LAB NO: 4 **Date:**

PROCESS AND THREAD MANAGEMENT

Objectives:

1. Understand the working of the different system calls.
2. Understand the creation of new processes with fork and altering the code space of process using exec.
3. Understand the concepts of the multithreading.
4. Grasp the execution of the different processes with respect to multithreading.

1. Executing a C program on UNIX platform

Compile/Link a Simple C Program - hello.c

Below is the Hello-world C program hello.c:

```
// hello.c
#include <stdio.h>
int main() {
   printf("Hello, world!\n");
   return 0;
}
```

To compile the hello.c:

```
> gcc hello.c          // Compile and link source file hello.c into executable a.out
```

The default output executable is called "a.out".

To run the program:

```
$ ./a.out
```

In Bash or Bourne shell, the default PATH does not include the current working directory. Hence, you may need to include the current path (./) in the command. (Windows include the current directory in the PATH automatically; whereas UNIXes do not - you need to include the current directory explicitly in the PATH.)

To specify the output filename, use -o option:

```
> gcc -o hello hello.c                    // Compile and link source file hello.c into executable hello

$ ./hello     // Execute hello specifying the current path (./)
```

2. Use of System Calls in C programming

The main system calls that will be needed for this lab are:

- fork()
- execl(), execlp(), execv(), execvp()
- wait()
- getpid(), getppid()
- getpgrp()

fork()

A new process is created by calling fork. This system call duplicates the current process, creating a new entry in the process table with many of the same attributes as the current process. The new process is almost identical to the original, executing the same code but with its own data space, environment, and file descriptors. Combined with the **exec** functions, **fork** is all we need to create new processes.

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

The return value of fork() is pid_t (defined in the header file sys/types.h). As seen in the Fig. 4.1, the call to fork in the parent process returns the PID of the new child process. The new process continues to execute just like the parent process, with the exception that in the child process, the PID returned is 0. The parent and child process can be determined by using the PID returned from fork() function. To the parent the fork() returns the PID of the child, whereas to the child the PID returned is zero. This is shown in the following Fig. 4.1.

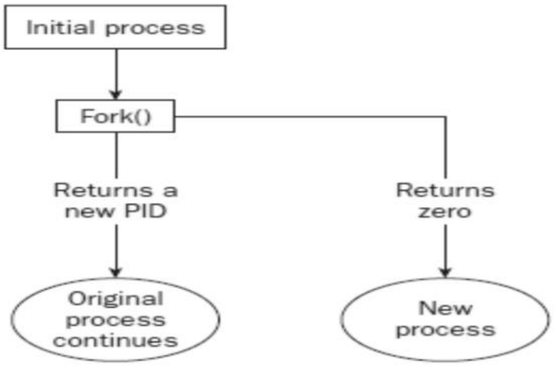

Figure 4.1: Fork system call

In Linux in case of any error observed in calling the system functions, then a special variable called errno will contain the error number. To use errno a header file named errno.h has to be included in the program. If fork fails, it returns -1. This is commonly due to a limit on the number of child processes that a parent may have (CHILD_MAX), in which case errno will be set to EAGAIN. If there is not enough space for an entry in the process table, or not enough virtual memory, the errno variable will be set to ENOMEM.

A typical code snippet using fork is

```
pid_t new_pid;
new_pid = fork();
switch(new_pid) {
case -1 : /* Error */
break;
case 0 : /* We are child */
break;
default : /* We are parent */
break;
}
```

Sample Program on fork1.c

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main()
{
        pid_t pid;
        char *message;
        int n;
        printf("fork program starting\n");
        pid = fork();
        switch(pid)
        {
                case -1:
                        perror("fork failed");
                        exit(1);
                case 0:
                        message = "This is the child";
                        n = 5;
                        break;
                default:
                        message = “This is the parent”;
                        n = 3;
                        break;
            }
            for(; n > 0; n--) {
                        puts(message);
                        sleep(1);
            }
            exit(0);
}
```

This program runs as two processes. A child process is created and prints a message five times. The original process (the parent) prints a message only three times. The parent process finishes before the child has printed all of its messages, so the next shell prompt appears mixed in with the output.

```
$ ./fork1
fork program starting
This is the parent
This is the child
This is the parent
This is the child
This is the parent
This is the child
This is the child
This is the child
```

When fork is called, this program divides into two separate processes. The parent process is identified by a nonzero return from fork and is used to set a number of messages to print, each separated by one second.

The wait() System Call

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.
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():

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

  void main()
  {
            int status; pid_t
            pid;
            pid = fork();
            if(pid = -1)
              printf("\nERROR child not created");
             else 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)
            }
}
```

A few notes on this program:

wait(&status) causes the parent to sleep until the child process has finished execution. The exit status of the child is returned to the parent.

The exit() System Call

This system call is used to terminate the current running process. A value of zero is passed to indicate that the execution of process was successful. A non-zero value is passed if the execution of process was unsuccessful. All shell commands are written in C including grep. grep will return 0 through exit if the command is successfully runs (grep could find pattern in file). If grep fails to find pattern in file, then it will call exit() with a non-zero value. This is applicable to all commands.

The exec() System Call

The exec function will execute a specified program passed as argument to it, in the same process (Fig. 4.2). The exec() will not create a new process. As new process is not created, the process ID

(PID) does not change across an execute, but the data and code of the calling process are replaced by those of the new process.

fork() is the name of the system call that the parent process uses to "divide" itself ("fork") into two identical processes. After calling fork(), the created child process is actually an exact copy of the parent - which would probably be of limited use - so it replaces itself with another process using the system call exec().

The versions of exec are:

- execl
- execv
- execle
- execve
- execlp
- execvp

The naming convention: exec*

- '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 string should be used when the system searches for executable files.

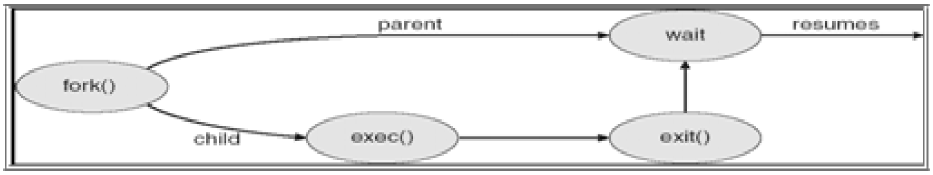

Figure 4.2: exec() system call

The parent process can either continue execution or wait for the child process to complete. If the parent chooses to wait for the child to die, then the parent will receive the exit code of the program that the child executed. If a parent does not wait for the child, and the child terminates before the parent, then the child is called **zombie** process. If a parent terminates before the child process then the child is attached to a process called init (whose PID is 1). In this case, whenever the child does not have a parent then child is called **orphan** process.

Sample Program:

C program forking a separate process.

```
#include<sys/types.h>
#include<stdio.h>
#include<unistd.h>
int main()
{
        pid_t  pid;
        /* fork another process */
        pid = fork();
        if (pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed");
                exit(-1);
        }
        else if (pid == 0) { /* child process */
                execlp("/bin/ls", "ls", NULL);
        }
        else { /* parent process */
        /* parent will wait for the child to complete */
                wait (NULL);
                printf ("Child Complete");
                exit(0);
        }
}
```

execl: is used with a list comprising the command name and its arguments:

int execl(const char *path, const char *arg0, …../*, (char *) 0 */);

This is used when the number of arguments are known in advance. The first argument is the pathname which could be absolute or a relative pathname, The arguments to the command to run are represented as separate arguments beginning with the name of the command (*arg0). The ellipsis representation in the syntax (…/*) points to the varying number of arguments.

Example: How to use execl to run the wc –l command with the filename foo as argument:

execl (“/bin/wc”, “wc”, “-l”, “foo”, (char *) 0);
execl doesn’t use PATH to locate wc so pathname is specified as the first argument.

execv: needs an array to work with.

int execv(const char *path, char *const argv[ ]);

Here path represents the pathname of the command to run. The second argument represents an array of pointers to char. The array is populated by addresses that point to strings representing the command name and its arguments, in the form they are passes to the main function of the program to be executed. In this case also the last element of the argv[ ] must be a null pointer.

Here the following program uses execv program to run grep command with two options to look up the author’s name in /etc/passwd. The array *cmdargs[ ] are populated with the strings comprising the command line to be executed by execv. The first argument is the pathname of the command:

```
#include<stdio.h>
int main(int argc, char **argv){
char *cmdargs[ ] = {“grep”, “-I”, “-n”, “SUMIT”, “/etc/passed”, NULL};
execv(“/bin/grep”, cmdargs);
printf (“execv error\n”);
}
```

Drawbacks:

Need to know the location of the command file since neither execl nor execv will use PATH to locate it. The command name is specified twice - as the first two arguments. These calls can’t be used to run a shell script but only binary executable. The program has to be invoked every time there is a need to run a command.

execlp and execvp: requires pathname of the command to be located. They behave exactly like their other counterparts but overcomes two of the four limitations discussed above. First the first argument need not be a pathname it can be a command name. Second these functions can also run a shell script.

int execlp(const char *file, const char *arg0, …./*, (char *) 0 */);
int execvp(const char *file, char *const argv[ ]);

execlp (“wc”, “wc”, “-l”, “foo”, (char *) 0);

execle and execve: All of the previous four exec calls silently pass the environment of the current process to the executed process by making available the environ[ ] variable to the overlaid process. Sometime there may be a need to provide a different environment to the new program - a restricted shell for instance. In that case these functions are used.

```
int execle(const char *path, const char *arg0, … /*, (char *) 0, char * const envp[ ] */);
int execve(const char *path, char * const argv[ ], char *const envp[ ]);
```

These functions unlike the others use an additional argument to pass a pointer to an array of environment strings of the form variable = value to the program. It's only this environment that is available in the executed process, not the one stored in envp[ ].

The following program (assume fork2.c) is the same as fork1.c, except that the number of messages printed by the child and parent processes is reversed. Here are the relevant lines of code:

```
switch(pid)
{
        case -1:
                perror("fork failed");
                exit(1);

        case 0:
                message = "This is the child";
                n = 3;
                break;
        default:
                message = "This is the parent";
                n = 5;
                break;
}
```

When the preceding program is run with ./fork2 & and then call the ps program after the child has finished but before the parent has finished, a line such as this. (Some systems may say <zombie> rather than <defunct>) is seen.

<u>exec*():</u>

```
#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." (man pages)
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 u (user) area of the process remains the same. 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.

```
% exec command  [arguments]
```

The versions of *exec* are:

- execl
- execv
- execle
- execve
- execlp
- execvp

The naming convention: exec*

- '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 string should be used when the system searches for executable files.

NOTE:

- In the four system calls where the PATH string is not used (execl, execv, execle, and execve) the path to the program to be executed must be fully specified.

Library Call Name	**Argument Type**	**Pass Cu r-rent Environ-ment Variables**	**Search PATH aut o-matic?**
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

```
/* using execvp to execute the contents of argv */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char *argv[]){
  execvp(argv[1], &argv[1]);
  perror("exec failure");
  exit(1);
}
```

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.

ϒ in the case of an error, the exec() returns a value back to the calling process

ϒ if no error occurs, the calling process is lost.

A few more 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, try: %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.

THREAD MANAGEMENT

A process will start with a single thread which is called main thread or master thread. Calling pthread_create() creates a new thread. It takes the following parameters.

- A pointer to a pthread_t structure. The call will return the handle to the thread in this structure.
- A pointer to a pthread attributes structure, which can be a null pointer if the default attributes are to be used. The details of this structure will be discussed later.

- The address of the routine to be executed.
- A value or pointer to be passed into the new thread as a parameter.

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

void* thread_code( void * param )
{
printf( "In thread code\n" );
}
int main()
{
pthread_t thread;
pthread_create(&thread, 0, &thread_code, 0 );
printf("In main thread\n" );
}
```

In this example, the main thread will create a second thread to execute the routine thread_code(), which will print one message while the main thread prints another. The call to create the thread has a value of zero for the attributes, which gives the thread default attributes. The call also passes the address of a pthread_t variable for the function to store a handle to the thread. The return value from the thread_create() call is zero if the call is successful; otherwise, it returns an error condition.

Thread termination:
Child threads terminate when they complete the routine they were assigned to run. In the above example child thread thread will terminate when it completes the routine thread_code().

The value returned by the routine executed by the child thread can be made available to the main thread when the main thread calls the routine pthread_join().

The pthread_join() call takes two parameters. The first parameter is the handle of the thread that is to be waited for. The second parameter is either zero or the address of a pointer to a void, which will hold the value returned by the child thread.

The resources consumed by the thread will be recycled when the main thread calls pthread_join(). If the thread has not yet terminated, this call will wait until the thread terminates and then free the assigned resources.

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

void* thread_code( void * param )
```

```
{
printf( "In thread code\n" );
}
int main()
{
pthread_t thread;
pthread_create( &thread, 0, &thread_code, 0 );
printf( "In main thread\n" );
pthread_join( thread, 0 );
}
```

Another way a thread can terminate is to call the routine pthread_exit(), which takes a single parameter—either zero or a pointer—to void. This routine does not return and instead terminates the thread. The parameter passed in to the pthread_exit() call is returned to the main thread through the pthread_join(). The child threads do not need to explicitly call pthread_exit() because it is implicitly called when the thread exits.

Passing Data to and from Child Threads
In many cases, it is important to pass data into the child thread and have the child thread return status information when it completes. To pass data into a child thread, it should be cast as a pointer to void and then passed as a parameter to pthread_create().

```
for ( int i=0; i<10; i++ )
pthread_create( &thread, 0, &thread_code, (void *)i );
```

Following is a program where the main thread passes a value to the Pthread and the thread returns a value to the main thread.

```
#include <pthread.h>
#include <stdio.h>
void* child_thread( void * param )
{
int id = (int)param;
printf( "Start thread %i\n", id );
return (void *)id;
}

int main()
{
pthread_t thread[10];
int return_value[10];
for ( int i=0; i<10; i++ )
{
```

```
pthread_create( &thread[i], 0, &child_thread, (void*)i );
}
for ( int i=0; i<10; i++ )
{
pthread_join( thread[i], (void**)&return_value[i] );
printf( "End thread %i\n", return_value[i] );
}
}
```

Setting the Attributes for Pthreads

The attributes for a thread are set when the thread is created. To set the initial thread attributes, first create a thread attributes structure, and then set the appropriate attributes in that structure, before passing the structure into the pthread_create() call.

```
#include <pthread.h>
...
int main()
{
pthread_t thread;
pthread_attr_t attributes;
pthread_attr_init( &attributes );
pthread_create( &thread, &attributes, child_routine, 0 );
}
```

Lab Exercises

1. Write a C program to create a child process. Display different messages in parent process and child process. Display PID and PPID of both parent and child process. Block parent process until child completes using wait system call.
2. Write a C program to load the binary executable of the previous program in a child process using exec system call.
3. Create a zombie (defunct) child process (a child with exit() call, but no corresponding wait() in the sleeping parent) and allow the init process to adopt it (after parent terminates). Run the process as background process and run “ps” command.
4. Write a multithreaded program that performs different sorting algorithms. The program should work as follows: the user enters on the command line the number of elements to sort and the elements themselves. The program then creates separate threads, each using a different sorting algorithm. Each thread sorts the array using its corresponding algorithm and displays the time taken to produce the

result. The main thread waits for all threads to finish and then displays the final sorted array.

5. Write multithreaded program that generates the Fibonacci series. The program should work as follows: The user will enter on the command line the number of Fibonacci numbers that the program is to generate. The program then will create a separate thread that will generate the Fibonacci numbers, placing the sequence in data that is shared by the threads (an array is probably the most convenient data structure). When the thread finishes execution the parent will output the sequence generated by the child thread. Because the parent thread cannot begin outputting the Fibonacci sequence until the child thread finishes, this will require having the parent thread wait for the child thread to finish.

Additional Exercises

1. Write a C program to simulate the unix commands: ls -l, cp and wc commands. [**NOTE:** DON'T DIRECTLY USE THE BUILT-IN COMMANDS]
2. Create a orphan process (parent dies before child – adopted by "init" process) and display the PID of parent of child before and after it becomes orphan. Use sleep(n) in the child to delay the termination.
3. Modify the program in the previous question to include wait (&status) in the parent and to display the exit return code (left most byte of status) of the child.
4. Create a child process which returns a 0 exit status when the minute of time is odd and returning a non-zero (can be 1) status when the minute of time is even.
5. Write a multithreaded program for matrix multiplication.