Processes in Unix

(References: Gray Chapters 1, 2 and 3)

Network Programming Lecture Notes by

Turhan TUNALI

1.1 Programs and Processes (Gray Chap1)

A program is an inactive static entity consisting of a set of instructions and associated data. It can be in one of the two basic formats:

- Source program: series of valid statements for a specific programming language (such as C)
- Executable program: a program that is ready to run by an OS

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

/* Display Hello World 3 times */
char *cptr = "Hello World\n"; /* static by placement */
char buffer1[25];

void main(void){
    void showit(char *);
    int i = 0; /* automatic variable */
    strcpy(buffer1, "A demonstration\n"); /* library function */
    write(1, buffer1, strlen(buffer1)+1); /* system call */
    for (; i < 3; ++i)
        showit(cptr); /* function call */
}
void showit(char *p){
    char *buffer2;
    if ((buffer2=(char *) malloc((unsigned) (strlen(p)+1)))!= NULL){
        strcpy(buffer2, p);
        printf("%s", buffer2);
        free(buffer2);
```
A **function** is a collection of declarations and statements that carries out a specific action and/or returns a value. Functions are either defined by the user or have been previously defined and made available to the user. The latter are stored in object code format in **library files**.

In Unix, the standard location for library files is the directory `/usr/lib`.

**System calls** are requests done by either functions or programs that ask the OS directly perform some work. When a system call is issued, an executing program switches from **user mode** to **kernel mode**.

The switching from user mode to kernel mode causes a certain amount of overhead and in some cases makes a system call less efficient than a library function.
Code from library files is combined with the object code from the source program at compile time on an as-needed basis. If a system call or a library function is unsuccessful, it returns a value –1 and assigns a value to an external variable called `errno` to indicate what the actual error is. `<sys/errno.h>` header file contains the defined constants for all error codes. The library function `perror` can be used to produce an error message.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
/*
   Checking errno and using perror
*/
extern int      errno;

void
main(int argc, char *argv[]) {
  int             n_char = 0, buffer[10];
  /*
   Initially these should be both 0
   */
  printf("n_char = %d \t errno = %d \n", n_char, errno);
  /*
   Display a prompt to stdout
   */
  n_char = write(1, "Enter a word ", 14);
  /*
   Use the read system call to obtain up to 10 characters from stdin
   */
  n_char = read(0, buffer, 10);
  printf("n\n\n_n_char = %d \t errno = %d \n", n_char, errno);
  /*
   If the read has failed ...
   */
  if (n_char == -1) {
    perror(argv[0]);
    exit(1);
  }
  /*
   Display the characters read
   */
  n_char = write(1, buffer, n_char);
}
```
Source files that have been compiled into an executable form to be run by the system are put into a special format called **a.out**.

In Unix, when an executable program is read into system memory by kernel it becomes a process. The system memory is divided into two regions:

- **User space**: the space where the user processes will run (in user mode)
- **Kernel space**: the space where the kernel executes and provides its services

User spaces can only access kernel space through the system calls. In this case the process is said to be in the kernel mode and the change in mode is called a **context switch**.

When residing in memory, a user process is divided into three segments:

- **Text segment**: contains the executable program code and constant data
- **Data segment**: contains the initialized data and uninitialized data segments
- **Stack segment**: used by the process for storage of variables

In addition to the text, data and stack segments, the OS also maintains a region called **u-area**. This **u area** contains information specific to the process such as open files, current directory, signal actions etc and a **system stack segment** for process use. If the process makes a system call, the stack frame information for the system call would be stored in the system stack segment. Processes normally do not have access to this area and
they have to use system calls to receive information from this area.

The address information is available to the process via referencing the external variables `etext`, `edata`, `end`.

```c
#include <stdio.h>

/*
* Displaying process segment addresses
*/
extern int      etext, edata, end;
void
main(void){
    printf("etext: %6X 	 edata: %6X 	 end: %6X 
",
        &etext, &edata, &end);
}
```

```c
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#define SHW_ADR(ID,I) printf("The id %s \t is at adr:%8X\n",ID,&I)
extern          etext, edata, end;
/*
```

**Fig. 1.6** System and process memory.
Program 1.1 modified to display identifier addresses
*
char          *cptr = "Hello World\n"; /* static by placement */
char          buffer1[25];
void
main(void){
    void        showit(char *);
    int         i = 0;                   /* automatic variable */
    /* display segment adr */
    printf("Adr etext: %8X \t Adr edata: %8X \t Adr end: %8X \n\n",
            &etext, &edata, &end);
    SHW_ADR("main", main);               /* display addresses */
    SHW_ADR("showit", showit);
    SHW_ADR("cptr", cptr);
    SHW_ADR("buffer1", buffer1);
    SHW_ADR("i", i);
    strcpy(buffer1, "A demonstration\n");   /* library function */
    write(1, buffer1, strlen(buffer1) + 1); /* system call */
    for (; i < 1; ++i)
        showit(cptr);                         /* function call */
}
void
showit(char *p){
    char           *buffer2;
    SHW_ADR("buffer2", buffer2);
    if ((buffer2=(char *)malloc((unsigned)(strlen(p)+1))) != NULL){
        strcpy(buffer2, p);
        printf("%s", buffer2);
        free(buffer2);
    } else {
        printf("Allocation error.\n");
        exit(1);
    }
}

Processes are created by *fork* system call. If the call fails, it returns a –1 and set errno to one of the error conditions. The failure might be due to

- system limit for number of processes exceeded
- there is no enough swap space

If successful, the fork returns the process ID of the child process in the parent process and it returns a 0 in the child process. This allows a process to check if it is a parent or child process.
The OS will pass to the child process most of the parent’s information however there will be some differences:

- the child has its own PID
- the parent and the child have different PPID
- system limits such as CPU time are reset
- all record locks on the files are reset
- the action to be taken when receiving signals is different

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

/*
   First example of a fork system call (no error check)
*/

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

### 1.2 Processing Environment (Gray Chap 2)

Associated with each process is a unique positive integer identification number called process ID (PID). In Solaris, process 0 is `sched`, process 1 is `init` and process 2 is `pageout`. The system call `getpid` can be used to obtain the process id. Ex:

```c
printf("My PID is %d \n", getpid( ));
```

Every process has an associated parent process ID (PPID) which can be obtained by `getppid` system call.
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

/*
 * Displaying process group ID information
 */
void
main( void )
{
  int         i;
  printf("Initial process \t PID %6d \t PPID %6d \t GID %6d\n\n", 
             getpid(), getppid(), getpgid(0));
  for (i = 0; i < 3; ++i)
    if (fork( ) == 0)                /* Generate some processes */
      printf("New process \t \t PID %6d \t PPID %6d \t GID %6d\n", 
             getpid(), getppid(), getpgid(0));
}

There is no system call to learn about the process ids of the children. Hence, a parent should save the child pid returned from the fork call if it is going to use it in the future.

Every process belongs to a process group that is identified by an integer process group id value. When the OS generates a child, it automatically generates a group. The initial parent process is known as the process leader. If a process leader is killed, then all other processes in that group are killed. The system call getpgid provides the group id.

If the parent of a process dies, the process init inherits the process, in which case process group id does not change. A process may change its process group by using the system call setpgid.
All Unix files have owner, group and other users. These users are granted read, write and execute rights. Same criterion applies to the directories with a difference that execute corresponds to examination of the directory. At system level, the permissions of a file are modified by using the chmod command.

When the OS creates files, the permissions are assigned by default which is determined by XORing the creation mask and the umask. The creation mask has fixed value 777 for executables and 666 for text files. The default umask is 022. The umask value can be changed by umask command.

Ex:
% umask
022
% umask 011
% umask
011

Unix keeps two different user ids for processes and groups:

- real user id, real group id
- effective user id, effective group id

The OS will use the real ids for things such as process accounting or sending mail and the
effective ids for to determine what additional permissions should be granted to the process.

Most of the time, real and effective ids are identical. Sometimes, the owner of a file may grant his own privileges to the executing user. This is done by setting the u-id bit in i-node. (SUID and SGID). This will tell the OS that when the program is run by some other user, the resulting process should have the privileges of the owner. That is the effective user id of the process becomes the owner of the file. The system level command id displays the current user and group ids.

In addition to the process ID information, the process environment contains file system information. For each open file, the OS uses an integer as an index to file descriptor table that is located in the uarea of the process. The file descriptor table references a system file table which is located in kernel space. The system file table maps to a system inode table that contains reference to a more complete internal description of the file. A child process receives a copy of its parent’s file descriptor table including stdin, stdout and stderr.
Fig. 2.10

The `stat` system call provides the process with a file information that can be obtained by using `ls` system level command. `lstat` will return information about the link entry.

The `chmod` system call modifies the access permissions of a file.

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#define N_BITS 3
/*
*/
Using the stat system call

```c
void main(int argc, char *argv[]){
    unsigned int i, mask = 0700;
    struct stat buff;
    static char *perm[] = {"---", "--x", "-w-", "-wx",
                          "r--", "r-x", "rw-", "rwx");
    if (argc > 1) {
        if ((stat(argv[1], &buff) != -1)) {
            printf("Permissions for %s ", argv[1]);
            for (i = 3; i; --i) {
                printf("%3s", perm[(buff.st_mode & mask) >> (i-1) * N_BITS]);
                mask >>= N_BITS;
            }
            putchar('\n');
        } else {
            perror(argv[1]);
            exit(1);
        }
    } else {
        fprintf(stderr, "Usage: %s file_name\n", argv[0]);
    }
}
```

The umask mask value, which is inherited from the parent process, may be modified by a process with *umask* system call. It changes the umask value to the integer value passed and returns the old umask value. Note that you may need to use umask twice to determine what the present umask value is.

Ex: Suppose that umask actually is 022 and the program wants to learn about this. It first arbitrarily issues a umask, say

```
umask 011
```

and the OS changes umask to 011 while returning 022 that is the old value. The program then issues a umask 022 to put the umask value back to its original.
getcwd library function may be used to find the current working directory of the process. The system call chdir changes the current working directory.

Since the system resources are finite, every process is subject to certain limits. At the command line, the limit command displays and modifies the current limits. getlimit/setlimit system calls do the same job in the programming environment.

Processes receive signals when an out of ordinary event occurs. Sources of signals may be
- Hardware (divide by zero)
- Kernel (notifying that an i/o is complete)
- Other processes (a child notifies the parent that it has terminated)
- User: Interrupt from keyboard

Upon receiving the signal, a process can take three courses of action:
- Perform the system specified default for the signal (for most signals this leads to termination of the process)
- Ignore the signal (this may not be possible all the time. For ex. SIGSTOP and SIGKILL can not be ignored)
- Catch the signal(excluding SIGSTOP and SIGKILL): The process invokes a special
signal handling routine and after its completion the process continues its execution from where it had left.

A child inherits the signal actions from its parent. However, if another executable is overlayed, such as by issuing an exec call, the all the signal catching routines are reset to their default.

Part of the processing environment of every process are the values passed to the process in the main function. These values can be from the command line or may be passed to a child process from the parent via an exec system call. These values are stored in an array called argv. The number of elements in argv array is stored in an integer argc. In programs the getopt library function can be used to learn the contents of argv.

```c
#include <stdio.h>
void
main(int argc, char *argv[ ]){
    for ( ; *argv; ++argv )
        printf("%s\n", *argv);
}
```

Each process also has access to a list of environment variables that are passed to process by its parent when it begins execution. They can be accessed in a program by using an external pointer called environ which is defined as
extern char **environ;

gennon and putenv library calls manipulate environment variables.

1.3 Using Processes (Gray Chap 3)

After the fork system call, both the parent and the child processes are running and continue their execution at the next statement after the fork. When a child is forked, one of the main purpose is to run a different program in the child process and exec precisely does this. If exec call is successful, the existing process is overlaid with a new set of program code. The text, data and stack segment of the process are replaced and only the u-area of the process remains same. The new program code begins its execution at the function main. In addition, the following actions are taken:

- Signals that were specified as being caught by the process are reset to their default action
- If the process was profiling (determining how much time is spent in individual routines), the profiling will be turned off in the overlaid process
- If the new program has its SUID bit set, the effective EUID and EGID are set accordingly.

If successful, the exec call does not return as the calling image is lost.

#include <stdio.h>
#include <sys/types.h>
Example: Suppose that a command is issued at shell. The following sequence of actions is taken:

- Shell forks a child and waits in the background while the child is executing.
- The child executes an exec to overlay the code of the current program with that of the command.
- When the command finishes, the child executes an `exit` or `return` in function `main`.
- Shell displays its prompt

Remarks:
- There is a `wait` system call that allows the shell to wait.
- The exit or return calls accept an integer argument that is made available to the parent process via an argument to the `wait` system call and the returned value is stored in the system variable named `status`.

If the user places an “&” at the end of the command the shell will not wait and return immediately with its prompt. In this case, the command will execute in the background.

Question: What happens if the user issues an `exec` at the command line?
Answer: The command will be executed and the user will be logged out!!

There are a total of six different `exec` calls. The classification comes with three factors:
- Whether the argument format is a list or an array
- Whether it automatically passes environment variables
- Whether there is automatic path search.

`execlp`
This command is used when the number of arguments to be passed to the program to be executed is known in advance.

```c
int execlp ( const char *file, const char *arg0,..,
const char *argn,
    char */*NULL*/ );
```

- `file` is a pointer to the file that contains the program code to be executed (use absolute path to be on the safe side)
- `arg0-argn` are pointers to the arguments that would be normally passed by the system to the program if it were invoked on the command line. That is `arg0` is the name of the program, `arg1` is the first parameter passed to the program etc.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int
main(int argc, char *argv[]) {
if (argc > 1) {
    execlp("/bin/cat", "cat", argv[1], (char *) NULL);
    perror("exec failure ");
    exit(1);
}
    fprintf(stderr, "Usage: %s text_file\n", *argv);
    exit(1);
}
```

`execvp`
This command is used when the number of arguments to be passed to the program to be executed is dynamic.

```
int execvp (const char *file, char *const argv[ ]);
```

- **file** is a pointer to the file that contains the program code to be executed (use absolute path to be on the safe side).
- **argv** specifies that a reference to an array of pointers to character strings should be passed.

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

**Example:**

```c
#include <stdio.h>
#include <unistd.h>
void main(void) {
    static char    *mesg[ ] = {"Fie", "Foh", "Fum"};
    int             display_msg(char *), i;
    for (i = 0; i < 3; ++i)
        (void) display_msg(mesg[i]);
    sleep(2);
}
```

```c
int display_msg(char *m) {
    char            err_msg[25];
    switch (fork( )) {
    case 0:
        execlp("/usr/bin/echo", "echo", m, (char *) NULL);
        sprintf(err_msg, "%s Exec failure", m);
        perror(err_msg);
        return (1);
    case -1:
        perror("Fork failure");
        return (2);
    ```
A process normally terminates in one of three ways:
- It issues a call to `exit` or `_exit`.
- It issues `return` in function main.
- It falls off the end of the function main ending implicitly.

The `exit` function does not return a value

```c
void exit (int-status);
```

It accepts a single parameter, an integer status value that will be returned to the parent. By convention, a 0 value is returned if the program has terminated normally, otherwise a nonzero value is returned.

Upon invocation, the exit function
1. will call, in reverse order, all functions that have been registered using the `atexit` library function

2. will call `_cleanup` function

3. will call `_exit` function

Remarks:
- Functions to be called in step 1 are registered by passing the `atexit` function the address of the function
- `_exit` does not return. It accepts an integer status value that will be made available to the parent process.

The system performs a number of housekeeping operations when terminating a process. Among them are:
- All open file descriptors are flushed and closed
- Parent is notified that the child is terminating
- The status is returned to the parent if it has issued a wait.
- All the children of the terminating process have their PPID changed to `init` (that is 1)

```c
#include <stdio.h>
#include <stdlib.h>

int main( void ){
    void            f1(void), f2(void), f3(void);
    atexit(f1);
    atexit(f2);
    atexit(f3);
    printf("Getting ready to exit\n");
    exit(0);
}
void
```
The `wait` system call allows the parent process to suspend its activity until a child has either stopped or terminated its actions.

```c
pid_t wait (int *stat_loc);
```

It 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, `wait` returns immediately with a value –1 and `errno` will be set to `ECHILD(10)`.

If there is a child, the calling process will block until child terminates upon which the status info for the child and its PID are returned to the parent. The status information is stored in `stat_loc` which either contains the exit code if the child terminated normally, or the signal number if the child terminated due to an uncaught signal.

```c
#include <stdio.h>
#include <sys/types.h>
```
If the parent does not issue a `wait` and child terminates, the child becomes a **Zombie** process
that can not be killed by any signal. When the parent of a Zombie issues a \textit{wait}, it is only then the resources of the Zombie process are recovered by the kernel.

If the parent of a Zombie process terminates, in that case the Zombie will become an Orphan and it will be inherited by init which, in turn, issue a \textit{wait}. 