Subsections

Process Control: `<stdlib.h>,<unistd.h>`

A ***process*** is basically a single running program. It may be a "system" program (***e.g*** login, update, csh) or program initiated by the user (textedit, dbxtool or a user written one).

When UNIX runs a process it gives each process a unique number - a process ID, `pid`.

The UNIX command `ps` will list all current processes running on your machine and will list the pid.

The C function `int getpid()` will return the pid of process that called this function.

A program usually runs as a single process. However later we will see how we can make programs run as several separate communicating processes.

Running UNIX Commands from C

We can run commands from a C program just as if they were from the UNIX command line by using the `system()` function. **NOTE:** this can save us a lot of time and hassle as we can run other (proven) programs, scripts ***etc.*** to do set tasks.

`int system(char *string)` -- where string can be the name of a unix utility, an executable shell script or a user program. System returns the exit status of the shell. System is prototyped in `<stdlib.h>`

Example: Call `ls` from a program

```
main()
{ printf("Files in Directory are:\n");
                 system("ls -l");
}
```

`system` is a call that is made up of 3 other system calls: `execl(), wait() and fork()` (which are prototyed in <unistd>)

execl()

`execl` has 5 other related functions -- see `man` pages.

`execl` stands for ***execute*** and ***leave*** which means that a process will get executed and then terminated by `execl`.

It is defined by:

```
execl(char *path, char *arg0,...,char *argn, 0);
```

The last parameter must always be 0. It is a ***NULL terminator***. Since the argument list is variable we must have some way of telling C when it is to end. The NULL terminator does this job.

where `path` points to the name of a file holding a command that is to be executed, `argo` points to a string that is the same as path (or at least its last component.

`arg1 ... argn` are pointers to arguments for the command and 0 simply marks the end of the (variable) list of arguments.

So our above example could look like this also:

```
main()
{ printf("Files in Directory are:\n");
                  execl("/bin/ls","ls", "-l",0);
}
```

fork()

`int fork()` turns a single process into 2 identical processes, known as the ***parent*** and the ***child***. On success, fork() returns 0 to the child process and returns the process ID of the child process to the parent process. On failure, fork() returns -1 to the parent process, sets errno to indicate the error, and no child process is created.

NOTE: The child process will have its own unique PID.

The following program illustrates a simple use of fork, where two copies are made and run together (multitasking)

```
main()
{ int return_value;

                  printf("Forking process\n");
                  fork();
                  printf("The process id is %d
                    and return value is %d\n",
                    getpid(), return_value);
                  execl("/bin/ls/","ls","-l",0);
                  printf("This line is not printed\n");
}
```

The Output of this would be:

```
Forking process
The process id is 6753 and return value is 0
The process id is 6754 and return value is 0
two lists of files in current directory
```

NOTE: The processes have unique ID's which will be different at each run.

It also impossible to tell in advance which process will get to CPU's time -- so one run may differ from the next.

When we spawn 2 processes we can easily detect (in each process) whether it is the child or parent since fork returns 0 to the child. We can trap any errors if fork returns a -1. ***i.e.***:

```
int pid; /* process identifier */

pid = fork();
if ( pid < 0 )
                { printf("Cannot fork!!\n");
                                exit(1);
                }
if ( pid == 0 )
                { /* Child process */ ..... }
else
                { /* Parent process pid is child's pid */
                .... }
```

wait()

`int wait (int *status_location)` -- will force a parent process to wait for a child process to stop or terminate. `wait()` return the pid of the child or -1 for an error. The exit status of the child is returned to `status_location`.

exit()

`void exit(int status)` -- terminates the process which calls this function and returns the exit `status` value. Both UNIX and C (forked) programs can read the status value.

By convention, a status of 0 means ***normal termination*** any other value indicates an error or unusual occurrence. Many standard library calls have errors defined in the `sys/stat.h` header file. We can easily derive our own conventions.

A complete example of forking program is originally titled `fork.c`:

```
/* fork.c - example of a fork in a program */
/* The program asks for UNIX commands to be typed and inputted to a string*/
/* The string is then "parsed" by locating blanks etc. */
/* Each command and sorresponding arguments are put in a args array */
/* execvp is called to execute these commands in child process */
```

```
/* spawned by fork() */

/* cc -o fork fork.c */

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

main()
{
    char buf[1024];
    char *args[64];

    for (;;) {
        /*
         * Prompt for and read a command.
         */
        printf("Command: ");

        if (gets(buf) == NULL) {
            printf("\n");
            exit(0);
        }

        /*
         * Split the string into arguments.
         */
        parse(buf, args);

        /*
         * Execute the command.
         */
        execute(args);
    }
}

/*
 * parse--split the command in buf into
 *         individual arguments.
 */
parse(buf, args)
char *buf;
char **args;
{
    while (*buf != NULL) {
        /*
         * Strip whitespace.  Use nulls, so
         * that the previous argument is terminated
         * automatically.
         */
        while ((*buf == ' ') || (*buf == '\t'))
            *buf++ = NULL;

        /*
         * Save the argument.
         */
        *args++ = buf;

        /*
         * Skip over the argument.
         */
        while ((*buf != NULL) && (*buf != ' ') && (*buf != '\t'))
            buf++;
    }

    *args = NULL;
```

```
}

/*
 * execute--spawn a child process and execute
 *           the program.
 */
execute(args)
char **args;
{
    int pid, status;

    /*
     * Get a child process.
     */
    if ((pid = fork()) < 0) {
        perror("fork");
        exit(1);

        /* NOTE: perror() produces a short  error  message  on  the  standard
           error describing the last error encountered during a call to
           a system or library function.
       */
    }

    /*
     * The child executes the code inside the if.
     */
    if (pid == 0) {
        execvp(*args, args);
        perror(*args);
        exit(1);

       /* NOTE: The execv() vnd execvp versions of execl() are useful when the
          number  of  arguments is unknown in advance;
          The arguments to execv() and execvp()  are the name
          of the file to be executed and a vector of strings  contain-
          ing  the  arguments.   The last argument string must be fol-
          lowed by a 0 pointer.

          execlp() and execvp() are called with the same arguments  as
          execl()  and  execv(),  but duplicate the shell's actions in
          searching for an executable file in a list  of  directories.
          The directory list is obtained from the environment.
        */
    }

    /*
     * The parent executes the wait.
     */
    while (wait(&status) != pid)
        /* empty */ ;
}
```

Exerises

Exercise 12727

Use `popen()` to pipe the `rwho` (UNIX command) output into `more` (UNIX command) in a C program.

Dave Marshall
1/5/1999