Lecture #5 – Process Basics (Chapter 7)

- Main function

  When a C program is started by the kernel (via an exec function), a special startup routine is called before the main function is called. The executable program file specifies this start-up routine as the starting address for the program (done at link time).

  int main(int argc, char **argv);

- Process Termination

  Five ways for a process to terminate:

  1. Normal: return from main
  2. Normal: calling exit
  3. Normal: calling _exit
  4. Abnormal: calling abort (Chapter 10)
  5. Abnormal: terminated by a signal (Chapter 10)

  The special startup routine mentioned in the previous section is coded to call "exit" with the return value of the main function (i.e. exit(main(argc, argv)).

  void exit(int status);
  void _exit(int status);

  "exit" performs certain cleanup processing and then returns to the kernel
  "_exit" returns to the kernel immediately

  Example (to be correct always call return or exit at the end of main):

```
int main(void)
{
        printf("hello world\n");
        return(0);
}
```

  Atexit:

  You can also register up to 32 functions to be called automatically by exit.
  These functions are registered with "atexit".

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

  "exit" calls these functions in reverse order of their registration
  In addition, each function is called as many times as it was registered.

Example (atexit_test.c):

```
#include   "ourhdr.h"

static void       my_exit1(void), my_exit2(void);

int main(void)
{
   if (atexit(my_exit2) != 0)
          err_sys("can't register my_exit2");
   if (atexit(my_exit1) != 0)
          err_sys("can't register my_exit1");
   if (atexit(my_exit1) != 0)
          err_sys("can't register my_exit1");
   printf("main is done\n");
   return(0);
}

static void my_exit1(void)
{
   printf("first exit handler\n");
}

static void my_exit2(void)
{
   printf("second exit handler\n");
}
```

- Command Line Arguments

```
int main(int argc, char **argv)
{
        int     i;

        for (i = 0; i < argc; i++)
                printf("argv[%d]: %s\n", i, argv[i]);

        exit(0);
}
```

- Environment List

Each program is also passed an environment list (i.e. extern char **environ)

By convention, the environment consists of variables in form “name=value”.

Most UNIX systems provide an alternate prototype for main:

```
int     main(int argc, char **argv, char **envp);
```

Access to specific environment variables can be obtained with getenv / putenv.

Example (getenv_test.c):

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

int main(void)
{
    char    *ptr;

    if ((ptr = getenv("TERM")) == NULL)
    {
        printf("TERM not found, setting it to vt100\n");
        putenv("TERM=vt100");

        ptr = getenv("TERM");
    }

    printf("TERM is set to %s\n", ptr);
}
```

- Memory Layout of a C Program

  Pieces of a program:

  1. text segment - i.e. the executable code of the program (sometimes read-only)

  2. initialized data segment - also called data segment

  3. uninitialized data segment - also called the "bss" (block started by symbol) segment. Does not include automatic variables.

  4. stack - automatic variables, information saved each time a function is called

  5. heap - dynamic memory allocations

  Note: Figure 7.6 on page 188 for typical memory layout
  Note: You can use the "size" command to see sizes of these segments

- Shared Libraries

  Most UNIX systems support shared libraries (i.e. load them in memory once, and everyone can use them).

  This reduces on memory consumption, etc.

- Memory Allocation

  Functions for memory allocation:

  1. malloc - allocate a specified number of bytes of memory without initialization

  2. calloc - allocate space for a specified number of objects of a specified size (initialized to 0).

  3. realloc - changes the size of a previously allocated area (increase or decrease).

  ```
  void    *malloc(size_t size);
  void    *calloc(size_t nobj, size_t size);
  void    *realloc(void *ptr, size_t newsize);

  void    free(void *ptr);
  ```

  Note: Technically speaking, these functions are library functions that make use of the sbrk function call.

  Note: alloca function is very similar to malloc except it allocates memory from the stack frame of the current function instead of the heap.

- Environment Variables

  ```
  char    *getenv(const char *name);
  ```
  Returns a pointer to value associated with name or NULL if not found.

  ```
  int     putenv(const char *str);
  ```
  Places a string in form "name=value" into environment replacing an existing defn.

  ```
  int     setenv(const char *name, const char *value, int rewrite);
  ```
  Same as putenv with separate args for name and value.

  ```
  void    unsetenv(const char *name);
  ```
  Remove any defiinition of name regardless of existence.

- Setjmp and longjmp

  ```
  int     setjmp(jmp_buf env);
  void    longjmp(jmp_buf env, int val);
  ```

  We call setjmp from the location that we want to return to.

The "val" argument to longjmp allows us to longjmp from multiple locations to the same setjmp. This value is returned to setjmp.

- Getrlimit and setrlimit

```
int     getrlimit(int resource, struct rlimit *rlptr);
int     setrlimit(int resource, const struct rlimit *rlptr);

struct rlimit {
        rlim_t  rlim_cur;
        rlim_t  rlim_max;
};
```

Rules:

1. A soft limit can be changed by any process to a values <= its hard limit
2. Any process can lower its hard limit to a value >= its soft limit (not reversable).
3. Only a superuser can raise a hard limit.

Note: Infinite limits are specified by RLIM_INFINITY

Constant	Description
RLIMIT_CORE	Max size of a core file in bytes (can be 0).
RLIMI_CPU	Max CPU in seconds (SIGXCPU)
RLIMIT_DATA	Max size of data segment in bytes (sum of initialized, BSS, and heap)
RLIMIT_FSIZE	Max size of file to be created
RLIMIT_NOFILE	Max number of open files per process.
RLIMIT_NPROC	Max number of child processes per UID.
RLIMIT_RSS	Max resident size size (RSS) in bytes.
RLIMIT_STACK	Max size of the stack
RLIMIT_VMEM	Max size of the mapped address space.

Process Control (Chapter 8)

- Process Identifiers

  Every process has a unique process identifier (PID)
  PID's are non-negative integers.

  Two special processes - 0 is usually the scheduler (**swapper**), and 1 is usually **init.**

  Swapper is a system process (i.e. part of the kernel).
  Init is responsible for starting the UNIX system after the kernel has been bootstrapped.
  Init is the "grandfather" of all other UNIX processes on the system.

pid_t	getpid(void);	PID of calling process
pid_t	getppid(void);	Parent PID of calling process
uid_t	getuid(void);	Real UID of calling process
uid_t	geteuid(void);	Effective UID of calling process
gid_t	getgid(void);	Real GID of calling process
gid_t	getegid(void);	Effective GID of calling process

  Note: Most of these values can also be observed with **ps**, with options like **-ef** or **-aux**.
  Note: None of these functions has a mechanism to return an error.

- Fork

  Every process except system processes like "swapper", and the init process are started via a function call named **fork**.

  pid_t fork(void);

  Fork creates a "copy" of the current process, and returns:

0	to the child
PID of child	to the parent
-1	on error

  The child is returned a zero since it can have only one parent (i.e. using getppid).
  The parent can have multiple children but there is no function to get PID of child process.

  After fork, both the child and parent continue executing with the instruction following the fork call (i.e. concurrently).

Note: From an implementation standpoint, most current UNIX systems do not perform a complete copy of the parent process since a call to fork is often followed by an "exec" call. Instead, a **COW** (copy on write) policy is observed. In other words, parent and child share memory until one of them needs to change it, and then the page needing modification is copied before it is changed.

Example 8.1 (fork_test.c):

```
int     glob = 6;
char    buf[] = "a write to stdout";

int main(void)
{
        int     var;
        pid_t   pid;

        var = 88;

        if (write(1, buf, sizeof(buf)-1) != sizeof(buf)-1)
        {
                perror("write failed");
                exit(1);
        }

        printf("before fork\n");

        if ((pid = fork()) < 0)
        {
                perror("fork failed\n");
                exit(1);
        }
        else if (pid == 0)                      /* child */
        {
                glob++;
                var++;
        }
        else                                    /* parent */
                sleep(2);

        printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
}
```

Note: the differences in output when run interactively versus redirected to a file.

File sharing:

One characteristic of fork is that all open file descriptors in the parent are duplicated in the child (i.e. with "dup"). The parent and child share a file table entry for every open descriptor (see Figure 8.2 page 214).

Normally, either (1) the parent waits for the child to complete, and file descriptors are not an issue or (2) the parent and child go separate ways, and the parent closes descriptors that it does not need, and the child does the same.

Properties of parent copied to child:

RUID, RGID, EUID, EGID, Supplementary group ID's
Process group ID
Session ID
Controlling terminal
SUID / SGID flags
Current working directory
Root directory
File mode creation mask
Signal mask and dispositions
Close-on-exec flag for any open file descriptors
Environment
Attached shared memory segments
Resource Limits

Differences between parent and child:

Return value from fork
PID
Parent PID
Child has tms_utime, tms_stime, tms_cutime, and tms_ustime set to 0
File locks set by parent are not inherited
Pending alarms are cleared for the child
Pending signals for the child is set to the empty set

Reasons for fork to fail:

1. Too many processes on the system
2. Too many processes for our RUID

Possible uses for fork:

1. Concurrency - a process may want to duplicate itself so that the parent and child can each execute different sections of code at the same time.
2. Program execution - a process may want to execute a different program…

- Vfork

  Vfork is intended to create a new process when the purpose of the new process is to exec a new program.

  Vfork creates the new process like fork, but until it calls either exec or exit, the child runs in the address space of the parent.

  This function provides an efficiency gain over calling fork then exec, but more recent versions of UNIX have improved on fork (i.e. Copy on write), and this difference is not as significant.

  Note: vfork also guarantees the child runs first, until the child calls exec or exit. This can lead to deadlock if the child somehow waits on the parent before calling exec or exit.

  Example (vfork_test.c):

```
int     glob = 6;

int main(void)
{
        int     var;
        pid_t   pid;

        var = 88;
        printf("before vfork\n");

        if ((pid = vfork()) < 0)
        {
                perror("vfork");
                exit(1);
        }
        else if (pid == 0)
        {
                glob++;
                var++;
                _exit(0);
        }

        printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
}
```

```
$ vfork_example
before vfork
pid = 607, glob = 7, var = 89
```

- Exit

  Normal process termination:

  1. Calling return from main
  2. Calling exit
  3. Calling _exit

  Abornmal termination:

  1. Calling abort (generating a SIGABRT to the process)
  2. When a process receives certain signals.

  Note: argument to exit functions is the exit status which can be retrieved by the parent.

  Note: If parent dies before child, **init** becomes the parent process of any process whose parent terminates.

  Note: If the child terminates before the parent, then we must keep a certain amount of information about the child after termination so the parent can check the exit status.

  Note: In UNIX terms, a process that has terminated, but whose parent has not yet waited for it (by calling wait or waitpid), is called a **zombie**. (Shown as a Z state in ps).

  When a process that is a child of init (either directly or through inheritance) terminates, init calls wait to clear the zombie.

- Wait and waitpid

  When a process terminates, the parent is notified by receiving a SIGCHLD signal.

  The parent can choose to ignore this signal, or it can provide a function that is called when the signal occurs (i.e. a signal handler). The default is to ignore this signal.

  When a process calls wait or waitpid it can do one of the following:

  1. block (if all children are still running)
  2. return immediately with the exit status of a child
  3. return immediately with an error (no more children)

  If the process is calling wait because of a received SIGCHLD signal, we can expect wait to return immediately, otherwise it can block.

```
pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);
```

Note: Both return process ID of terminated child, -1 on error.

Wait can block until a child terminates, while we can specify an option to waitpid to prevent blocking.

Waitpid doesn't wait for the first child to terminate; it usually waits for a specific child process.

If statloc is a non-null pointer, the termination status of the child process is stored in that location. The following macros can be used to decipher the status:

1. **WIFEXITED(status)** - true for a child that terminated normally; we can call WEXITSTATUS(status) to return the exit status.

2. **WIFSIGNALED(status)** - true for a child that terminated abnormally due to a signal; we can call WTERMSIG(status) to give the signal number.

3. **WIFSTOPPED(status)** - true is status was returned for a child that is currently stopped; we can call WSTOPSIG(status) to return the signal number.

Example:

```
void pr_exit(int status)
{
        if (WIFEXITED(status))
                printf("normal term, status = %d\n", WEXITSTATUS(status));
        else if (WIFSIGNALED(status))
                printf("abnormal term, signal = %d\n", WTERMSIG(status));
        else if (WIFSTOPPED(status))
                printf("child stopped, signal = %d\n", WSTOPSIG(status));
}
```

Example (wait_test.c):

```
int main(void)
{
        pid_t   pid;
        int     status;

        /*  Generate a child with "normal" exit */

        if ((pid = fork()) < 0)
                perror("fork failed");
        else if (pid == 0)
                exit(7);

        if (wait(&status) != pid)
                perror("wait failed");
        pr_exit(status);

        /*  Generate a child with "abnormal" exit */

        if ((pid = fork()) < 0)
                perror("fork failed");
        else if (pid == 0)
                abort();

        if (wait(&status) != pid)
                perror("wait failed");
        pr_exit(status);

        /*  Generate a child that terminates by signal */

        if ((pid = fork()) < 0)
                perror("fork failed");
        else if (pid == 0)
                status /= 0;

        if (wait(&status) != pid)
                perror("wait failed");
        pr_exit(status);

        exit(0);
}
```

- Waitpid (continued)

pid_t waitpid(pid_t pid, int *statloc, int options);

Options constants:

WNOHANG - Do not block (i.e. no child ready to wait for)
WUNTRACED - Status of stopped jobs

Use of pid arg:

pid == -1	wait for any child process (equivalent to wait)
pid > 0	wait for child whose process ID equals pid
pid == 0	wait for any child whose process group ID equals caller
pid < -1	waits for any child whose process group ID equals |pid|

Benefits of waitpid over wait:

1. waitpid allows us to wait for one particular process
2. waitpid allows a nonblocking option
3. waitpid supports job control

Example: (don't wait on child process to terminate, but prevent zombie)

Note: This is accomplished by fork'ing twice and exiting the first child.
Note: This is also known as "adoption" since init becomes parent of 2nd child.

```
int main(void)
{
        pid_t   pid;

        if ((pid = fork()) < 0)  {
                perror("fork failed");
                exit(1);
        }
        else if (pid == 0) {                            /* first child */
                if ((pid = fork()) < 0) {
                        perror("fork failed");
                        exit(1);
                }
                else if (pid > 0)                       /* parent of second = first child */
                        exit(0);

                sleep(30);      printf("second child, parent pid = %d\n", getppid());
                exit(0);
        }
```

```
        if (waitpid(pid, NULL, 0) != pid)
        {
                perror("waitpid error");
                exit(1);
        }

        exit(0);
}
```

- Wait3 and wait4

  These are provided by 4.3+ BSD UNIX, and provide one additional arg (resource sum).

  ```
  pid_t wait3(int *statloc, int options, struct rusage *rusage);
  pid_t wait4(pid_t pid, int *statloc, int options, struct rusage *rusage);
  ```

  The rusage structure contains information like user CPU, system CPI, page faults, etc.