Micro Assignment - 02

Process Creation

fork() system call allows a process to create a new process (child of the creating process). The new process is an exact copy of the parent with a new process ID and its own process control block.

The name "fork" comes from the idea that parent process is dividing to yield two copies of itself.

```
#include <unistd.h>

pid_t fork(void);
```

In parent: returns process ID of child on success, or −1 on error;
in successfully created child: always returns 0

exit() system call terminates a process and makes all resources available for subsequent reallocation by the kernel. exit(status) provides status as an integer to denote the exiting condition. The parent could use **wait()** system call to retrieve the status returned by the child.

The following idiom is sometimes employed when calling *fork()*:

```
pid_t childPid;             /* Used in parent after successful fork()
                               to record PID of child */
switch (childPid = fork()) {
case -1:                    /* fork() failed */
    /* Handle error */

case 0:                     /* Child of successful fork() comes here */
    /* Perform actions specific to child */

default:                    /* Parent comes here after successful fork() */
    /* Perform actions specific to parent */
}
```

Just having a copy of the parent process is not very useful. The child needs to do something different from the parent to be useful. To load another program into the child process, we can use the execve(pathname, argv, envp) system call.

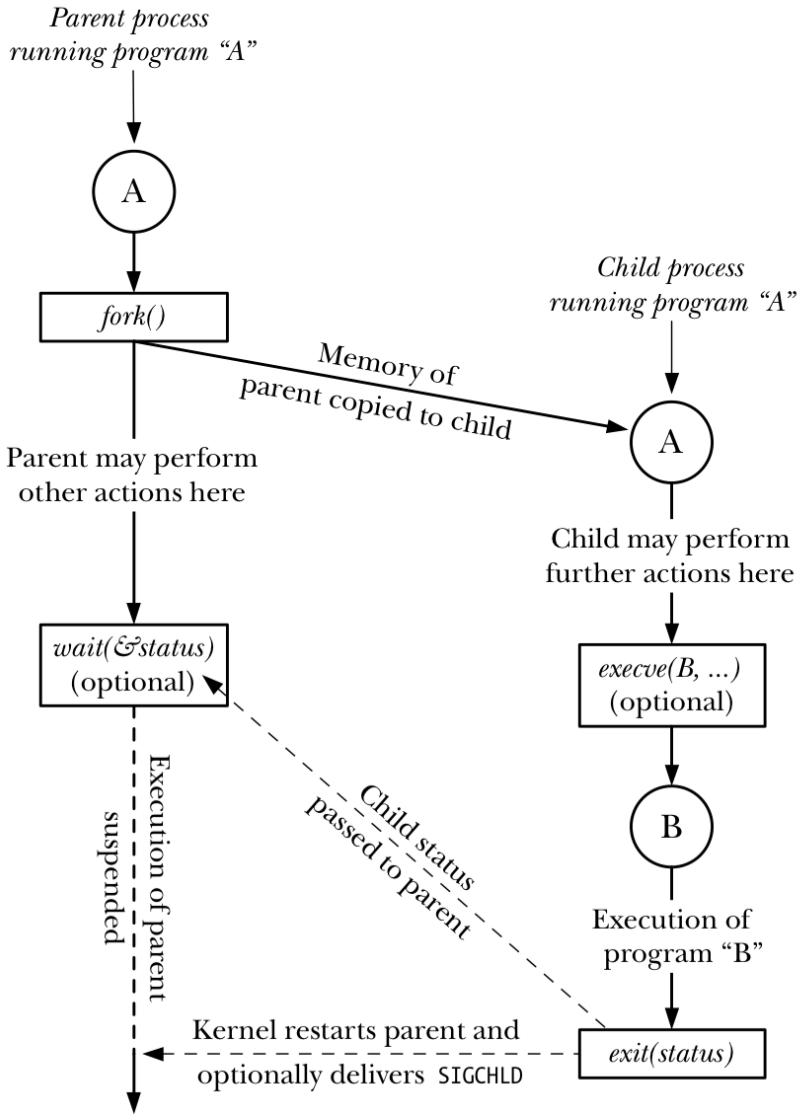

Figure 1: Flow of control with fork().

execve(pathname, argv, envp) loads a new program at pathname with argument list argv, and environment list envp into the child process' memory.

There are many variations of the "exec" system calls. You need to select the

appropriate one for your situation.

It is important to note that after fork() is called, there will be two processes executing. The order in which the CPU scheduler will allocate the CPU to the newly created process is undefined. That is it is important not to make assumptions regarding the process that will be running in CPU soon after executing the fork() – parent or child could be running.

Process Termination

A process may terminate in two different ways: abnormally due to the receipt of a signal or the execution of an _exit() system call.

```
#include <unistd.h>

void _exit(int status);
```

The status value is available for the parent process that is waiting for the termination of this process. Only the bottom 8 bits of the int is available. An exit status of 0 means successful termination.

Programs do not explicitly call _exit(status). Instead, most programs use exit(status) a library function that calls _exit(status) after performing some additional processing.

```
#include <stdlib.h>

void exit(int status);
```

The following actions are performed by *exit()*:

- Exit handlers (functions registered with *atexit()* and *on_exit()*) are called, in reverse order of their registration (Section 25.3).
- The *stdio* stream buffers are flushed.
- The *_exit()* system call is invoked, using the value supplied in *status*.

Another way a program can terminate is by returning from the main() or falling off the end. Performing return n at the end of the main() is equivalent to calling exit(n).

Falling off the end is equivalent to calling exit(0) depending on the version of C.

Exit handlers can be registered in two different ways:

```
#include <stdlib.h>

int atexit(void (*func)(void));
```

Returns 0 on success, or nonzero on error

The *atexit()* function adds *func* to a list of functions that are called when the process terminates. The function *func* should be defined to take no arguments and return no value, thus having the following general form:

```
void
func(void)
{
    /* Perform some actions */
}
```

```
#define _BSD_SOURCE           /* Or: #define _SVID_SOURCE */
#include <stdlib.h>

int on_exit(void (*func)(int, void *), void *arg);
```

Returns 0 on success, or nonzero on error

The *func* argument of *on_exit()* is a pointer to a function of the following type:

```
void
func(int status, void *arg)
{
    /* Perform cleanup actions */
}
```

Waiting for Processes

Instead of waiting for any process using wait() (see Figure 2), it is possible to wait for a specific process using the waitpid() system call (see Figure 3).

What Do You Need To Do?

Read the HANDOUT carefully and understand the process creation, termination, and process monitoring (waiting) functions supported by a UNIX/Linux operating system.

You need to develop a tiny shell which presents a command prompt and reads an input command. Uses fork() to create a child process and executes the command provided by the user in the child. If there is an error such as unable to find the command, then the error is reported to the user. The parent process waits for

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

pid_t wait(int *status);
                                Returns process ID of terminated child, or -1 on error
```

The *wait()* system call does the following:

1. If no (previously unwaited-for) child of the calling process has yet terminated, the call blocks until one of the children terminates. If a child has already terminated by the time of the call, *wait()* returns immediately.
2. If *status* is not NULL, information about how the child terminated is returned in the integer to which *status* points. We describe the information returned in *status* in Section 26.1.3.
3. The kernel adds the process CPU times (Section 10.7) and resource usage statistics (Section 36.1) to running totals for all children of this parent process.
4. As its function result, *wait()* returns the process ID of the child that has terminated.

Figure 2: Waiting for any child processes.

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

pid_t waitpid(pid_t pid, int *status, int options);
                                Returns process ID of child, 0 (see text), or -1 on error
```

The return value and *status* arguments of *waitpid()* are the same as for *wait()*. (See Section 26.1.3 for an explanation of the value returned in *status*.) The *pid* argument enables the selection of the child to be waited for, as follows:

- If *pid* is greater than 0, wait for the child whose *process ID* equals *pid*.
- If *pid* equals 0, wait for any child in the *same process group as the caller* (parent). We describe process groups in Section 34.2.
- If *pid* is less than -1, wait for any child whose *process group* identifier equals the absolute value of *pid*.
- If *pid* equals -1, wait for *any* child. The call *wait(&status)* is equivalent to the call *waitpid(-1, &status, 0)*.

Figure 3: Waiting for a specific child process.

the execution of the command by the child. Once the child returns, the parent process presents another prompt for the next command.