System Programming

Prof. Chuan-Ju Wang
Dept. of Computer Science
University of Taipei

`setjmp` and `longjmp` Functions

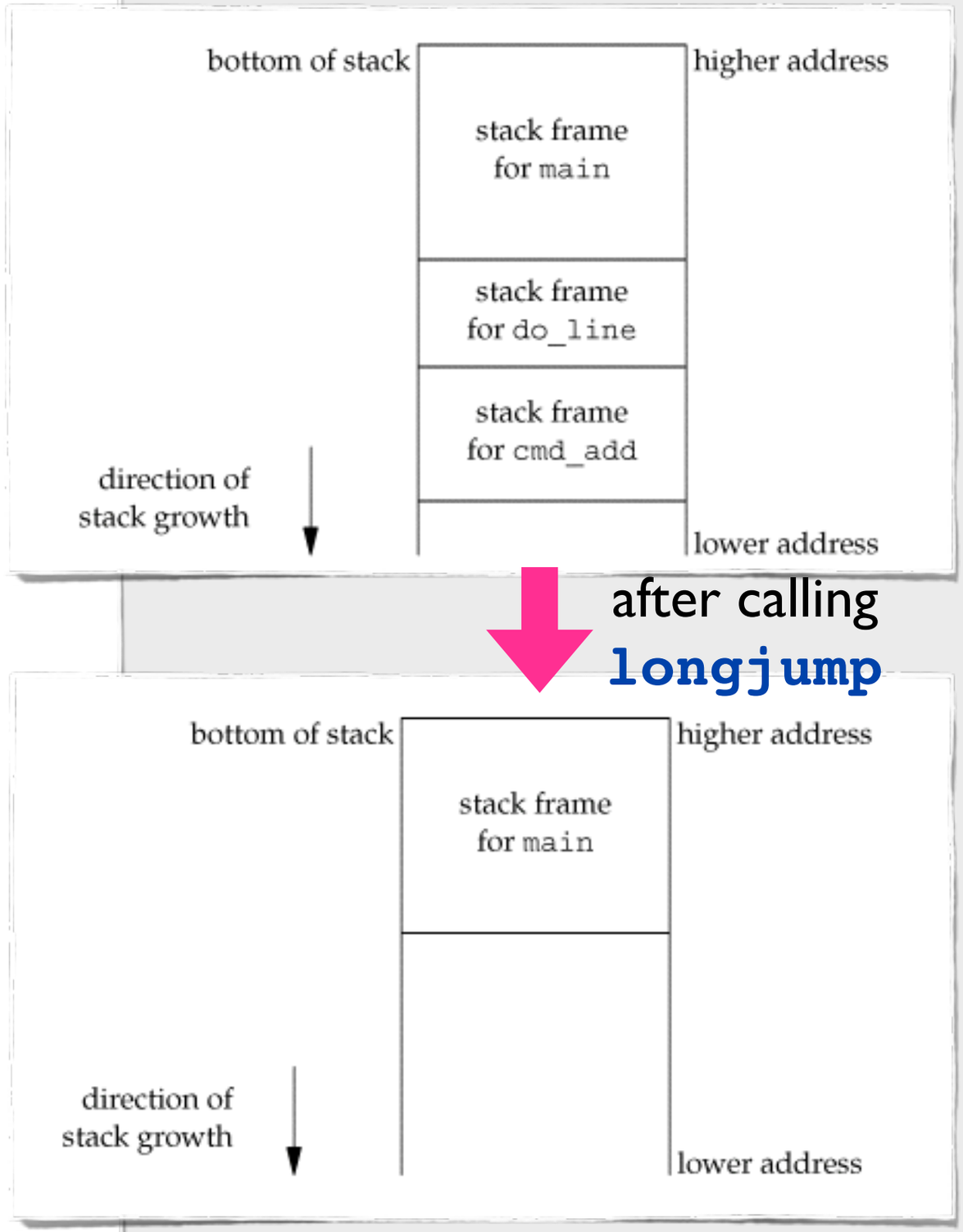
- In C, we can't goto a label that's in another function.
- Use the **`setjmp`** and **`longjmp`** functions to perform this type of branching

```
#include <setjmp.h>

int setjmp(jmp_buf env);


      Returns: 0 if called directly, nonzero if
                  returning from a call to longjmp
void longjmp(jmp_buf env, int val);
```

- Things you can (but not necessarily should) do using these functions:
  - use as "non-local goto"
  - handle error conditions in deeply nested functions
  - jump out of a signal handler back into your program
  - fake multi-threading
- Remember: `longjmp` may not be called after the routine which called `setjmp`.

Figure 7.11. Example of `setjmp` and `longjmp`

```
#include "apue.h"
#include <setjmp.h>

#define TOK_ADD    5

jmp_buf jmpbuffer;

int
main(void)
{
     char    line[MAXLINE];

     if (setjmp(jmpbuffer) != 0)
          printf("error");
     while (fgets(line, MAXLINE, stdin) != NULL)
          do_line(line);
     exit(0);
}

 ...

void
cmd_add(void)
{
     int     token;

     token = get_token();
     if (token < 0)     /* an error has occurred */
          longjmp(jmpbuffer, 1);
     /* rest of processing for this command */
}
```



Automatic, Register, and Volatile Variables

- What are the states of the automatic variables and register variables in the `main` function?
  - Unfortunately, the answer is "it depends."
  - If you have an automatic variable that you don't want rolled back, define it with the **`volatile`** attribute.
  - Variables that are declared **`global`** or **`static`** are left alone when `longjmp` is executed.

Figure 7.13. Effect of `longjmp` on various types of variables

```
#include "apue.h"
#include <setjmp.h>

static void f1(int, int, int, int);
static void f2(void);

static jmp_buf jmpbuffer;
static int     globval;

int
main(void)
{
    int              autoval;
    register int     regival;
    volatile int     volaval;
    static int       statval;

    globval = 1; autoval = 2; regival = 3; volaval = 4; statval = 5;

    if (setjmp(jmpbuffer) != 0) {
        printf("after longjmp:\n");
        printf("globval = %d, autoval = %d, regival = %d,"
            " volaval = %d, statval = %d\n",
            globval, autoval, regival, volaval, statval);
        exit(0);
    }

    /*
     * Change variables after setjmp, but before longjmp.
     */
    globval = 95; autoval = 96; regival = 97; volaval = 98;
    statval = 99;

    f1(autoval, regival, volaval, statval); /* never returns
    exit(0);
}
```

```
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] gcc fig7-13.c
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./a.out
in f1():
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
after longjmp:
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] gcc -O fig7-13.c
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./a.out
in f1():
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
after longjmp:
globval = 95, autoval = 2, regival = 3, volaval = 98, statval = 99
```

```
static void
f1(int i, int j, int k, int l)
{
    printf("in f1():\n");
    printf("globval = %d, autoval = %d, regival = %d,"
        " volaval = %d, statval = %d\n", globval, i, j, k, l);
    f2();
}
static void
f2(void)
{
    longjmp(jmpbuffer, 1);
}
```

`getrlimit` and `setrlimit` Functions

- Every process has a set of resource limits, some of which can be queried and changed by the `getrlimit` and `setrlimit` functions

```
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlptr);

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

Both return: 0 if OK, nonzero on error

```
struct rlimit {
  rlim_t  rlim_cur;   /* soft limit: current limit */
  rlim_t  rlim_max;   /* hard limit: maximum value for rlim_cur */
};
```

`getrlimit` and `setrlimit` Functions

- Changing resource limits follows these rules:
  - A soft limit can be changed by any process to a value less than or equal to its hard limit.
  - Any process can lower its hard limit greater than or equal to its soft limit.
  - Only superuser can raise hard limits.
  - Changes are per process only.

Process Control

Introduction

- We now turn to the process control provided by the UNIX System.
  - Creation of new processes
  - Program execution
  - Process termination
  - Various IDs that are the property of the process
  - Interpreter files
  - System function

Information for Processes

- **ps(1)**

```
[root@www ~]# ps aux  <==觀察系統所有的程序資料
[root@www ~]# ps -lA  <==也是能夠觀察所有系統的資料
[root@www ~]# ps axjf <==連同部分程序樹狀態
選項與參數：
-A ：所有的 process 均顯示出來，與 -e 具有同樣的效用；
-a ：不與 terminal 有關的所有 process ；
-u ：有效使用者 (effective user) 相關的 process ；
x  ：通常與 a 這個參數一起使用，可列出較完整資訊。
輸出格式規劃：
l  ：較長、較詳細的將該 PID 的的資訊列出；
j  ：工作的格式 (jobs format)
-f ：做一個更為完整的輸出。
```

Information for Processes

By default, ps selects all processes with the same effective user ID (euid=EUID) as the current user and associated with the same terminal as the invoker. It displays the process ID (pid=PID), the terminal

範例一：將目前屬於您自己這次登入的 PID 與相關資訊列示出來(只與自己的 bash 有關)

```
[root@www ~]# ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0 13639 13637  0  75   0 -  1287 wait   pts/1    00:00:00 bash
4 R     0 13700 13639  0  77   0 -  1101 -      pts/1    00:00:00 ps
```

範例二：列出目前所有的正在記憶體當中的程序：

```
[root@www ~]# ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0   2064   616 ?        Ss   Mar11   0:01 init [5]
root         2  0.0  0.0      0     0 ?        S<   Mar11   0:00 [migration/0]
root         3  0.0  0.0      0     0 ?        SN   Mar11   0:00 [ksoftirqd/0]
.....(中間省略).....
root     13639  0.0  0.2   5148  1508 pts/1    Ss   11:44   0:00 -bash
root     14232  0.0  0.1   4452   876 pts/1    R+   15:52   0:00 ps aux
root     18593  0.0  0.0   2240   476 ?        Ss   Mar14   0:00 /usr/sbin/atd
```

Information for Processes

```
範例三：以範例一的顯示內容，顯示出所有的程序：
[root@www ~]# ps -lA
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0     1     0  0  76   0 -   435 -      ?        00:00:01 init
1 S     0     2     1  0  94  19 -     0 ksofti ?        00:00:00 ksoftirqd/0
1 S     0     3     1  0  70  -5 -     0 worker ?        00:00:00 events/0
....(以下省略)....
# 你會發現每個欄位與 ps -l 的輸出情況相同，但顯示的程序則包括系統所有的程序。

範例四：列出類似程序樹的程序顯示：
[root@www ~]# ps axjf
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
    0     1     1     1 ?           -1 Ss       0   0:01 init [5]
.....(中間省略).....
    1  4586  4586  4586 ?           -1 Ss       0   0:00 /usr/sbin/sshd
 4586 13637 13637 13637 ?           -1 Ss       0   0:00  \_ sshd: root@pts/1
13637 13639 13639 13639 pts/1    14266 Ss       0   0:00      \_ -bash
13639 14266 14266 13639 pts/1    14266 R+       0   0:00          \_ ps axjf
.....(後面省略).....
```

Information for Processes

- **top(1)**

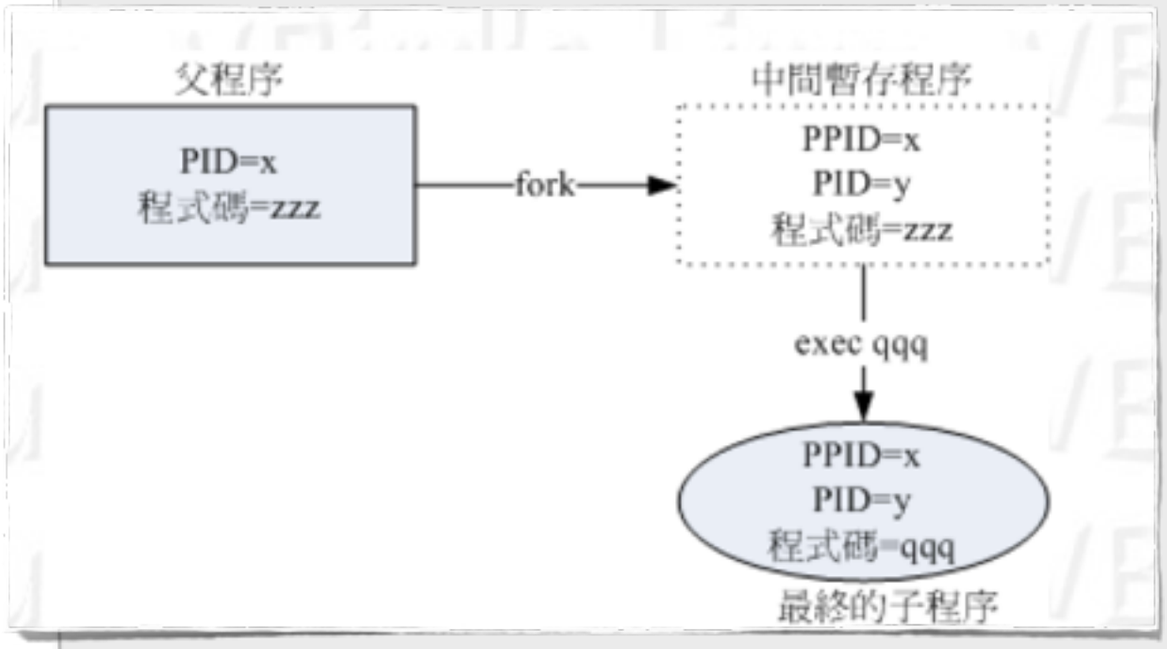
```
[root@www ~]# top [-d 數字] | top [-bnp]
選項與參數：
-d ：後面可以接秒數，就是整個程序畫面更新的秒數。預設是 5 秒；
-b ：以批次的方式執行 top，還有更多的參數可以使用喔！
    通常會搭配資料流重導向來將批次的結果輸出成為檔案。
-n ：與 -b 搭配，意義是，需要進行幾次 top 的輸出結果。
-p ：指定某些個 PID 來進行觀察監測而已。
在 top 執行過程當中可以使用的按鍵指令：
     ?：顯示在 top 當中可以輸入的按鍵指令；
     P：以 CPU 的使用資源排序顯示；
     M：以 Memory 的使用資源排序顯示；
     N：以 PID 來排序喔！
     T：由該 Process 使用的 CPU 時間累積 (TIME+) 排序。
     k：給予某個 PID 一個訊號 (signal)
     r：給予某個 PID 重新制訂一個 nice 值。
     q：離開 top 軟體的按鍵。
```

Information for Processes

```
範例一：每兩秒鐘更新一次 top，觀察整體資訊：
[root@www ~]# top -d 2
top - 17:03:09 up 7 days, 16:16,  1 user,  load average: 0.00, 0.00, 0.00
Tasks:  80 total,   1 running,  79 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.5%us,  0.5%sy,  0.0%ni, 99.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:    742664k total,   681672k used,    60992k free,   125336k buffers
Swap:  1020088k total,       28k used,  1020060k free,   311156k cached
    <==如果加入 k 或 r 時，就會有相關的字樣出現在這裡喔！
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
14398 root      15   0  2188 1012  816 R  0.5  0.1   0:00.05 top
    1 root      15   0  2064  616  528 S  0.0  0.1   0:01.38 init
    2 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 migration/0
    3 root      34  19     0    0    0 S  0.0  0.0   0:00.00 ksoftirqd/0
```

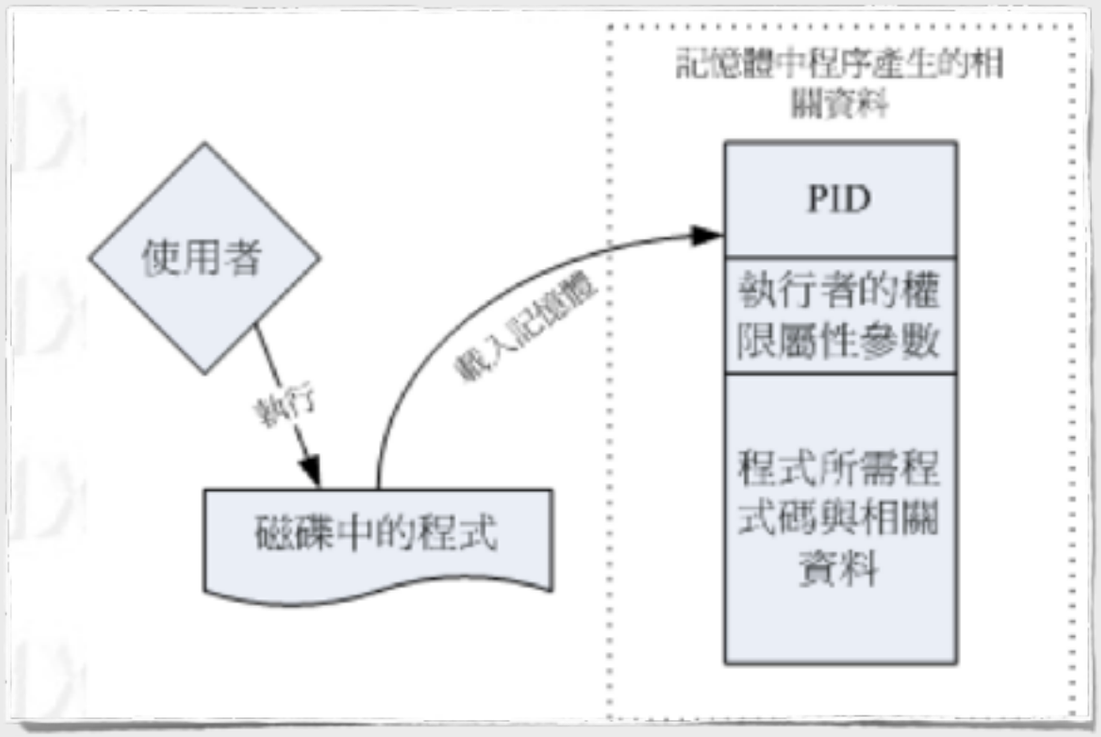
The World's Simplest Shell

```
int
main(int argc, char **argv)
{
    char buf[1024];
    pid_t pid;
    int status;

    while (getinput(buf, sizeof(buf))) {
        buf[strlen(buf) - 1] = '\0';

        if((pid=fork()) == -1) {
            fprintf(stderr, "shell: can't fork: %s\n",
                        strerror(errno));
            continue;
        } else if (pid == 0) {
            /* child */
            execlp(buf, buf, (char *)0);
            fprintf(stderr, "shell: couldn't exec %s: %s\n", buf,
                        strerror(errno));
            exit(EX_DATAERR);
        }

        if ((pid=waitpid(pid, &status, 0)) < 0)
            fprintf(stderr, "shell: waitpid error: %s\n",
                strerror(errno));
    }

    exit(EX_OK);
}
```



Process Identifiers

- Process ID's are guaranteed to be unique and identify a particular executing process with a non-negative integer.
- Certain processes have fixed, special identifiers. They are:
  - *swapper*, process ID 0 – responsible for scheduling
  - *init*, process ID 1 – bootstraps a Unix system, owns orphaned processes
  - *pagedaemon*, process ID 2 – responsible for the VM system (some Unix systems)



```
#include <unistd.h>

pid_t getpid(void);
```

```
jere@VirtualBox-MBP [~/SystemProgramming] ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000 21846 21837  0  80   0 -  1849 wait   pts/0    00:00:00 bash
0 R  1000 21981 21846  0  80   0 -  1193 -      pts/0    00:00:00 ps
```

Returns: process ID of calling process

```
pid_t getppid(void);
```

Returns: parent process ID of calling process

```
uid_t getuid(void);
```

Returns: real user ID of calling process

```
uid_t geteuid(void);
```

Returns: effective user ID of calling process

```
gid_t getgid(void);
```

Returns: real group ID of calling process

```
gid_t getegid(void);
```

Returns: effective group ID of calling process

`fork(2)` Function

- The new process created by **`fork`** is called the child process.

```
#include <unistd.h>

pid_t fork(void);



    Returns: 0 in child, process ID of child in
                                parent, -1 on error
```

- This function is called once but returns twice.
  1. The return value in the child is 0.
     - A process can have more than one child.
     - There is no function that allows a process to obtain the process IDs of its children
  2. The return value in the parent is the process ID of the new child.
     - A process can have only a single parent.
     - The child can always call **`getppid`** to obtain the process ID of its parent.

`fork(2)` Function

- Both the child and the parent continue executing with the instruction that follows the call to **`fork`**.
- The child is a copy of the parent.
  - The child gets a copy of the parent's data space, heap, and stack.
  - Note that this is a copy for the child; the parent and the child do not share these portions of memory.
  - The parent and the child share the text segment.

`fork(2)` Function

- Current implementations don't perform a complete copy of the parent's data, stack, and heap, since a fork is often followed by an **`exec`**.
  - Instead, a technique called copy-on-write (COW) is used.
  - These regions are shared by the parent and the child and have their protection changed by the kernel to read-only.
  - If either process tries to modify these regions, the kernel then makes a copy of that piece of memory only.

`fork(2)` Function

- The new process (child process) is an exact copy of the calling process (parent process) except for the following:
  - The return value from `fork`
  - The process IDs are different
  - The two processes have different parent process IDs
  - The child process has its own copy of the parent's descriptors.

Figure 8.1. Example of `fork` function

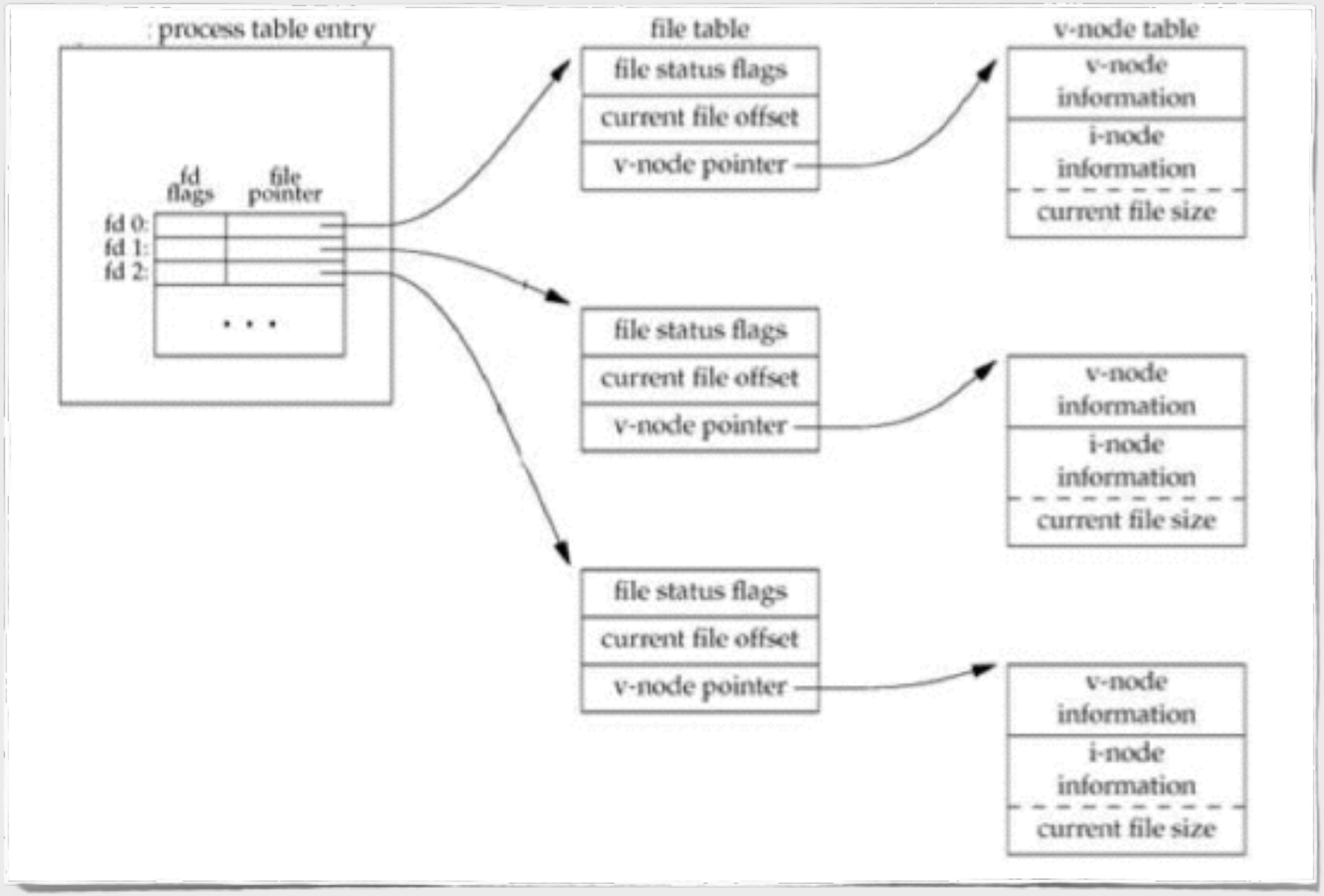
```
#include "apue.h"

int     glob = 6;       /* external variable in initialized data */
char    buf[] = "a write to stdout\n";

int
main(void)
{
    int       var;      /* automatic variable on the stack */
    pid_t     pid;

    var = 88;
    if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
        err_sys("write error");
    printf("before fork\n");    /* we don't flush stdout */

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {      /* child */
        glob++;                 /* modify variables */
        var++;
    } else {
        sleep(2);               /* parent */
    }

    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
}
```

- In general, we never know whether the child starts executing before the parent or vice versa.
- This depends on the scheduling algorithm used by the kernel.
- No order of execution between child and parent is guaranteed!

```
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./fig8-1
a write to stdout
before fork
pid = 22076, glob = 7, var = 89
pid = 22075, glob = 6, var = 88
```

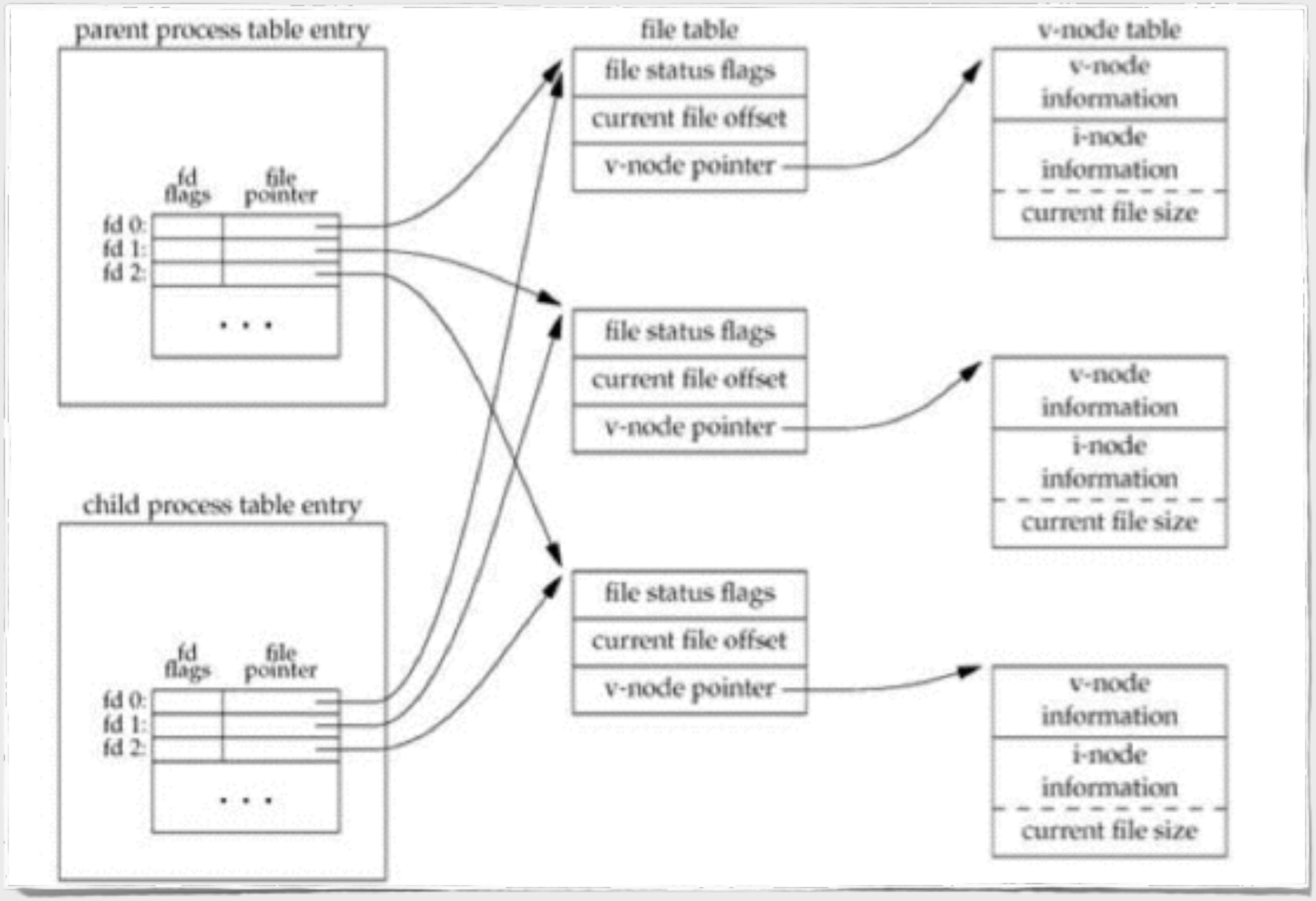
File Sharing

- All file descriptors that are open in the parent are duplicated in the child.



File Sharing

- All file descriptors that are open in the parent are duplicated in the child.



File Sharing

- There are two normal cases for handling the descriptors after a `fork`.
  - The parent waits for the child to complete.
    - When the child terminates, any of the shared descriptors that the child read from or wrote to will have their file offsets updated accordingly.
  - Both the parent and the child go their own ways.
    - After the fork, the parent closes the descriptors that it doesn't need, and the child does the same thing.

Usage of `fork(2)`

- There are two uses for fork:

  1. When a process wants to duplicate itself so that the parent and child can each execute different sections of code at the same time.

     - Network servers: The parent waits for a service request from a client. When the request arrives, the parent calls `fork` and lets the child handle the request. The parent goes back to waiting for the next service request to arrive.

  2. When a process wants to execute a different program.

     - Shells: The child does an **`exec`** (which we describe later) right after it returns from the `fork`.

`vfork(2)` Function

- The function `vfork` has the same calling sequence and same return values as `fork`.
  - Intended to create a new process when the purpose of the new process is to `exec` a new program
  - Creates the new process, just like `fork`, without copying the address space of the parent into the child
  - The child simply calls `exec` (or `exit`) right after the `vfork`.
  - While the child is running, the child runs in the address space of the parent.
  - Guarantees that the child runs first, until the child calls `exec` or `exit`.
    - When the child calls either of these functions, the parent resumes.

Figure 8.3. Example of `vfork` function

```c
#include "apue.h"

int     glob = 6;       /* external variable in initialized data */

int
main(void)
{
    int     var;        /* automatic variable on the stack */
    pid_t   pid;

    var = 88;
    printf("before vfork\n");   /* we don't flush stdio */
    if ((pid = vfork()) < 0) {
        err_sys("vfork error");
    } else if (pid == 0) {      /* child */
        glob++;                 /* modify parent's variables */
        var++;
        _exit(0);               /* child terminates */
    }
    /*
     * Parent continues here.
     */
    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
}
```

d runs in the space of the parent.

```
-lapue  -o fig8-3
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./fig8-3
before vfork
pid = 22640, glob = 7, var = 89
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test]
```

`exit(2)` Functions

- Five ways to "normally" terminate:
  1. return from `main` function
  2. `exit`
  3. `_exit` or `_Exit`
  4. return from the start routine of the last thread in the process
  5. `pthread_exit`

`exit(2)` Functions

- Three ways to "abnormally" terminate:
  1. `abort`
  2. the process receives certain signals
  3. last thread responds to a cancellation request

`exit(2)` Functions

- Regardless of how a process terminates, the same code in the kernel is eventually executed.
  - Closes all the open descriptors for the process
  - Releases the memory that it was using

- Notify the terminating's parent how it terminated.
- For the three exit functions (`exit`, `_exit`, and `_Exit`)
  - Done by passing an exit status as the argument to the function
- The exit status is converted into a termination status by the kernel when `_exit` is finally called.

Figure 8.4. Macros to examine the termination status returned by `wait` and `waitpid`

Macro	Description
`WIFEXITED`(*status*)	True if status was returned for a child that terminated normally. In this case, we can execute `WEXITSTATUS` (*status*) to fetch the low-order 8 bits of the argument that the child passed to `exit`, `_exit`,or `_Exit`.
`WIFSIGNALED` (*status*)	True if status was returned for a child that terminated abnormally, by receipt of a signal that it didn't catch. In this case, we can execute `WTERMSIG` (*status*) to fetch the signal number that caused the termination. Additionally, some implementations (but not the Single UNIX Specification) define the macro `WCOREDUMP` (*status*) that returns true if a core file of the terminated process was generated.
`WIFSTOPPED` (*status*)	True if status was returned for a child that is currently stopped. In this case, we can execute `WSTOPSIG` (*status*) to fetch the signal number that caused the child to stop.
`WIFCONTINUED` (*status*)	True if status was returned for a child that has been continued after a job control stop (XSI extension to POSIX.1; `waitpid` only).

`exit(2)` Functions

- What happens if the parent terminates before the child?
  - The **`init`** process becomes the parent process of any process whose parent terminates.
    - Whenever a process terminates, the kernel goes through all active processes to see whether the terminating process is the parent of any process that still exists.
      - If so, the parent process ID of the surviving process is changed to be 1.

`exit(2)` Functions

- What happens when a child terminates before its parent?
  - The parent wouldn't be able to fetch its termination status.
  - The kernel keeps a small amount of information for every terminating process, so that the information is available when the parent of the terminating process calls `wait` or `waitpid` (these two functions will be discussed later).
  - A process that has terminated, but whose parent has not yet waited for it, is called a zombie.

`exit(2)` Functions

- What happens when a process that has been inherited by `init` terminates?
  - Does it become a zombie? No!
  - **`init`** is written so that whenever one of its children terminates, `init` calls one of the wait functions to fetch the termination status.

`wait(2)` and `waitpid(2)` Functions

- When a process terminates, either normally or abnormally, the kernel notifies the parent by sending the `SIGCHLD` signal to the parent.
  - The parent can choose to ignore this signal, or it can provide a signal handler that is called when the signal occurs.
  - The default action for this signal is to be ignored (will be discussed in the next Chapter).

`wait(2)` and `waitpid(2)` Functions

- A parent that calls `wait(2)` or `waitpid(2)` can:
  - block (if all of its children are still running)
  - return immediately with the termination status of a child
  - return immediately with an error

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

pid_t wait(int *statloc);

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


    Both return: process ID if OK, 0 (see later), or 1 on
                                                 error
```

`wait(2)` and `waitpid(2)` Functions

- The differences between these two functions are as follows.
  - `wait(2)` can block until the process terminates, whereas `waitpid(2)` has an option to prevent it from blocking.
  - `waitpid(2)` can wait for a specific process to finish.

`wait(2)` and `waitpid(2)` Functions

- If a child has already terminated and is a zombie, `wait` returns immediately with that child's status.
- Otherwise, it blocks the caller until a child terminates.
  - If the caller blocks and has multiple children, `wait` returns when one terminates.
  - The process ID is returned by the function.

`wait(2)` and `waitpid(2)` Functions

- The argument *statloc* is a pointer to an integer.
  - Not a null pointer: The termination status of the terminated process is stored in the location pointed to by the argument.

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

pid_t wait(int *statloc);

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

Both return: process ID if OK, 0 (see later), or 1 on error

wait(2) and waitpid(2) Functions

Figure 8.5. Print a description of the `exit` status

```
#include "apue.h"
#include <sys/wait.h>

void
pr_exit(int status)
{
    if (WIFEXITED(status))
        printf("normal termination, exit status = %d\n",
                WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
        printf("abnormal termination, signal number = %d%s\n",
                WTERMSIG(status),
#ifdef  WCOREDUMP
                WCOREDUMP(status) ? " (core file generated)" : "");
#else
                "");
#endif
    else if (WIFSTOPPED(status))
        printf("child stopped, signal number = %d\n",
                WSTOPSIG(status));
}
```

- Uses the macros from Figure 8.4 to print a description of the termination status

Figure 8.6. Demonstrate various `exit` statuses

```
#include "apue.h"
#include <sys/wait.h>

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

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)                  /* child */
        exit(7);

    if (wait(&status) != pid)           /* wait for child */
        err_sys("wait error");
    pr_exit(status);                    /* and print its status */

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)                  /* child */
        abort();                        /* generates SIGABRT */

    if (wait(&status) != pid)           /* wait for child */
        err_sys("wait error");
    pr_exit(status);                    /* and print its status */

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)                  /* child */
        status /= 0;                    /* divide by 0 generates SIGFPE */

    if (wait(&status) != pid)           /* wait for child */
        err_sys("wait error");
    pr_exit(status);                    /* and print its status */

    exit(0);
}
```

```
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] make fig8-6
gcc -DLINUX -ansi -I/home/jere/SystemProgramming/apue.2e/include -Wall -D_GNU_S
-lapue  -o fig8-6
fig8-6.c: In function ‘main’:
fig8-6.c:31:10: warning: division by zero [-Wdiv-by-zero]
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./fig8-6
normal termination, exit status = 7
abnormal termination, signal number = 6
abnormal termination, signal number = 8
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test]
```

`wait(2)` and `waitpid(2)` Functions

- If we have more than one child, `wait` returns on termination of any of the children.
- What if we want to wait for a specific process to terminate?
  - `waitpid` can provides this function.

pid == 1	Waits for any child process. In this respect, `waitpid` is equivalent to `wait`.
pid > 0	Waits for the child whose process ID equals *pid*.
pid == 0	Waits for any child whose process group ID equals that of the calling process. (We discuss process groups in Section 9.4.)
pid < 1	Waits for any child whose process group ID equals the absolute value of *pid*.

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

pid_t wait(int *statloc);

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

Both return: process ID if OK, 0 (see later), or 1 on error

Figure 8.8. Avoid zombie processes by calling `fork` twice

```
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {      /* first child */
        if ((pid = fork()) < 0)
            err_sys("fork error");
        else if (pid > 0)
            exit(0);    /* parent from second fork == first child */
        /*
         * We're the second child; our parent becomes init as soon
         * as our real parent calls exit() in the statement above.
         * Here's where we'd continue executing, knowing that when
         * we're done, init will reap our status.
         */
        sleep(2);
        printf("second child, parent pid = %d\n", getppid());
        exit(0);
    }

    if (waitpid(pid, NULL, 0) != pid)  /* wait for first child */
        err_sys("waitpid error");

    /*
     * We're the parent (the original process); we continue executing,
     * knowing that we're not the parent of the second child.
     */
    exit(0);
}
```

Executing the program in Figure 8.8 gives us

```
$ ./a.out
$ second child, parent pid = 1
```

Other Functions

- `waitid(2)`, `wait3(2)` and `wait4(2)`
  - See Stevens Chap. 8.7 and 8.8 for details

Figure 8.11. Arguments supported by `wait` functions on various systems

Function	*pid*	*options*	*rusage*	POSIX.1	Free BSD 5.2.1	Linux 2.4.22	Mac OSX 10.3	Solaris 9
`wait`				•	•	•	•	•
`waitid`	•	•		XSI				•
`waitpid`	•	•		•	•	•	•	•
`wait3`		•	•		•	•	•	•
`wait4`	•	•	•		•	•	•	•

Race Conditions

- A race condition occurs when
  - multiple processes are trying to do something with shared data and the final outcome depends on the order in which the processes run.
- A process that wants to wait for a child to terminate must call one of the **`wait`** functions.
- A process wants to wait for its parent to terminate, as in the program from Figure 8.8, a loop of the following form could be used.

```
while (getppid() != 1)
    sleep(1);
```

The problem with this type of loop, called polling.

Race Conditions

- To avoid race conditions and to avoid polling, some form of signaling is required between multiple processes.
  - E.g., the parent could update a record in a log file with the child's process ID, and the child might have to create a file for the parent. We require that each process tell the other when it has finished its initial set of operations, and that each wait for the other to complete, before heading off on its own.

```c
#include  "apue.h"

TELL_WAIT();    /* set things up for TELL_xxx & WAIT_xxx */

if ((pid = fork()) < 0) {
    err_sys("fork error");
} else if (pid == 0) {              /* child */

    /* child does whatever is necessary ... */

    TELL_PARENT(getppid());     /* tell parent we're done */
    WAIT_PARENT();              /* and wait for parent */

    /* and the child continues on its way ... */

    exit(0);
}

/* parent does whatever is necessary ... */

TELL_CHILD(pid);                /* tell child we're done */
WAIT_CHILD();                   /* and wait for child */

/* and the parent continues on its way ... */

exit(0);
```

Figure 8.12. Program with a race condition

```
#include "apue.h"

static void charatat

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()
        err_sys("fork
    } else if (pid ==
        charatatime("output from child\n");
    } else {
        charatatime("output from parent\n");
    }
    exit(0);
}

static void
charatatime(char *s
{
    char    *ptr;
    int     c;

    setbuf(stdout, NULL);           /* set unbuffered */
    for (ptr = str; (c = *ptr++) != 0; )
        putc(c, stdout);
}
```

```
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./fig8-12
output from parent
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] output from child

jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./fig8-12
output from parent
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] output from child

jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./fig8-12
pouuttppuutt  ffrroom m cphairledn
t
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./fig8-12
output from parent
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] output from child
```

```
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./fig8-12
output from parent
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] output from child

jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./fig8-12
output from paorent
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] utput from child
```

Figure 8.13. Modification of Figure 8.12 to avoid race condition

```
#include "apue.h"

static void charatatime(char *);

int
main(void)
{
    pid_t   pid;

+       TELL_WAIT();
+
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {
+           WAIT_PARENT();      /* parent goes first */
        charatatime("output from child\n");
    } else {
        charatatime("output from parent\n");
+           TELL_CHILD(pid);
    }
    exit(0);
}
static void
charatatime(char *str)
{
    char    *ptr;
    int     c;

    setbuf(stdout, NULL);           /* set unbuffered */
    for (ptr = str; (c = *ptr++) != 0; )
        putc(c, stdout);
}
```

```
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./fig8-13
output from parent
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] output from child

jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./fig8-13
output from parent
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] output from child
```

exec Functions

- There are six different exec functions.
- With fork, we can create new processes; and with the exec functions, we can initiate new programs.
- The exec family of functions are used to completely replace a running process with a new executable, and the new program starts executing at its **main** function.
- The process ID does not change across an exec because a new process is not created.
  - exec merely replaces the current process its text, data, heap, and stack segments with a brand new program from disk.

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0,
➥ ... /* (char *)0 */ );

int execv(const char *pathname, char *const argv []);

int execle(const char *pathname, const char *arg0, ...
            /* (char *)0,  char *const envp[] */ );

int execve(const char *pathname, char *const
➥  argv[], char *const envp []);

int execlp(const char *filename, const char *arg0,
➥  ... /* (char *)0 */ );

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


                    All six return: 1 on error, no return on success
```

- If it has a **v** in its name, *argv* is a vector: **`char *const argv[]`**
- If it has an **l** in its name, *argv* is are a list: **`const char *arg0, ...s /* (char *) 0 */`**
- If it has an **e** in its name, it takes a **`char *const envp[]`** array of environment variables

```
PATH=/bin:/usr/bin:/usr/local/bin/:.
```

- If it has a **p** in its name, it uses the `PATH` environment variable to search for the file

exec Functions

- Every system has a limit on the total size of the argument list and the environment list.
- On some systems, for example, the command

```
grep getrlimit /usr/share/man/*/*
```

can generate a shell error of the form.

```
Argument list too long
```

- xargs(1)

```
[root@www ~]# xargs [-0epn] command
選項與參數：
-0 ：如果輸入的 stdin 含有特殊字元，例如`,\, 空白鍵等等字元時，這個 -0 參數
     可以將他還原成一般字元。這個參數可以用於特殊狀態喔！
-e ：這個是 EOF (end of file) 的意思。後面可以接一個字串，當 xargs 分析到
     這個字串時，就會停止繼續工作！
-p ：在執行每個指令的 argument 時，都會詢問使用者的意思；
-n ：後面接次數，每次 command 指令執行時，要使用幾個參數的意思。看範例三。
當 xargs 後面沒有接任何的指令時，預設是以 echo 來進行輸出喔！

範例一：將 /etc/passwd 內的第一欄取出，僅取三行，使用 finger 這個指令將每個
     帳號內容秀出來
[root@www ~]# cut -d':' -f1 /etc/passwd |head -n 3| xargs finger
Login: root                             Name: root
Directory: /root                        Shell: /bin/bash
Never logged in.
No mail.
No Plan.
.....底下省略.....
# 由 finger account 可以取得該帳號的相關說明內容，例如上面的輸出就是 finger root
# 後的結果。在這個例子當中，我們利用 cut 取出帳號名稱，用 head 取出三個帳號，
# 最後則是由 xargs 將三個帳號的名稱變成 finger 後面需要的參數！
```

- **xargs(1)**

```
範例三：將所有的 /etc/passwd 內的帳號都以 finger 查閱，但一次僅查閱五個帳號
[root@www ~]# cut -d':' -f1 /etc/passwd | xargs -p -n 5 finger
finger root bin daemon adm lp ?...y
.....(中間省略)....
finger uucp operator games gopher ftp ?...y
.....(底下省略)....
# 在這裡鳥哥使用了 -p 這個參數來讓您對於 -n 更有概念。一般來說，某些指令後面
# 可以接的 arguments 是有限制的，不能無限制的累加，此時，我們可以利用 -n
# 來幫助我們將參數分成數個部分，每個部分分別再以指令來執行！這樣就 OK 啦！^_^

範例四：同上，但是當分析到 lp 就結束這串指令？
[root@www ~]# cut -d':' -f1 /etc/passwd | xargs -p -e'lp' finger
finger root bin daemon adm ?...
# 仔細與上面的案例做比較。也同時注意，那個 -e'lp' 是連在一起的，中間沒有空白鍵。
# 上個例子當中，第五個參數是 lp 啊，那麼我們下達 -e'lp' 後，則分析到 lp
# 這個字串時，後面的其他 stdin 的內容就會被 xargs 捨棄掉了！
```

```
find /usr/share/man -type f -print | xargs grep getrlimit
```

```
find /usr/share/man -type f -print | xargs bzgrep getrlimit
```

exec Functions

- Handling of open files
  - close-on-exec flag for each descriptor (`FD_CLOEXEC`)*
    - If this flag is set, the descriptor is closed across an `exec`.
    - Otherwise, the descriptor is left open across the `exec` (default).

* See the man page of **`fcntl(2)`**

exec Functions

- The real user ID and the real group ID remain the same across the exec.
- But the effective IDs can change.
  - set-user-ID/set- group-ID bits
  - If the set-user-ID bit is set for the new program, the effective user ID becomes the owner ID of the program file.
  - Otherwise, the effective user ID is not changed.
  - The group ID is handled in the same way.

exec Functions

Figure 8.15. Relationship of the six exec functions

[View full size image]

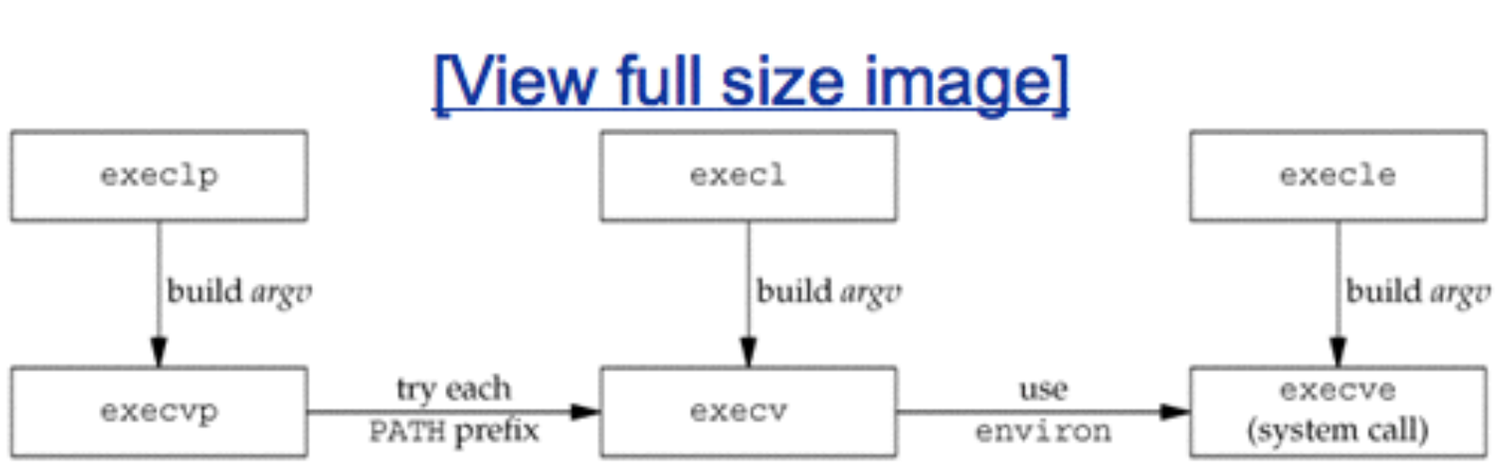


Figure 8.16. Example of exec functions

```
#include "apue.h"
#include <sys/wait.h>

char    *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {  /* specify pathname, specify environment */
        if (execle("/home/sar/bin/echoall", "echoall", "myarg1",
                "MY ARG2", (char *)0, env_init) < 0)
            err_sys("execle error");
    }

    if (waitpid(pid, NULL, 0) < 0)
        err_sys("wait error");

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {  /* specify filename, inherit environment */
        if (execlp("echoall", "echoall", "only 1 arg", (char *)0) < 0)
            err_sys("execlp error");
    }

    exit(0);
}
```

Figure 8.17. Echo all command-line arguments and all environment strings

```
#include "apue.h"

int
main(int argc, char *argv[])
{
    int         i;
    char        **ptr;
    extern char **environ;

    for (i = 0; i < argc; i++)      /* echo all command-line args */
        printf("argv[%d]: %s\n", i, argv[i]);

    for (ptr = environ; *ptr != 0; ptr++)   /* and all env strings */
        printf("%s\n", *ptr);

    exit(0);
}
```

```
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./fig8-16
argv[0]: echoall
argv[1]: myarg1
argv[2]: MY ARG2
USER=unknown
PATH=/tmp;/home/jere/SystemProgramming/apue.2e/test/bin
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] argv[0]: echoall
argv[1]: only 1 arg
SSH_AGENT_PID=1457
GPG_AGENT_INFO=/tmp/keyring-pPNXAg/gpg:0:1
TERM=xterm
SHELL=/bin/bash
```

- How to make Figs. 8.16 and 8.17 run?
  - Left to be your homework.

Changing User IDs and Group IDs

- In the UNIX System, privileges are based on user and group IDs.
  - Use the least-privilege model when we design our applications.
    - Reduce the likelihood that security can be compromised by a malicious user.

Changing User IDs and Group IDs

```
#include <unistd.h>

int setuid(uid_t uid);

int setgid(gid_t gid);
```

Both return: 0 if OK, 1 on error

- Rules for who can change the IDs:
  1. If the process has superuser privileges, the `setuid` function sets the real user ID, effective user ID, and saved set-user-ID to *`uid`*.
  2. If the process does not have superuser privileges, but *`uid`* equals either the real user ID or the saved set-user-ID, `setuid` sets only the effective user ID to *`uid`*. The real user ID and the saved set-user-ID are not changed.
  3. If neither of these two conditions is true, `errno` is set to `EPERM`, and -1 is returned.

Changing User IDs and Group IDs

- Three user IDs that the kernel maintains:
  - Only a superuser process can change the real user ID.
    - Normally, the real user ID is set by the `login(1)` program and never changes. Because `login` is a superuser process, it sets all three user IDs when it calls `setuid`.
  - The effective user ID is set by the `exec` functions only if the set-user-ID bit is set.
  - The saved set-user-ID is copied from the effective user ID by `exec`.
    - If the file's set-user-ID bit is set, this copy is saved after `exec` stores the effective user ID from the file's user ID.

Changing User IDs and Group IDs

Figure 8.18. Ways to change the three user IDs

ID	exec		setuid(*uid*)	
	set-user-ID bit off	set-user-ID bit on	superuser	unprivileged user
real user ID	unchanged	unchanged	set to *uid*	unchanged
effective user ID	unchanged	set from user ID of program file	set to *uid*	set to *uid*
saved set-user ID	copied from effective user ID	copied from effective user ID	set to *uid*	unchanged

Utility of the Saved Set-User-ID

- **`man(1)`**
  - Might have to execute several other commands to process the files containing the manual page to be displayed.
  - To prevent being tricked into running the wrong commands or overwriting the wrong files, the `man` command has to switch between two sets of privileges:
    1. those of the user running the **`man`** command
    2. those of the user that owns the **`man`** executable file

Utility of the Saved Set-User-ID

1. Assume that the **`man`** is owned by the user name **`man`** and has its set-user-ID bit set. When we **`exec`** it,

```
real user ID = our user ID
effective user ID = man
saved set-user-ID = man
```

2. The **`man`** program accesses the required configuration files and manual pages.

3. Before **`man`** runs any command on our behalf, it calls `setuid(getuid())`.

```
real user ID = our user ID (unchanged)
effective user ID = our user ID
saved set-user-ID = man (unchanged)
```

This means that we can access only the files to which we have normal access.

Utility of the Saved Set-User-ID

- When the filter is done, **`man`** calls `setuid(`*`euid`*`)`, where *`euid`* is the user ID for the user name **`man`**.
  - This call is allowed because the argument to `setuid` equals the saved set-user-ID.

```
      real user ID = our user ID (unchanged)
 effective user ID = man
saved set-user-ID = man (unchanged)
```

- The **`man`** program can now operate on its files again.

Other Functions

- Swapping of the real user ID and the effective user ID

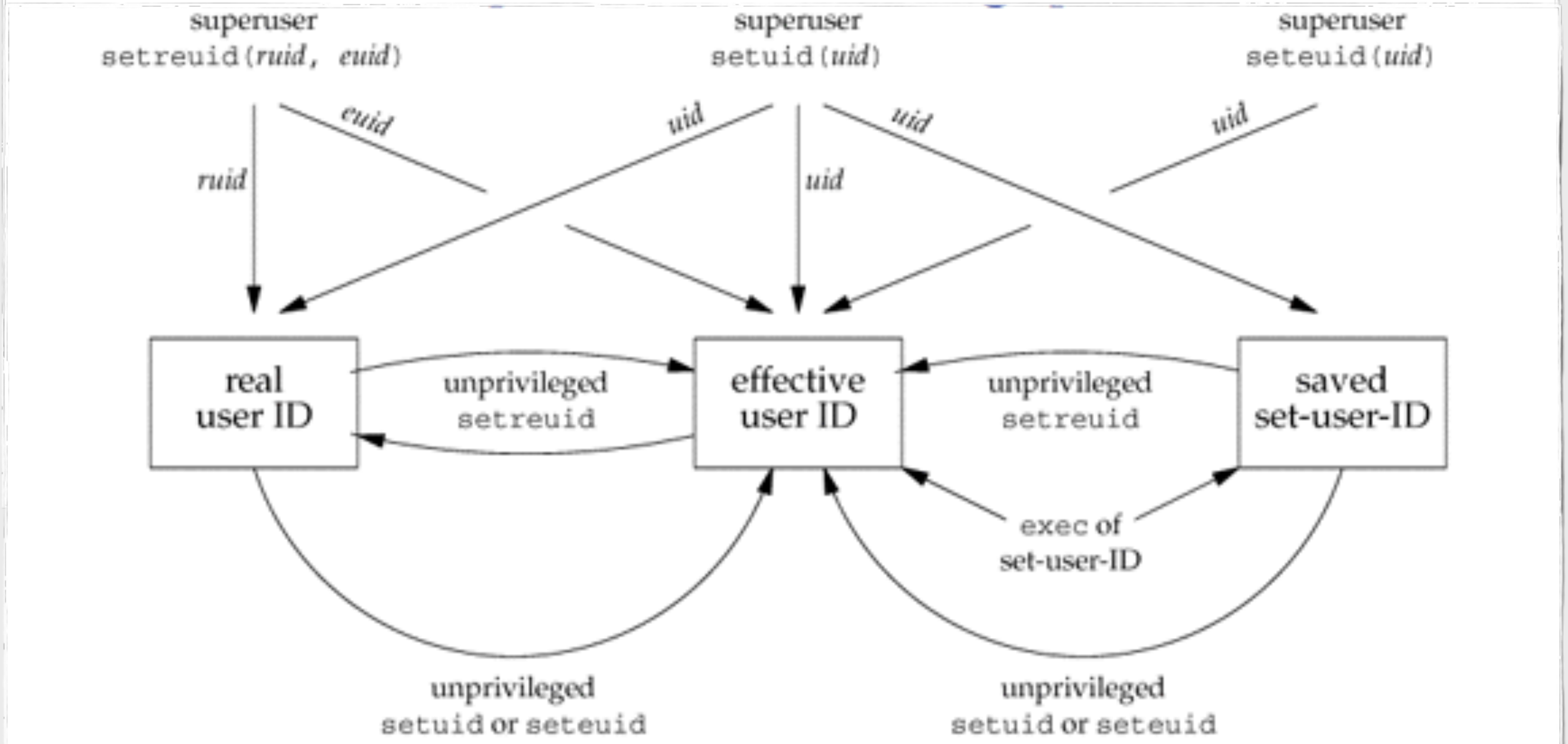
```
#include <unistd.h>

int setreuid(uid_t ruid, uid_t euid);

int setregid(gid_t rgid, gid_t egid);


                    Both return: 0 if OK, 1 on error
```

- Similar to `setuid` and `setgid`, but only the effective user ID or effective group ID is changed.

```
#include <unistd.h>

int seteuid(uid_t uid);

int setegid(gid_t gid);


                    Both return: 0 if OK, 1 on error
```

All the Functions

Figure 8.19. Summary of all the functions that set the various user IDs



Interpreter Files

- All contemporary UNIX systems support interpreter files.
  - Begin with a line of the form

```
#!  pathname [ optional-argument ]
```

- The most common one is

```
#!/bin/sh
```

The pathname is normally an absolute pathname

Interpreter Files

Figure 8.20. A program that `exec`s an interpreter file

```
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    pid_t   pid;
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {          /* child */
        if (execl("/home/sar/bin/testinterp",
                  "testinterp", "myarg1", "MY ARG2", (char *)0) < 0)
            err_sys("execl error");
    }
    if (waitpid(pid, NULL, 0) < 0) /* parent */
        err_sys("waitpid error");
    exit(0);
}
```

```
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./fig8-20
argv[0]: /home/jere/SystemProgramming/apue.2e/test/bin/echoarg
argv[1]: foo
argv[2]: /home/jere/SystemProgramming/apue.2e/test/testinterp
argv[3]: myarg1
argv[4]: MY ARG2
```

- How to make Fig. 8.20 run?
  - Left to be your homework.

`system` Function

- It is convenient to execute a command string within a program.
  - If we want to put a time-and-date stamp into a certain file, use functions `time`, `localtime`, `strftime`...
  - Much easier way:

```
system("date > file");
```

```
#include <stdlib.h>

int system(const char *cmdstring);

                                Returns: (see below)
```

`system` Function

- Because `system` is implemented by calling **`fork`**, **`exec`**, and **`waitpid`**, there are three types of return values.

  1. If either the `fork` fails or `waitpid` returns an error other than `EINTR`, system returns `1` with `errno` set to indicate the error.

  2. If the `exec` fails, implying that the shell can't be executed, the return value is as if the shell had executed `exit(127)`.

  3. Otherwise, all three functions `fork`, `exec`, and `waitpid` succeed, and the return value from system is the termination status of the shell.

Figure 8.22. The `system` function, without signal handling

```
#include    <sys/wait.h>
#include    <errno.h>
#include    <unistd.h>

int
system(const char *cmdstring)    /* version without signal handling */
{
    pid_t   pid;
    int     status;

    if (cmdstring == NULL)
        return(1);      /* always a command processor with UNIX */

    if ((pid = fork()) < 0) {
        status = -1;    /* probably out of processes */
    } else if (pid == 0) {              /* child */
        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        _exit(127);     /* execl error */
    } else {                                /* parent */
        while (waitpid(pid, &status, 0) < 0) {
            if (errno != EINTR) {
                status = -1; /* error other than EINTR from waitpid() */
                break;
            }
        }
    }

    return(status);
}
```

Figure 8.23. Calling the `system` function

```
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    int      status;

    if ((status = system("date")) < 0)
        err_sys("system() error");
    pr_exit(status);

    if ((status = system("nosuchcommand")) < 0)
        err_sys("system() error");
    pr_exit(status);

    if ((status = system("who; exit 44")) < 0)
        err_sys("system() error");
    pr_exit(status);

    exit(0);
}
```

```
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./fig8-23
Sat Mar 17 21:10:09 CST 2012
normal termination, exit status = 0
sh: nosuchcommand: not found
normal termination, exit status = 127
jere     pts/0        2012-03-12 22:08 (:0)
normal termination, exit status = 44
```

Figure 8.24. Execute the command-line argument using system

```
#include "apue.h"

int
main(int argc, char *argv[])
{
    int     status;

    if (argc < 2)
        err_quit("command-line argument required");

    if ((status = system(argv[1])) < 0)
        err_sys("system() error");
    pr_exit(status);

    exit(0);
}
```

Figure 8.25. Print real and effective user IDs

```
#include "apue.h"

int
main(void)
{
    printf("real uid = %d, effective uid = %d\n", getuid(), geteuid());
    exit(0);
}
```

```
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] make fig8-24
gcc -DLINUX -ansi -I/home/jere/SystemProgramming/apue.2e/include -Wall -D_GNU_S
 -lapue  -o fig8-24
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] make fig8-25
gcc -DLINUX -ansi -I/home/jere/SystemProgramming/apue.2e/include -Wall -D_GNU_S
 -lapue  -o fig8-25
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] mv fig8-24 tsys
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] mv fig8-25 printuids
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./tsys printuids
sh: printuids: not found
normal termination, exit status = 127
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./tsys ./printuids
real uid = 1000, effective uid = 1000
normal termination, exit status = 0
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] sudo chown root tsys
[sudo] password for jere:
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] sudo chmod u+s tsys
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ls -l tsys
-rwsrwxr-x 1 root jere 7879 2012-03-17 21:14 tsys
jere@VirtualBox-MBP [~/SystemProgramming/apue.2e/test] ./tsys ./printuids
real uid = 1000, effective uid = 0
normal termination, exit status = 0
```

Assignment

- Reading (do it at home):
  - Read and try: http://linux.vbird.org/linux_basic/0440processcontrol.php
  - Manual pages for the functions covered
  - Stevens Chap. 8

Assignment

- Assignment 7
  1. How can we kill a zombie process?
     - ➡ A report should be handed in.
  2. Write a report to explain how you make Fig. 8.16(8.17) and Fig. 8.20 run?
     - ➡ A report with detailed steps you did should be handed in.
  3. Coding:
     - Write a program that creates a zombie, and then call `system` to execute the `ps(1)` command to verify that the process is a zombie.
     - ➡ A program with brief report should be handed in.
       - There are many ways to create zombie programs. Please describe at least one additional way instead of that written in your program in the report.
       - Also, please discuss plausible ways to avoid creating zombie programs in your report.