Processes in Unix

As part of process management, first need to know how to create processes.

Fork

In Unix the system call *fork* creates new processes. Fork has the following semantics:

- it creates an exact copy of the forking process
- it returns:
  - an error (-1) if unsuccessful; the global variable *errno* gives the specific failure
  - 0 to the child process
  - process id of child to the parent
- child does not share any memory with the parent
- child and parent *share* open file descriptors
- the child is said to *inherit* its environment from its parent.

Unix Process Creation Example

```
/* myfork.C */

#include <iostream>
using namespace std;
#include <sys/types.h>
#include <unistd.h>

main(int argc, char *argv[])
     /* argc -- number of arguments */
     /* argv -- an array of strings */
{
    int pid;
    int i;

    /* print out the arguments */
    cout << "There are " << argc << " arguments:\n";
    for (i = 0; i < argc; i++)
        cout << argv[i] << "\n";

    if ((pid = fork()) < 0) {
        cerr << "Fork error\n";
        exit(1);
    }
    else if (pid == 0) {
        /* child process */
        for (i = 0; i < 5; i++)
            cout << "child (" << getpid() << ") : " << argv[2] << "\n";
        exit(0);
    }
    else {
        /* parent */
        for (i = 0; i < 5; i++)
            cout << "parent (" << getpid() << ") : " << argv[1] << "\n";
        exit(0);
    }
}
```

What will be the output? Important to note the variable "i" is not shared.

exit(n) — terminates the program with a return code of n. Programs that succeed should exit with a code of 0.

```
% g++ -o myfork myfork.C
% myfork a b
There are 3 arguments:
myfork
a
b
parent (7023) : a
child (7024) : b
child (7024) : b
child (7024) : b
child (7024) : b
child (7024) : b
parent (7023) : a
parent (7023) : a
parent (7023) : a
parent (7023) : a

% myfork a b
There are 3 arguments:
myfork
a
b
parent (24722) : a
parent (24722) : a
parent (24722) : a
child (1263) : b
parent (24722) : a
child (1263) : b
parent (24722) : a
child (1263) : b
child (1263) : b
child (1263) : b
```

Exec

The system call *execve* executes a file, transforming the calling process into a new process. After a successful *exec*, there can be no return to the calling process.

Arguments to *execve(name, argv, envp)*:

name — name of the file to execute.

argv — NULL-terminated array of pointers to NULL-terminated character strings.

envp — NULL-terminated array of pointers to NULL-terminated strings. Used to pass *environment* information to the new process.

When a process first starts up (after having been started via *exec*), the startup code for a process does the following:

- makes the arguments passed to *exec* available as arguments to the main procedure in the new process.
- places a copy of *envp* in the global variable *environ*.

Exec() Example

```
/* myexec.C */

#include <iostream>
using namespace std;
#include <unistd.h>
#include <sys/wait.h>

extern char **environ;                /* environment info */

main(int argc, char **argv)
     /* argc -- number of arguments */
     /* argv -- an array of strings */
{
    char *argvNew[5];
    int pid;

    if ((pid = fork()) < 0) {
        cerr << "Fork error\n";
        exit(1);
    }
    else if (pid == 0) {
        /* child process */
        argvNew[0] = "/bin/ls";
        argvNew[1] = "-l";
        argvNew[2] = NULL;
        if (execve(argvNew[0], argvNew, environ) < 0) {
            cerr << "Execve error\n";
            exit(1);
        }
    }
    else {
        /* parent */
        wait(0);               /* wait for the child to finish */
    }
}
```

Use of *wait()* wait for the child to finish. Many variants including *waitpid()* and *wait3()*.

In addition to *execve*, Unix provides library routines that provide a more convenient interface to *execve*.

- `execl(name, arg0, arg1, arg2, ..., 0)` — used when the arguments are known in advance. 0 terminates the argument list.
- `execv(name, argv)` — where argv is the same as for *execve*.
- `execvp(name, argv)` — where argv is the same as for *execve*. The executable file is searched for in the path of directories given in the environment.

For these calls, the library routines eventually invoke *execve()* directly, handing it the global variable *environ* in place of the *envp* argument. Thus, by default, child processes *inherit* the parent's environment.

How does Unix make use of the environment?

Login procedure (hopefully) sets the environment variable *TERM* to indicate the user's terminal type. Programs that move the cursor in non-standard ways (e.g., editors like *vi*, or *emacs*) need to know the effect of different character sequences. You better get it set correctly when you first log in!

PATH variable defines the set of directories to be searched when the user types a command.

Note:

- Unix provides the mechanism for passing environments; interpretation of the "environment variables" is application specific.
- Environment useful in cases where passing arguments to commands is too cumbersome.

Note: the mechanism for passing environment information is used by the *csh* in Unix.

- *setenv* is a command processed by the shell.
- *printenv* displays your current environment.
- *source* (.e.g., source .cshrc) executes the file .cshrc within the context of the shell. Normally, commands are executed as new processes. Why is *source* needed?

Other Environments

In Java, *fork()* and *exec()* rolled into *exec()* and then use *waitfor()* (Process object).

In Win32 API, *fork()* does *fork()* and *exec()* of Unix. Then have a *WaitForSingleObject()* call.