  - WIFEXITED evaluates as true, or 0, if the process ended normally with an exit or return call.
  - WEXITSTATUS if a process ended normally you can get the value that was returned with this macro.

exec family of functions

```
#include <unistd.h>

extern char **environ;

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg , ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

- The exec family of functions replaces the current process image with a new process image.
- Commonly a process generates a child process because it would like to transform the child process by changing the program code the child process is executing.
- The text, data and stack segment of the process are replaced and only the user area of the process remains the same (user area: which holds the registers and other information about the process, see `<sys/user.h>`).
- If successful, the **exec** system calls do not return to the invoking program as the calling image is lost.
- It is possible for a user at the command line to issue an exec system call, but it takes over the current shell and terminates the shell.
  Example:

```
$exec ls
```

The versions of exec are:

(a) execl
(b) execv
(c) execle
(d) execve
(e) execlp
(f) execvp

The naming convention

- 'l' indicates a list arrangement (a series of null terminated arguments)
- 'v' indicate the array or vector arrangement (like the argv structure).
- 'e' indicates the programmer will construct (in the array/vector format) and pass their own environment variable list
- 'p' indicates the current PATH environment variable should be used when the system searches for executable files.
- NOTE: In the four system calls where the PATH environment variable is not used (execl, execv, execle, and execve) the path to the program to be executed must be fully specified.

exec system call functionality

Call Name	Argument List	Pass Current Environment Variables	Search PATH automatic?
execl	list	yes	no
execv	array	yes	no
execle	list	no	no
execve	array	no	no
execlp	list	yes	yes
execvp	array	yes	yes

execlp

this system call is used when the number of arguments to be passed to the program to be executed is known in advance

execvp

this system call is used when the numbers of arguments for the program to be executed is dynamic

Things to remember about exec*:

- this system call simply replaces the current process with a new program -- the PID does not change
- the exec() is issued by the calling process and what is exec'ed is referred to as the new program -- not the new process since no new process is created
- it is important to realize that control is not passed back to the calling process unless an error occurred with the exec() call
- in the case of an error, the exec() returns a value back to the calling process
- if no error occurs, the calling process is lost

Examples of valid exec commands:

```
execl("/bin/date","",NULL); // since the second argument is the program name,
                            // it may be null

execl("/bin/date","date",NULL);

execlp("date","date", NULL); //uses the PATH to find date (Use $echo $PATH)
```

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.

getpgrp()

```
#include <unistd.h>
pid_t getpgrp(void);
```

- Every process belongs to a process group that is identified by an integer process group ID value.
- When a process generates a child process, the operating system will automatically create a process group.
- The initial parent process is known as the process leader.
- getpgrp() will obtain the process group id.

[Q.1] Program 1 (fork_wait.c)

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

int main(){

 int status;
 pid_t fork_return;

 fork_return = fork();

 if (fork_return == 0) /* done by child process */
  {
    printf("\n I am the child process!");
    printf("\n my process id is: %d", getpid());
    printf("\n my parent process id is: %d", getppid());
    exit(0);
  }
 else /* done by parent process */
  {
    wait(&status);
    printf("\n I am the parent process");
    printf("\n my process id is: %d", getpid());
```

```
    printf("\n my parent process id is: %d", getppid());
    if (WIFEXITED(status))
    printf("\n Child returned: %d\n", WEXITSTATUS(status));
   }
}
```

(a) Complete the program and compile it and run the program (understand the program)

```
$gcc -o fork_wait fork_wait.c
And run it
$./fork_wait
```

(b) Before you run the program, list the current processes by the following command:

```
$ps
```

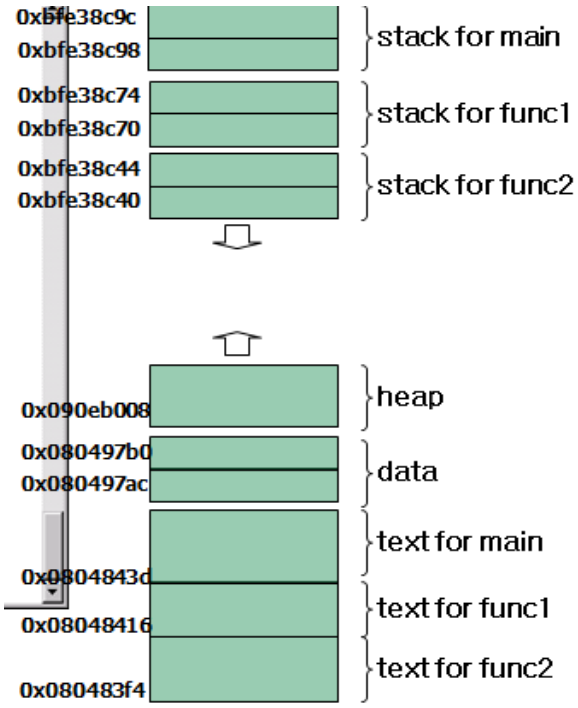
Run the program again, and explain the messages printed to the terminal by the program.

[Q.2] Program 2 (task_struct.c)

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

int glob1, glob2;

int func2(){
  int f2_local1, f2_local2;
  printf("func2_local:\n\t%p, \n\t%p\n", &f2_local1, &f2_local2);
}

int func1(){
  int f1_local1, f1_local2;
  printf("func1_local:\n\t%p, \n\t%p\n", &f1_local1, &f1_local2);
  func2();
}

main(){
  int m_local1, m_local2;
  int *dynamic_addr;

  printf("main_local:\n\t%p, \n\t%p\n", &m_local1, &m_local2);
  func1();

  dynamic_addr = malloc(16);
  printf("dynamic: \n\t%p\n", dynamic_addr);
  printf("global:\n\t%p, \n\t%p\n", &glob1, &glob2);
  printf("functions:\n\t%p, \n\t%p, \n\t%p\n", main, func1, func2);
}
```

(a) Compile and run the program and see the memory regions (run time stack, heap, data, text (or code) of the process (task).

Refer to the following figure (you will have different address values). Note: memory allocated for run time stack grows from high memory to low memory (heap is growing in reversed direction).



[Q.3] Program 3 (fork_test.c)

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

int g = 2;

int main(void)
{
        pid_t pid;
        int l = 3;

        printf("process id(%d): parent g=%d, l=%d\n", getpid(), g, l);

        if((pid=fork())<0){
                perror("fork error");
                exit(1);
        } else if(pid == 0) {
```

```
            g++;
            l++;
            printf("I am a new child and my procedd id is: %d \n", getpid());
            printf("My parent process id is: %d \n", getppid());
            printf("\n The child process now terminates");
        } else {
            g = g * 100;
            l = l * 300;
            printf("I am the PARENT process and my procedd id is: %d \n", getpid());
            printf("Parent process id of the Parent Process is: %d \n", getppid());
            printf("\n The parent process now terminates");
        }

        printf("\n\n ....Who Am I ?? (%d): g=%d, l=%d\n", getpid(), g, l); //statement A
        printf("\n\n ....Who is my parent ?? (%d): ", getppid()); //statement B

        return 0;
}
```

Compile and run fork_test.c. Answer the following questions.

(a) What are the process IDs of the parent and the child processes?
(b) Type ps from the shell prompt and identify process ID of the nonlogin(login) shell. What is the parent process ID of the parent process?
(c) What are the values of the variables g and l of the parent process and the child process created by the fork(), respectively?
(d) Show the output printed by the statements A and B, explain why you get the result.

[Q.4] Program 4 (clone.c)

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

int g=2;

int sub_func(void *arg)
{
        g = g* 100;
        printf("I am a new child as a thread: my process id is: %d  g=%d\n", getpid(), g);
        printf("My parent process id is: %d \n", getppid());
        // return 0;
}

int main(void)
```

```
{
        int l = 3;

        printf("PID(%d) : Parent g=%d, l=%d\n", getpid(), g, l);

        int *dynamic = malloc(4096 * sizeof(int)); //statement A
        clone (sub_func, (void *)(dynamic + 4095), CLONE_VM | CLONE_THREAD |
CLONE_SIGHAND, NULL); //statement B

        sleep(1);

        printf("PID(%d) : Parent g=%d, l=%d\n", getpid(), g, l);
        return 0;
}
```

(a) Compile (with or without #include <linux/sched.h>), and briefly explain why you may have compile errors/warnings without #include <linux/sched.h>
(b) After successfully compile the program, execute ./clone
(c) Show the values of g and l printed by the parent process and the child process (by the clone() system call). Explain why you get those values.

[Q.5] Program 5 (fork_exec.c)

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

int main(void)
{
    pid_t pid;
    int exit_status;

    if((pid=fork())<0){
            perror("fork error");
            exit(1);
    } else if(pid == 0) {
            printf("Hi\n");
            execl("./fork", "fork", "-l", (char *)0);
            printf("Can I print out after execl....?? \n");
    } else {
            pid = wait(&exit_status);
            printf("\n\nI am the parent of fork_exec and my process id is: %d\n",
getpid());
    }

    return 0;
```

```
}
```

(a) Compile and run the program and show what will be the execution result of the process.
(b) Explain how the program performs the task.

[Q.6] Program 6 (forkPractice.c)

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

int value = 5;

int main()
{
        pid_t pid;
        pid = fork();

        if (pid == 0) { /* child process */
                value += 15;
                return 0;
        }
        else if (pid > 0) { /* parent process */
                wait(NULL);
                printf ("PARENT: value = %d\n",value); /* LINE A */
                return 0;
        }
}
```

(a) What output will be shown at Line A? Explain why.
(b) Show how to make new child process orphaned, and show the parent process of the orphaned child process.

[Q.7] Program 7 (shm-posix-producer.c and shm-posix-consumer.c)

shm-posix-producer.c

```
/ * Simple program demonstrating shared memory in POSIX systems.
  This is the producer process that writes to the shared memory region. */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>
```

```
int main()
{
        const int SIZE = 4096;
        const char *name = "OS";
        const char *message0= "Studying ";
        const char *message1= "Operating Systems ";
        const char *message2= "Is Fun!";

        int shm_fd;
        void *ptr;

        /* create the shared memory segment */
        shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

        /* configure the size of the shared memory segment */
        ftruncate(shm_fd,SIZE);

        /* now map the shared memory segment in the address space of the process */
        ptr = mmap(0,SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
        if (ptr == MAP_FAILED) {
                printf("Map failed\n");
                return -1;
        }

        /**
         * Now write to the shared memory region.
         *
         * Note we must increment the value of ptr after each write.
         */
        sprintf(ptr,"%s",message0);
        ptr += strlen(message0);
        sprintf(ptr,"%s",message1);
        ptr += strlen(message1);
        sprintf(ptr,"%s",message2);
        ptr += strlen(message2);

        return 0;
}
```

shm-posix-consumer.c

```
/**
 * Simple program demonstrating shared memory in POSIX systems.
 *
 * This is the consumer process
```

```
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/mman.h>

int main()
{
        const char *name = "OS";
        const int SIZE = 4096;

        int shm_fd;
        void *ptr;
        int i;

        /* open the shared memory segment */
        shm_fd = shm_open(name, O_RDONLY, 0666);
        if (shm_fd == -1) {
                printf("shared memory failed\n");
                exit(-1);
        }

        /* now map the shared memory segment in the address space of the process */
        ptr = mmap(0,SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);
        if (ptr == MAP_FAILED) {
                printf("Map failed\n");
                exit(-1);
        }

        /* now read from the shared memory region */
        printf("%s",ptr);

        /* remove the shared memory segment */
        if (shm_unlink(name) == -1) {
                printf("Error removing %s\n",name);
                exit(-1);
        }

        return 0;
}
```

(a) Compile the producer and consumer by:
    a. $gcc -o shm-posix-producer shm-posix-producer.c –lrt

b. $gcc -o shm-posix-consumer shm-posix-consumer.c –lrt

(b) Run the programs as shown below and show what would be printed on your terminal
   a. ./shm-posix-producer
   b. ./shm-posix-consumer

(c) Explain how the program performs the task

[Q.8] Program 8 (unix-pipe.c)

unix-pipe.c

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

#define BUFFER_SIZE 25
#define READ_END    0
#define WRITE_END 1

int main(void)
{
        char write_msg[BUFFER_SIZE] = "Greetings";
        char read_msg[BUFFER_SIZE];
        pid_t pid;
        int fd[2];

        /* create the pipe */
        if (pipe(fd) == -1) {
                fprintf(stderr,"Pipe failed");
                return 1;
        }

        /* now fork a child process */
        pid = fork();

        if (pid < 0) {
                fprintf(stderr, "Fork failed");
                return 1;
        }

        if (pid > 0) {  /* parent process */
                /* close the unused end of the pipe */
                close(fd[READ_END]);

                /* write to the pipe */
                write(fd[WRITE_END], write_msg, strlen(write_msg)+1);
```

```
        /* close the write end of the pipe */
        close(fd[WRITE_END]);
    }
    else { /* child process */
        /* close the unused end of the pipe */
        close(fd[WRITE_END]);

        /* read from the pipe */
        read(fd[READ_END], read_msg, BUFFER_SIZE);
        printf("child read %s\n",read_msg);

        /* close the write end of the pipe */
        close(fd[READ_END]);
    }

    return 0;
}
```

(a) Compile and run the program, and show what would be printed on your terminal
(b) Explain how the program performs the task

[Q.9] Answer the following questions

(a) Explain the difference between UNIX/Linux ordinary pipe and named pipe

(b) Run the following command and see what would be printed on your terminal:
    a. $cat unix-pipe.c | wc

Note: The cat command is used to display the contents of the unix-pipe.c file, but the display is not sent to the screen; it goes through a pipe to the wc (word count) command. The wc command then counts the lines, words, and characters of the contents of the file from the pipe.

(c) Write a simple text file simple.txt that has three lines:
    a. Today = Tuesday, Is tomorrow Wednesday?, next year
    b. Show what would be printed by
        i. $ cat simple.txt | sed -e "s/next/this/g"
        ii. $ cat simple.txt | sed -e "s/t/ttt/g"

Note: The command line filter sed command changed every occurrence of the word "next" to the word "this".
The sed took as input the data through the pipe. The sed command displayed its output to the terminal. It is important to note that, the contents of the simple.txt file itself were not changed.

(d) Show what would be printed by
    a. cat simple.txt | awk -F = '{print $1}'
    b. cat simple.txt | awk -F = '{print "next " $2}'

Note: The contents of the simple.txt are sent through a pipe to the awk filter command. The awk command displays the first word on each line that comes before the = sign.

The "=" is the field delimiter. $1 represents the first field. $2 represents the second field

(e) Show what would be printed by
    a. cat simple.txt | grep next

Note: The Unix/Linux grep filter command helps you search for strings in a file.

[Q.10] Answer the following questions

(a) Explain the difference between built-in commands and external commands. What is a shell command to check which command is built-in in Linux/UNIX?
(b) Explain a way to pass the system call parameters to the kernel based on main memory segment.
(c) Explain difference(s) between interrupt driven IO and DMA.
(d) Explain why we need memory hierarchy in modern computer systems.
(e) Discuss the pros and cons of microkernel architecture.
(f) Explain a significant problem associated with many to one threading model

[Q.11] Answer the following questions based on the following program

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

int main()
{
   char x[]="/home/uname/test";
   chdir(x);
   char y[100];
   getcwd(y,100); //copies current working directory path name to buffer y
   printf("some message…. =%s\n",y); //statement A

   pid_t  pid;
   pid = fork();
   if (pid < 0) {
     printf("some other message…\n"); //statement B
      exit(-1);
   }
   else if (pid == 0) {
     printf("some other message… before execlp() system call\n"); //statement C
     execlp("/bin/ls", "ls", NULL);
     printf("some other message… after execlp() system call\n"); //statement D

   }
   else {
     printf ("Another message\n"); //statement E
     exit(0);
```

```
  }
}
```

(a) Let us assume the program is saved in a file named q11.c. Show the command to compile q11.c and generate q11 as an executable file in Linux. Use GNU c compiler.

(b) If you remove #include <unistd.h> and #include <stdlib.h> statements from q1.c. There will be compilation errors and/or warnings. Identify which statement(s) may cause the compilation problem(s).

(c) How many children processes are created? Why do we need to have if (pid < 0) clause in the program? Is the if(pid<0) clause executed by a child process? Argue accordingly.

(d) Let us assume your current working directory (CWD) is /home/uname/programs, and also assume that the CWD has two files q11.c and q11. Show the command to execute program q11 in the CWD, and explain what would be the execution results.

(e) Is the statement D printed on the terminal? Argue accordingly.

(f) Explain why there is possibility of having an orphaned child process in q11.c. Which process would be the parent process of the orphaned process?

(g) Modify the program (add one statement) in such a way that there will not be any orphaned child process in q11.c.

[Q.12] Answer the following questions based on the following program

(a) In the context of the IPC in a single system boundary, explain message based communication and shared memory communication.

(b) Linux clone() system call is used to create a task as a process or a thread. Name two important flags of the clone() required to create a task as a thread.

(c) We want to make the parent and child processes (Note: not threads) to cooperate through the variable var1 as shown below. Suggest a way to share the variable var1 using the shared memory based IPC, and explain steps required to do that (in pseudo-code)

```
…….
int var1 = 10;
int main()
{
  int pid = fork();
  if (pid == 0) {
    value += 20;
   }
  else if (pid > 0){
    wait(NULL);
   }
}
```

.

(d) What are the values printed out by the statements shown in Line X and Line Y

```
//some include statements
int value = 0;
void *runner(void *param);

int main(int argc, char *argv[])
```

```
{
int pid;
pthread_t tid;
pthread_attr_t attr;
pid = fork();
if (pid==0) {
pthread_attr_init(&attr);
pthread_create(&tid,&attr,runner,NULL);
pthread_join(tid,NULL);
printf("value = %d\n", value); //Line X
}
else if (pid > 0) {
        wait(NULL);
        printf("value = %d\n", value); //Line Y
}

void *runner(void *param)
{
        value = value + 5;
        pthread_exit(0);
}
```

[Q.13] Answer the following questions in the context of process synchronization

(a) What is the busy waiting? Under what situation(s) a lock with busy waiting (e.g., spinlock, adaptive mutex) would be useful?
(b) Describe a semaphore structure without busy waiting.
(c) Explain three requirements of the solution to the critical section problem in the context of process/thread synchronization
(d) Explain what is dining philosophers' problem. If you solve the problem by semaphores, deadlock problem may occur. Explain why?
(e) To handle synchronization by a monitor construct, you may need condition variables.
    a. Explain what are the condition variables
    b. Explain the difference between "signal and wait" and "signal and continue"