**Operating Systems**
Assignment 1
*Department of Mathematics and Computer Science,*
Lehman College, the City University of New York, Spring 2017

*Due by February 22 (Wednesday), 2017 (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**

- Write the following programs (Program 1 to Program 9) as explained below. Show how to compile and run the programs by typescript (`$script outfile`) including source program listings (e.g., `$cat –n prog.c`). When a scripting session is completed, use exit command to exit from the current scripting session. Use "ls" command to make sure scripting file (e.g., outfile is created in your current working directory)
- Write a document including typescript and answers to the questions, and submit your answer in hard copy. Note: You should have a cover page to include your name and submission date.

**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.<br>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.<br>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**<br>**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:**

- execl
- execv
- execle
- execve

- execlp
- 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.

**Program 1 (fork_wait.c)**

```
//add include directives

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!");
      exit(0);
    }
  else /* done by parent process */
    {
      wait(&status);
      printf("\n I am the parent process");
      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)

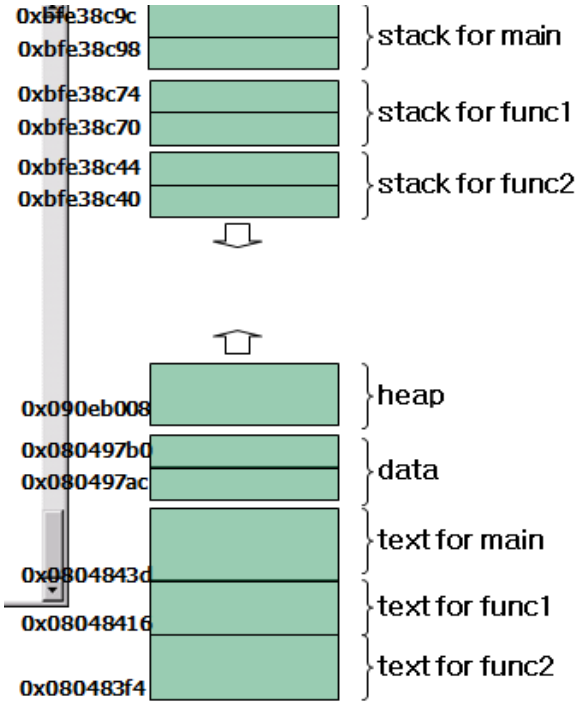
$gcc -o fork_wait fork_wait.c
And run it
$./fork_wait

**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: stack area is growing from high memory to low memory (heap is growing in reversed direction).




**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 of fork, respectively?
(d) Show the output printed by statements A and B, explain why you get the result.

**Program 4 (clone.c)**

```
#include <sys/types.h>
#include <unistd.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>)
(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.
(d) Rewrite the program by changing dynamic + 4095 to dynamic (without 4095) in statement B, and compile and run the program. Show whether you can compile and run the program. If there is any runtime error, explain why you have runtime error (core dump). //To see the core file dumped, do $ulimit -c unlimited from yor command shell
(e) Declare the following array at the beginning of the main function body.

```
int child_stack[4096];
```

Also replace statements A and B by the statement shown below.

```
clone (sub_func, (void *)(child_stack + 4095), CLONE_VM | CLONE_THREAD |
CLONE_SIGHAND, NULL);
```

- compile and run the program, show the result

**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);
```

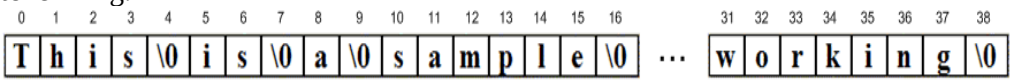
```
            printf("\n\nI am the parent of fork_exec and my process id is: %d\n",
    getpid());
        }

        return 0;
    }
```

(a) Compile and run the program and show what will be the execution result of the process.

**Splitting C Strings into Tokens (reference for the program 6)**

Sometimes you may want to split a line into tokens or words. To do that, there is a C String function called `strtok`.
The function prototype is:
`char * strtok (char * str, const char * delimiters)`
where `str` is the line (or C string) that you want to split into tokens or words, and delimiters are an array of characters in which any one of the characters delimits or marks the boundaries between words.

- In the first call to strtok, the first argument is the line or C string to be tokenized; in subsequent calls to strtok, the first argument is NULL.
    - token=strtok(cstr1, delim)
    - token=strtok(NULL, delim)
- The original C string is modified when it is tokenized so that delimiters are replaced by null terminators ('\0'). The following represents what a C string will look like after tokenizing:




**Program 6 (simple_shell.c)**

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAXLINE 80
#define WHITE " "
#define MAXARG 20

int main()
{
  char cmd[MAXLINE];
  void background(char *cmd);

  for(;;)
   {
     printf("mysh ready$$"); /* prompting mysh reddy $$ */
```

```
    gets(cmd); /* read a command */
    if (strcmp(cmd, "exit") == 0)
        return(0);
    background(cmd); /* start a background job */
  }

}

void background(char *cmd)
{
 char *argv[MAXARG];
 int pid, i = 0;
 /* to fill in argv */
 argv[i++] = strtok(cmd, WHITE);
 while (i < MAXARG && (argv[i++] = strtok(NULL, WHITE)) != NULL);
  if ( (pid = fork()) == 0) /* The child process executes background job */
        {
         execvp(argv[0], argv); //statement A
         // execl("/bin/ls", "ls", "-a", "-l", 0);
         exit(1); /* execv failed */
      }
         else
        if (pid < 0) {

            fprintf(stderr, "fork failed \n");
            perror("background");
         }
}
```

(a) Compile and run the program, after you get "mysh ready$$" shell prompt execute several commands (e., ls, gedit text1.txt, cat text1.txt, ctc..)

(b) Change execvp to execv, and compile and run. Explain why you do not get the shell functionality.