CS330 Intro to Processes, Forks & Exec

Highlights of this lab:

Preamble

The following is taken from page 1 of ***Interprocess Communications in Unix***:

> ***Fundamental to all operating systems is the concept of a process. While somewhat abstract, a process consists of an executing (running) program, its current values, state information, and the resources used by the operating system to manage the execution of the process. A process is a dynamic entity. In a UNIX-based operating system, at any given point in time, multiple processes appear to be executing concurrently. From the viewpoint of each of the processes involved it appears they have access to, and control of, all system resources as if they were in their own stand-alone setting. Both viewpoints are an illusion. The majority of UNIX operating systems run on platforms that have a single processing unit capable of supporting many active processes. However, at any point in time only one process is actually being worked upon. By rapidly changing the process it is currently executing, the UNIX operating system gives the appearance of concurrent process execution. The ability of the operating system to multiplex its resources among multiple processes in various stages of execution is called multiprogramming (or multitasking). Systems with multiple processing units, which by definition can support true concurrent processing are called multiprocessing.***

With the exception of some initial processes, all processes in UNIX are created by the fork system call. The initiating process is termed the parent and the newly generated process the child.

In fact, our shell is constantly forking processes.

The following is taken from http://www.tldp.org/HOWTO/Unix-and-Internet-Fundamentals-HOWTO/running-programs.html

> ***The shell is just a user process, and not a particularly special one. It waits on your keystrokes, listening (through the kernel) to the keyboard I/O port. As the kernel sees them, it echoes them to your screen. When the kernel sees an 'Enter' it passes your line of text to the shell. The shell tries to interpret those keystrokes as commands.***
>
> ***Let's say you type 'ls' and Enter to invoke the Unix directory lister. The shell applies its built-in rules to figure out that you want to run the executable command in the file /bin/ls. It makes a system call asking the kernel to start /bin/ls as a new child process and give it access to the screen and keyboard through the kernel. Then the shell goes to sleep, waiting for ls to finish.***
>
> ***When /bin/ls is done, it tells the kernel it's finished by issuing an exit system call.***

The kernel then wakes up the shell and tells it it can continue running. The shell issues another prompt and waits for another line of input.

Other things may be going on while your 'ls' is executing, however (we'll have to suppose that you're listing a very long directory). You might switch to another virtual console, log in there, and start a game of Quake, for example. Or, suppose you're hooked up to the Internet. Your machine might be sending or receiving mail while /bin/ls runs.

Shell Commands for Process Control

The following table summarizes some 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 backgrounded 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.
kill %jobnum **kill processID**	Cancel and quit a job running in the background
jobs	List all background jobs. The jobs command is not available in the Bourne shell, unless it is using the jsh shell
ps	list all currently running processes including background jobs
at time date	Execute commands at a specified time and date. The time can be entered with hours and minutes and qualified with AM or PM

Examples of usage might be the following:

`%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)

```
%vi myfile
```

`CTRL -Z` (to suspend vi)

`%jobs` (to list the jobs including the suspend vi. The output follows)

```
[1] + Suspended vim myfile
```

`%fg %1` (to put vi back in the foreground)

Instead, you could say

`%kill %1` (to kill the vi process)

System Calls for Process Control

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

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

Each of these will be covered in their own subsection

---

fork():

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

Notes:

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 child 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:

- The child has its own process ID (PID)
- The child will have a different PPID than its parent
- System imposed process limits are reset to zero
- All recorded locks on files are reset
- The action to be taken when receiving signals is different

The following is a simple example of fork()

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

int main(void)
{
   printf("Hello \n");
   fork();
   printf("bye\n");
   return 0;
}
```

Hello - is printed once by parent process
bye - is printed twice, once by the parent and once by the child

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.

A summary of fork() return values follows:

- `fork_return > 0`: this is the parent
- `fork_return == 0`: this is the child
- `fork_return == -1`: fork() failed and there is no child. See code below to see how to check errors.

Example of multiple activities PARENT AND CHILD processes

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

#define BUFLEN 10

int main(void)
{
   int i;
   char  buffer[BUFLEN+1];
   pid_t fork_return;

   fork_return  = fork( );

   if (fork_return == 0)
   {
      strncpy(buffer, "CHILD\n", BUFLEN); /*in the child process*/
      buffer[BUFLEN] = '\0';
   }
   else if(fork_return > 0)
   {
      strncpy(buffer, "PARENT\n", BUFLEN); /*in the parent process*/
      buffer[BUFLEN] = '\0';
   }
   else if(fork_return == -1)
   {
      printf("ERROR:\n");
      switch (errno)
      {
         case EAGAIN:
            printf("Cannot fork process: System Process Limit Reached\n");
         case ENOMEM:
            printf("Cannot fork process: Out of memory\n");
      }
      return 1;
   }

   for (i=0; i<5; ++i) /*both processes do this*/
   {
      sleep(1); /*5 times each*/
      write(1, buffer, strlen(buffer));
   }

   return 0;
}
```

A few notes on this program:

- The function call `sleep` will result in a process "sleeping" a specified number of seconds. It can be used to prevent the process from running to completion within one time slice.
- One process will always end before the other. If there is enough intervening time before the second process ends, the system call will redisplay the prompt, producing the last line of output where the output from the child process is appended to the end of the prompt (ie. %child)

A few additional notes about `fork()`:

- an orphan is a child process that continues to execute after its parent has finished execution (or died)
- to avoid this problem, the parent should execute: `wait(&return_code);`

---

wait()

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

pid_t wait(int *status);
```

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.
(see the programs in ***Interprocess Communication in Unix*** pg 73 and 74)

Example of wait()

```
int status;
pid_t fork_return;

fork_return = fork();

if (fork_return == 0) /* child process */
{
  printf("\n I'm the child!");
  exit(0);
}
else /* parent process */
{
  wait(&status);
  printf("\n I'm the parent!");
  if (WIFEXITED(status))
      printf("\n Child returned: %d\n", WEXITSTATUS(status));
}
```

A few notes on this program:

- `wait(&status)` causes the parent to sleep until the child process is finished execution
- details of how the child stopped are returned via the status variable to the parent. 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.

  Consult a man file for more.

---

exec*()

```
#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.

exec system call functionality

Library 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

```
/* 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
- 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

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.

References

- What happens when you run programs from the shell? from: http://www.tldp.org/HOWTO/Unix-and-Internet-Fundamentals-HOWTO/running-programs.html
- ***Interprocess Communications in Unix--The Nooks and Crannies*** by John Shapley Gray (pages 15 to 24 and Chapter 3)
- man pages