Lecture 10. Process Control

Process control refers to new process creation, executing programs, and process termination. This section will also cover various process IDs (real, effective, saved; user and group IDs), interpreter files, the `system()` function; process accounting.

Process ID

Every process has a unique process ID, which is a non-negative integer.

The following special processes have reserved IDs:

- the **scheduler process** (a.k.a. the "swapper")—process and thread scheduling process, according to system-specific scheduling classes, resulting in scheduling priorities for each type of process. This has PID=0;
- the **init process**—starts at the end of the bootstrap procedure and ends at shutdown. It brings up the UNIX system after the kernel has been bootstrapped. This has PID=1;
- the **pagedæmon**—process supporting the virtual memory system. This has PID=2.

In addition to PID, each process has other identifiers, returned by these functions:

```
#include <sys_types.h>
#include <unistd.h>

pid_t getpid(void);
```
Returns the process ID of the calling process.

```
pid_t getppid(void);
```
Returns the parent process ID of the calling process.

```
uid_t getuid(void);
```
Returns the real user ID of the calling process.

```
uid_t geteuid(void);
```
Returns the effective user ID of the calling process.

```
gid_t getgid(void);
```
Returns the real group ID of the calling process.

```
gid_t getegid(void);
```
Returns the effective group ID of the calling process.

None of these functions has an error return.

The `fork()` Function

Other than the three kernel processes mentioned earlier, the only way a process is created under UNIX is by an existing process calling `fork()`:

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

pid_t fork(void);
```

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

The new process created with `fork()` is called the child process. `fork()`, when called once, returns twice. The return values in each of these cases are:

- 0 in the child (not a PID! just a return value);
- child's PID in the parent.

Both processes continue with the execution of the instruction following `fork()`. The child is a copy of the parent, thus getting a copy of the parent's data space, heap, and stack. These areas of the memory are disjoint. They may share, however, the text (instructions) segment, if read-only. Often, the data space is not duplicated, but rather only the written data is saved (similar to the dirty bit technique of only writing back data from cache to memory that has changed in the cache).

The order of execution of parent and child is only guaranteed by the order of priorities established by the scheduler, which is system-specific. If a form of synchronization between the parent and the child is necessary, then interprocess communication is required.

The following code illustrates the call to `fork()`, resulting in two different return values `pid_t` *`ret`*, and two different PIDs, one for the child, one for the parent:

```
#include <unistd.h> /* for fork() */
#include <stdio.h> /* for printf() */
#include <sys/types.h> /* for getpid() */

int global=6;
char buf[]="Hi, class!\n";

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

     var=88;
     if (write(STDOUT_FILENO, buf, sizeof(buf)-1)!
=sizeof(buf)-1)
          printf("Error on write().\n");
     printf("Before call to fork().\n");

     ret=fork();
     if (ret==0) /* child */
     {
          global++;
          var++;
          printf("Return value from fork() is ret=%d\n", ret);
          printf("pid=%d, global=%d, var=%d\n", getpid(),
global, var);
     }
     else /* parent */
     {
          sleep(2);
          printf("Return value from fork() is ret=%d\n", ret);
          printf("pid=%d, global=%d, var=%d\n", getpid(),
global, var);
     }
return 0;
}
```

The output of this program is:

```
$ ./myfork
Hi, class!
Before call to fork().
Return value from fork() is ret=0
pid=17042, global=7, var=89
Return value from fork() is ret=17042
pid=17041, global=6, var=88
```

Looking at the values of the global variables, we see that the parent executed first. This output also shows the return values from `fork()`, 0 for the child, non-zero for the parent, as well as their PIDs, two large consecutive numbers.

Do not confuse the return value from `fork()` *for the child* with the child's PID. On the other hand, the return value from `fork()` *for the parent* is the child's PID.

File Sharing

In what follows, to illustrate how the number and the manner of interaction between several child processes and the parent, we will use an example program which writes into a file, with small modifications. This program writes 3 names into a text file:

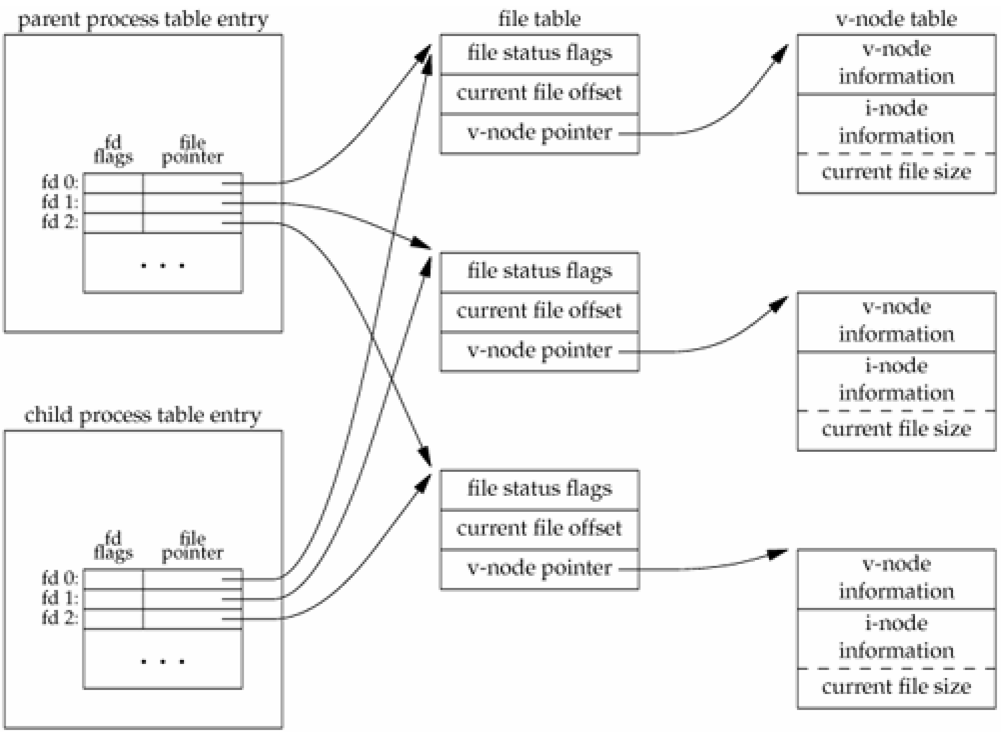
```
#include <unistd.h> /* for read(), write() */
#include <fcntl.h> /* for open() */
#include <string.h> /* for strlen() */
#include <stdio.h> /* for printf() */
#include <stdlib.h> /* for calloc() */

#define BUFFSIZE 4096
#define FILESIZE 1474560

int Input(int);

int main(int argc, char *argv[])
{
     int fd=open("/Users/awise/Stevens/Lecture10/file1.txt",
O_CREAT|O_RDWR|O_APPEND, S_IRUSR|S_IWUSR);
     int ret=Input(fd);
     if (ret==0) /* child process */
     {
          printf("Child process, ret=%d.\n", ret);
          printf("Child PID=%d.\n", getpid());
     }
     else /* parent process */
     {
          printf("Parent process, ret=%d.\n", ret);
          printf("Parent PID=%d.\n", getpid());
     }

return 0;
}

int Input(int filedes)
{
     pid_t r;
     r=fork();
     char **name=calloc(3, 20*sizeof(char));
     char **it;
     name[0]="Adriana Wise\n";
     name[1]="Richard Stevens\n";
```

```
        name[2]="Evi Nemeth\n";
        for (it=&name[0]; *it!=NULL; it++)
        {
                ssize_t num_bytes=write(filedes, *it, strlen(*it));
                sleep(2);
        }
return r;
}
```

The output in the file `file1.txt` is **interleaved**:

```
Adriana Wise
Adriana Wise
Richard Stevens
Richard Stevens
Evi Nemeth
Evi Nemeth
```

The following diagram shows how the parent and the child share the file descriptors for the files the two processes are supposed to modify:



Any change in the parent's output (e.g. standard output redirected by a shell into a log file) is reflected in the child. Both process will share that resource.

If both parent and child write to the same file descriptor (like in our example) without any form of synchronization (also like in our example), the outputs are interleaved (…). To fix this, there are 2 ways to handle file descriptors after a `fork()`:

- the parent waits for the child to complete, using `wait()`;
- they each have their separate file descriptors.

The following other resources are inherited by the child:

- real user ID, real group ID, effective user ID, effective group ID;
- supplementary group IDs;
- process group IDs;
- session ID;
- controlling terminal;
- set-user-ID flag and set-group-ID flag;
- CWD;
- root directory;
- file mode creation mask;
- signal mask and dispositions (later…);
- the close-on-exec flag for open file descriptors;
- environment;
- shared memory segments;
- resource limits.

The following are different:

- return value from `fork()`, `pid_t ret`;
- process IDs;
- parent process IDs;
- the child's `tms_utime`, `tms_stime`, `tms_cutime`, `tms_ustime` are set to 0;
- file locks set by parent are not inherited by the child (if both processes need to access a file, but parent locks, then waits for child, deadlock results);
- pending alarms are cleared for the child;
- pending signals are cleared for the child.

The 2 failure modes for `fork()` are:

1. too many processes in the system—which is indicative of some other error situation;
2. total number of processes for a given user ID is exceeded (system specific max value).

There are two uses of `fork()`:

1. For a process to duplicate itself, so that parent and child can execute different sections of the code at the same time. This is common for network servers, where a parent waits for a service request from a client, then, when the request arrives, forks and lets the child handle the request. It then initiates a new wait for the next request etc.
2. For a process to execute a different program (e.g. shells). The child does an `exec()` after `fork()`. Sometimes these two are atomic, a single operation called `spawn()`.

The `vfork()` Function

`vfork()` causes the child to run in the same address space as the parent (and not a copy, as with `fork()`). This provides some efficiency gain on certain paged virtual memory implementations of UNIX.

Another difference between `fork()` and `vfork()` is that `vfork()` guarantees that the child executes first, until it either calls `exec()` or terminates with `exit()`.

The following variant of our earlier program calls `vfork()`. Because both parent and child write into the same file, the output will no longer be interleaved.

```
#include <unistd.h> /* for read(), write() */
#include <fcntl.h> /* for open() */
#include <string.h> /* for strlen() */
#include <stdio.h> /* for printf() */
#include <stdlib.h> /* for calloc() */

#define BUFFSIZE 4096
#define FILESIZE 1474560

void Input(int);

int main(int argc, char *argv[])
{
     int fd=open("/Users/awise/Stevens/Lecture10/file2.txt",
O_CREAT|O_RDWR|O_APPEND, S_IRUSR|S_IWUSR);
     int ret=vfork();
     if (ret==0)
     {
          Input(fd);
          exit(0);
     }
     else
          Input(fd);
return 0;
}

void Input(int filedes)
{
     char **name=calloc(3, 20*sizeof(char));
     char **it;
     name[0]="Adriana Wise\n";
     name[1]="Richard Stevens\n";
     name[2]="Evi Nemeth\n";
     for (it=&name[0]; *it!=NULL; it++)
     {
          ssize_t num_bytes=write(filedes, *it, strlen(*it));
          sleep(2);
     }
}
```

The output, saved in `file2.txt`, will be **sequential**:

```
Adriana Wise
Richard Stevens
Evi Nemeth
Adriana Wise
Richard Stevens
Evi Nemeth
```

Multiple Children

The following example illustrates multiple calls to `fork()`:

```
#include <unistd.h> /* for read(), write() */
#include <fcntl.h> /* for open() */
#include <string.h> /* for strlen() */
#include <stdio.h> /* for printf() */
#include <stdlib.h> /* for calloc() */

#define BUFFSIZE 4096
#define FILESIZE 1474560

int Input(int);

int main(int argc, char *argv[])
{
     int fd=open("/Users/awise/Stevens/Lecture10/file3.txt",
O_CREAT|O_RDWR|O_APPEND, S_IRUSR|S_IWUSR);
     int ret1=Input(fd);
     int ret2=Input(fd);
     if (ret1==0) /* child process */
     {
          printf("Child process, ret1=%d.\n", ret1);
          printf("PID1=%d.\n", getpid());
     }
     else if (ret2==0)
     {
          printf("Child process, ret2=%d.\n", ret2);
          printf("PID2=%d.\n", getpid());
     }
     else /* parent process */
     {
          printf("Parent process, ret1=%d.\n", ret1);
          printf("PID=%d.\n", getpid());
          printf("Parent process, ret2=%d.\n", ret2);
          printf("PID=%d.\n", getpid());
     }
return 0;
}

int Input(int filedes)
{
     int r;
     r=fork();
     char **name=calloc(3, 20*sizeof(char));
     char **it;
     name[0]="Adriana Wise\n";
     name[1]="Richard Stevens\n";
     name[2]="Evi Nemeth\n";
     for (it=&name[0]; *it!=NULL; it++)
     {
          ssize_t num_bytes=write(filedes, *it, strlen(*it));
          sleep(2);
     }
return r;
}
```

The command line output shows the return values from `fork()` and the PIDs of children and parent:

```
$ ./myfork3
Parent process, ret1=17234.
```

```
PID=17233.
Parent process, ret2=17236.
Child process, ret1=0.
Child process, ret1=0.
PID=17233.
PID1=17234.
PID1=17235.
Child process, ret2=0.
PID2=17236.
```

The output in file3.txt shows the number of times the “work” defined in `Input()` was done:

```
Adriana Wise
Adriana Wise
Richard Stevens
Richard Stevens
Evi Nemeth
Evi Nemeth
Adriana Wise
Adriana Wise
Adriana Wise
Adriana Wise
Richard Stevens
Richard Stevens
Richard Stevens
Richard Stevens
Evi Nemeth
Evi Nemeth
Evi Nemeth
Evi Nemeth
```

The first 2 interleaved entries were created by the first `fork()`, which resulted in 2 instances of the process. The following 4 interleaved entries were created by the second call to `fork()`, which created 2 children for each of the previous 2 instances of the process (created by the first `fork()`). That makes 2*2=4.

The total is 2+2*2=6.

Exit Status vs. Termination Status

With the exit functions, the user process generated an exit number (status) to pass to the exit function, and handled each exit condition accordingly (with an exit handler). In an abnormal situation (beyond the predictability of the programmer), the kernel generates a termination status to indicate the reason for the abnormal termination.

In either case, the parent process can obtain the termination status of the child by calling the `wait()` or the `waitpid()` functions. The following distinction is necessary: while we call **exit status** the argument to the `exit()` function (or the return value from `main()`), we call **termination status** the return value from `wait()` and `waitpid()`.

Zombie Processes

If a parent terminates before the child, the termination status gets returned to `init`. We say that init **inherited** the (child) process. The PID of the parent is changed to 1.

If a child terminates before the parent, the kernel maintains process information about the child in order to be able to pass it on to the parent when the parent calls `wait()` or `waitpid()`. This information contains:

- PID of the child;
- termination status of the child;
- amount of CPU time taken by the child.

The process that has terminated, but whose parent has not, and has not called `wait()` on it, is called a **zombie**. Basically, processes whose termination status has not been fetched to the parent process are zombies.

The processes inherited by `init` do not become zombies, because `init` has the call to `wait()` built-in.

The `wait()` and `waitpid()` Functions

We have talked, but not described yet, these functions.

When a child process terminates, the kernel notifies the parent by sending it a SIGCHLD signal. This signal is asynchronous (i.e. can happen at any time while the parent is still running). Therefore, the parent must be equipped to deal with this signal by providing a signal handler.

For now, when a child terminates, the parent process may take the following actions:

- block (i.e. "wait" its own execution until all its children terminate);
- return immediately with the termination status of a child;
- return immediately with an error (if the call to `wait()` was performed, and no children remain to apply it to).

```
#include <sys/types.h>
#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, or -1 if error.

The differences between these two functions are:

- `wait()` can block the parent until the child terminates, while `waitpid()` has an option that prevents the blocking;
- `waitpid()` doesn't wait for the children to terminate in sequence, rather it has a number of options that control which process it waits for.

The arguments are:

- *`statloc`*—a pointer to an integer which gets filled in by the call to `wait()` or `waitpid()`. The value is used with macros defined in `<sys/wait.h>` (see table above) to retrieve a termination status number. This status number can be cross-referenced with a list of entries in `<signal.h>` to provide a more verbose description of the termination condition;
- *`pid`*—the PID of the child waited on, or…
  - *`pid`*`==1`—waits for any child process
  - *`pid`*`>0`—waits for the child whose ID is specified
  - *`pid`*`==0`—waits for any child whose process GID is the same as the one of the parent (weird, we were told that the child inherits the parent's process GID)
  - *`pid`*`<-1`—waits for any child whose process GID is the same as the absolute value of the PID passed
- *`options`*—either 0 or constructed as a bitwise OR operation with constants from the following table:

Constant	Description
`WCONTINUED`	If the implementation supports job control, the status of any child specified by pid that has been continued after being stopped, but whose status has not yet been reported, is returned (XSI extension to POSIX.1).
`WNOHANG`	The `waitpid` function will not block if a child specified by pid is not immediately available. In this case, the return value is 0.
`WUNTRACED`	If the implementation supports job control, the status of any child specified by pid that has stopped, and whose status has not been reported since it has stopped, is returned. The `WIFSTOPPED` macro determines whether the return value corresponds to a stopped child process.

The termination status both fill in, is passed onto the following macros:

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).

Return values:

- if `wait()` returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. Otherwise, a value of -1 is returned and errno is set to indicate the error.
- if `wait3()`, `wait4()`, or `waitpid()` returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process.
- if there are no children not previously awaited, -1 is returned with errno set to [ECHILD]. Otherwise, if WNOHANG is specified and there are no stopped or exited children, 0 is returned. If an error is detected or a caught signal aborts the call, a value of -1 is returned and errno is set to indicate the error.

The following two programs show these functions:

```
#include <unistd.h> /* for read(), write() */
#include <fcntl.h> /* for open() */
#include <string.h> /* for strlen() */
#include <stdio.h> /* for printf() */
#include <stdlib.h> /* for calloc() */
#include <stdbool.h> /* for true */
#include <sys/wait.h> /* for the macros */

#define BUFFSIZE 4096
#define FILESIZE 1474560

void Input(int);
void PrintStatus(int status);

int main(int argc, char *argv[])
{
     int *status_child=malloc(sizeof(int));
     int fd=open("/Users/awise/Stevens/Lecture10/file4.txt",
O_CREAT|O_RDWR|O_APPEND, S_IRUSR|S_IWUSR);
     int ret=fork();
     if (ret==0) /* child process */
     {
          Input(fd);
          sleep(2);
     }
     else /* parent process */
     {
          int pid_child=wait(status_child);
          printf("pid_child=%d\n", pid_child);
          printf("status_child=%d\n", *status_child);
          PrintStatus(*status_child);
          free(status_child);
          Input(fd);
     }
return 0;
}

void Input(int filedes)
{
     char **name=calloc(3, 20*sizeof(char));
     char **it;
     name[0]="Adriana Wise\n";
     name[1]="Richard Stevens\n";
     name[2]="Evi Nemeth\n";
     for (it=&name[0]; *it!=NULL; it++)
     {
          ssize_t num_bytes=write(filedes, *it, strlen(*it));
          sleep(2);
     }
}

void PrintStatus(int status)
{
     int lower_8_bits;
     if ((lower_8_bits=WIFEXITED(status))==true)
     {
          printf("Exit status for child=%d\n", status);
          printf("lower_8_bits=%x\n", status);
     }
     else if (WIFSIGNALED(status)==true)
          printf("Exit status for child=%d\n", status);
     else if (WIFSTOPPED(status)==true)
          printf("Exit status for child=%d\n", status);
}
```

With the call to `wait()`, even though we're calling `fork()`, and not `vfork()`, the output to our shared text file will be still sequential, because the parent will not write until after waiting for the child to complete:

```
Adriana Wise
Richard Stevens
Evi Nemeth
Adriana Wise
Richard Stevens
Evi Nemeth
```

The next program uses `waitpid()`, with the WNOHANG option, meaning the parent won't block waiting for the child to complete:

```
#include <unistd.h> /* for read(), write() */
#include <fcntl.h> /* for open() */
#include <string.h> /* for strlen() */
#include <stdio.h> /* for printf() */
#include <stdlib.h> /* for calloc() */

#define BUFFSIZE 4096
#define FILESIZE 1474560

void Input(int, char *);

int main(int argc, char *argv[])
{
     int *status_child;
     int fd=open("/Users/awise/Stevens/Lecture10/file6.txt",
O_CREAT|O_RDWR|O_APPEND, S_IRUSR|S_IWUSR);
     int ret=fork();
     int pid_child, pid_parent;
     char *whichProcess;
     if (ret==0) /* child process */
     {
          whichProcess="Child process:
\n---------------------------\n";
          pid_child=getpid();
          printf("In child process: pid_child=%d\n", pid_child);
          Input(fd, whichProcess);
          sleep(2);
     }
     else /* parent process */
     {
          whichProcess="Parent process:
\n---------------------------\n";
          pid_parent=getpid();
          pid_child=waitpid(pid_child, status_child, WNOHANG);
          printf("In parent process: pid_child=%d\n",
pid_child);
          Input(fd, whichProcess);
     }
return 0;
}

void Input(int filedes, char *process)
```

```
{
     ssize_t first_line=write(filedes, process,
strlen(process));

     char **name=calloc(3, 20*sizeof(char));
     char **it;
     name[0]="Adriana Wise\n";
     name[1]="Richard Stevens\n";
     name[2]="Evi Nemeth\n";
     for (it=&name[0]; *it!=NULL; it++)
     {
          ssize_t num_bytes=write(filedes, *it, strlen(*it));
          sleep(2);
     }
}
```

The command line output is:

```
$ ./mywaitpid
In parent process: pid_parent=17439
Returned status_child=0
In child process: pid_child=17440
```

The file output is:

```
Child process:
--------------------------
Adriana Wise
Richard Stevens
Evi Nemeth
Parent process:
--------------------------
Child process:
--------------------------
Adriana Wise
Adriana Wise
Richard Stevens
Richard Stevens
Evi Nemeth
Evi Nemeth
```

The purpose was to show the interleaving due to the WNOHANG option, but since the line is written inside the "work" function, it'll only appear once each process is started, and not each time they reenter the scene.

The following example introduces abnormal exit statuses for the children processes:

```
#include <unistd.h> /* for read(), write() */
#include <fcntl.h> /* for open() */
#include <string.h> /* for strlen() */
#include <stdio.h> /* for printf() */
#include <stdlib.h> /* for calloc() */
#include <stdbool.h> /* for true */

#define BUFFSIZE 4096
#define FILESIZE 1474560

void Input(int);
void PrintStatus(int status);

int main(int argc, char *argv[])
{
     int fd=open("/Users/awise/Stevens/Lecture10/file5.txt",
O_CREAT|O_RDWR|O_APPEND, S_IRUSR|S_IWUSR);
     int ret, status;
     for (int n=0; n<3; n++)
     {
          if ((ret=fork())==0) /* child process */
          {
               Input(fd);
               printf("Child process, ret=%d.\n", ret);
               status=wait(&status);
               switch(n)
               {
               case 0:
                    exit(7);
                    break;
               case 1:
                    abort();
                    break;
               case 2:
                    status=status/0;
                    break;
               }
          }
          else
          {
               printf("Parent process, child ret==%d.\n", ret);
               wait(&status);
               PrintStatus(status);
          }
     }
return 0;
}

void Input(int filedes)
{
     char **name=calloc(3, 20*sizeof(char));
     char **it;
     name[0]="Adriana Wise\n";
     name[1]="Richard Stevens\n";
     name[2]="Evi Nemeth\n";
     for (it=&name[0]; *it!=NULL; it++)
     {
          ssize_t num_bytes=write(filedes, *it, strlen(*it));
          sleep(2);
     }
}

void PrintStatus(int status)
{
        int lower_8_bits;
        if ((lower_8_bits=WIFEXITED(status))==true)
        {
                printf("WIFEXITED(status)=%d\n",
WIFEXITED(status));
                printf("Exit status for child=%d\n",
WEXITSTATUS(status));
                printf("lower_8_bits=%x\n", status);
        }
        else if (WIFSIGNALED(status)==true)
     {
                printf("WIFSIGNALED(status)=%d\n",
WIFSIGNALED(status));
                printf("Exit status for child=%d\n",
WTERMSIG(status));
                printf("Exit status for child=%d\n",
WCOREDUMP(status));
     }
        else if (WIFSTOPPED(status)==true)
     {
                printf("WIFSTOPPED(status)=%d\n",
WIFSTOPPED(status));
                printf("Exit status for child=%d\n",
WSTOPSIG(status));
     }
}
```

The command line output is:

```
$ ./myexitstatus
Parent process, child ret==17455.
Child process, ret=0.
WIFEXITED(status)=1
Exit status for child=7
lower_8_bits=700
Parent process, child ret==17456.
Child process, ret=0.
WIFSIGNALED(status)=1
Exit status for child=6
Exit status for child=0
Parent process, child ret==17458.
Child process, ret=0.
WIFSIGNALED(status)=1
Exit status for child=8
Exit status for child=0
```

The double line comes from using both options provided for parsing the `WIFSIGNALED(status)` termination status.

Homework (due Mar-29-2016):

Add functionality to this program to print the error messages associated with the numeric termination status values from `<signal.h>`. Also add functionality to print the values of the members of `struct rusage`, which will get filled in by calling `wait3()`. Check these entries with `man 2`.