CSE 344 System Programming

Week 5

Processes continued…

- *Waiting for children processes*
- *Orphans, zombies ,*
- *Daemons*

Monitoring Child Processes

In many application designs, a parent process need to know when one of its child processes changes state—when the child terminates or is stopped by a signal.

This lecture we will focus on two techniques used to monitor child processes: the *wait()* system call (and its variants) and the use of the SIGCHLD signal.

The *wait()* system Call

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

pid_t wait(int  *status);

                                    Returns process ID of terminated child, or -1 on error
```

- The *wait()* system call waits for one of the children of the calling process to terminate and returns the termination status of that child in the buffer pointed to by status
- If no (previously unwaited-for) child of the calling process has yet terminated, the call blocks until one of the children terminates. If a child has already terminated by the time of the call, *wait()* returns immediately.
- If status is not NULL , information about how the child terminated is returned in the integer to which status points.

```
#include <sys/wait.h>
#include <time.h>
#include "curr_time.h"              /* Declaration of currTime() */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int numDead;       /* Number of children so far waited for */
    pid_t childPid;    /* PID of waited for child */
    int j;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s sleep-time...\n", argv[0]);

    setbuf(stdout, NULL);           /* Disable buffering of stdout */

    for (j = 1; j < argc; j++) {    /* Create one child for each argument */
        switch (fork()) {
        case -1:
            errExit("fork");

        case 0:                     /* Child sleeps for a while then exits */
            printf("[%s] child %d started with PID %ld, sleeping %s "
                    "seconds\n", currTime("%T"), j, (long) getpid(), argv[j]);
            sleep(getInt(argv[j], GN_NONNEG, "sleep-time"));
            _exit(EXIT_SUCCESS);

        default:                    /* Parent just continues around loop */
            break;
        }
    }

    numDead = 0;
    for (;;) {                      /* Parent waits for each child to exit */
        childPid = wait(NULL);
        if (childPid == -1) {
            if (errno == ECHILD) {
                printf("No more children - bye!\n");
                exit(EXIT_SUCCESS);
            } else {                /* Some other (unexpected) error */
                errExit("wait");
            }
        }

        numDead++;
        printf("[%s] wait() returned child PID %ld (numDead=%d)\n",
                currTime("%T"), (long) childPid, numDead);
    }
}
```

```
$ ./multi_wait 7 1 4
[13:41:00] child 1 started with PID 21835, sleeping 7 seconds
[13:41:00] child 2 started with PID 21836, sleeping 1 seconds
[13:41:00] child 3 started with PID 21837, sleeping 4 seconds
[13:41:01] wait() returned child PID 21836 (numDead=1)
[13:41:04] wait() returned child PID 21837 (numDead=2)
[13:41:07] wait() returned child PID 21835 (numDead=3)
No more children - bye!
```

The *waitpid()* System Call

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

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

                                                Returns process ID of child, 0, or -1 on error
```

- If a parent process has created multiple children, it is not possible to *wait()* for the completion of a specific child; we can only wait for the next child that terminates.
- If no child has yet terminated, *wait()* always blocks. Sometimes, it would be preferable to perform a nonblocking wait so that if no child has yet terminated, we obtain an immediate indication of this fact.
- Using *wait()*, we can find out only about children that have terminated. It is not possible to be notified when a child is stopped by a signal (such as SIGSTOP or SIGTTIN) or when a stopped child is resumed by delivery of a SIGCONT signal.

waitpid ()

The return value and status arguments of *waitpid()* are the same as for *wait()*.

The pid argument enables the selection of the child to be waited for, as follows :

- If pid is greater than 0, wait for the child whose process ID equals pid
- If pid equals 0, wait for any child in the same process group as the caller
- If pid is less than –1, wait for any child whose process group identifier equals the absolute value of pid.
- If pid equals –1, wait for any child. The call `wait(&status)` is equivalent to the call `waitpid(-1, &status, 0)`.

```
#include <sys/wait.h>
#include "print_wait_status.h"   /* Declares printWaitStatus()  at listing 26.2 pp546 */
#include "tlpi_hdr.h"

int main(int argc, char *argv[])
{
 int status;
 pid_t childPid;
   if (argc > 1 && strcmp(argv[1], "--help") == 0)
      usageErr("%s [exit-status]\n", argv[0]);

  switch (fork()) {
   case -1: errExit("fork");

   case 0: /* Child: either exits immediately with given status or loops waiting for
              signals */
           printf("Child started with PID = %ld\n", (long) getpid());
           if (argc > 1)                          /* Status supplied on command line? */
              exit(getInt(argv[1], 0, "exit-status"));
           else                                   /* Otherwise, wait for signals */
               for (;;)
                 pause();
               exit(EXIT_FAILURE);               /* Not reached, but good practice */
   default:                                 /* Parent: repeatedly wait on child until it
                                               either exits or is terminated by a signal */
     for (;;) {
               childPid = waitpid(-1, &status, WUNTRACED| WCONTINUED );
               if (childPid == -1)
                 errExit("waitpid");
                 printf("waitpid() returned: PID=%ld; status=0x%04x (%d,%d)\n",
                   (long) childPid, (unsigned int) status, status >> 8, status & 0xff);
                printWaitStatus(NULL, status);
              if (WIFEXITED(status) || WIFSIGNALED(status))
          exit(EXIT_SUCCESS); }
       }
}
```

What is going on ?

- The *printWaitStatus()* function is used in Listing 26-3. This program creates a child process that either loops continuously calling *pause()* or, if an integer command-line argument was supplied, exits immediately using this integer as the exit status.
- In the meantime, the parent monitors the child via *waitpid()*, printing the returned status value and passing this value to *printWaitStatus()*.
- The parent exits when it detects that the child has either exited normally or been terminated by a signal.

Process Termination from a Signal Handler

- In some circumstances, we may wish to have certain cleanup steps performed before a process terminates. For this purpose, we can arrange to have a handler catch such signals, perform the cleanup steps, and then terminate the process.
- If we do this, we should bear in mind that the termination status of a process is available to its parent via *wait()* or *waitpid()*.
- For example, calling *_exit(EXIT_SUCCESS)* from the signal handler will make it appear to the parent process that the child terminated successfully

Process Termination from a Signal Handler

- If the child needs to inform the parent that it terminated because of a signal, then the child's signal handler should first disestablish itself, and then raise the same signal once more. The signal handler would contain code such as the following:

```
void handler(int sig)
{
    /* Perform cleanup steps */

    signal(sig, SIG_DFL);          /* Disestablish handler */
    raise(sig);                    /* Raise signal again */

}
```

The `waitid()` System Call

Like *waitpid()*, *waitid()* returns the status of child processes. However, *waitid()* provides extra functionality that is unavailable with *waitpid()*

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

int waitid(idtype_t idtype , id_t id , siginfo_t * infop , int options );
```

Returns 0 on success or if WNOHANG was specified and there were no children to wait for, or –1 on error

Orphans and Zombies

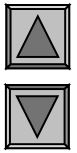
The lifetimes of parent and child processes are usually not the same—either the parent outlives the child or vice verse This raises two questions:

- Who becomes the parent of an orphaned child? The orphaned child is adopted by *init*, the ancestor of all processes, whose process ID is 1. In other words, after a child's parent terminates, a call to *getppid()* will return the value 1 (This can be used as a way of determining if a child's true parent is still alive)
- What happens to a child that terminates before its parent has had a chance to perform a *wait()*? The point here is that, although the child has finished its work, the parent should still be permitted to perform a *wait()* at some later time to determine how the child terminated. The kernel deals with this situation by turning the child into a zombie.

Zombies

- Regarding zombies, UNIX systems imitate the movies—a zombie process can't be killed by a signal, not even the (silver bullet) SIGKILL. This ensures that the parent can always eventually perform a *wait()*
- When the parent does perform a *wait()*, the kernel removes the zombie, since the last remaining information about the child is no longer required. On the other hand, if the parent terminates without doing a *wait()*, then the *init* process adopts the child and automatically performs a *wait()*, thus removing the zombie process from the system.
- If a parent creates a child, but fails to perform a *wait()*, then an entry for the zombie child will be maintained indefinitely in the kernel's process table. If a large number of such zombie children are created, they will eventually fill the kernel process table, preventing the creation of new processes.

```
#include <signal.h>
#include <libgen.h>
#include "tlpi_hdr.h"                   /* For basename() declaration */
#define CMD_SIZE 200

int main(int argc, char *argv[])
{
   char cmd[CMD_SIZE];
   pid_t childPid;
   setbuf(stdout, NULL);                /* Disable buffering of stdout */
   printf("Parent PID=%ld\n", (long) getpid());

    switch (childPid = fork()) {
     case -1:
          errExit("fork");
     case 0:                            /* Child: immediately exits to become zombie */
          printf("Child (PID=%ld) exiting\n", (long) getpid());
         _exit(EXIT_SUCCESS);
     default:                           /* Parent */
          sleep(3);                     /* Give child a chance to start and exit */
          snprintf(cmd, CMD_SIZE, "ps | grep %s", basename(argv[0]));
          cmd[CMD_SIZE - 1] = '\0';     /* Ensure string is null-terminated */
          system(cmd);                  /* View zombie child, Books example
                                         * use system() with caution
                                            */
          /* Now send the "sure kill" signal to the zombie */
          if (kill(childPid, SIGKILL) == -1)
               errMsg("kill");
          sleep(3);                     /* Give child a chance to react to signal */
          printf("After sending SIGKILL to zombie (PID=%ld):\n", (long) childPid);
          system(cmd);                  /* View zombie child again *
          exit(EXIT_SUCCESS);
       }
}
```



make_zombie.c

The SIGCHLD Signal

The termination of a child process is an event that occurs asynchronously. A parent can't predict when one of its child will terminate. We have already seen that the parent should use *wait()* (or similar) in order to prevent the accumulation of zombie children, and have looked at two ways in which this can be done:

- The parent can call *wait()*, or *waitpid()* without specifying the WNOHANG flag, in which case the call will block if a child has not already terminated.
- The parent can periodically perform a nonblocking check (a poll) for dead children via a call to *waitpid()* specifying the WNOHANG flag

Both of these approaches are inconvenient .

On the one hand, we may not want the parent to be blocked waiting for a child to terminate. Making repeated nonblocking *waitpid()* calls wastes CPU time and adds complexity to an application design.

Establishing a Handler for SIGCHLD

- The SIGCHLD signal is sent to a parent process whenever one of its children terminates. By default, this signal is ignored
- A common way of reaping dead child processes is to establish a handler for the SIGCHLD signal. This signal is delivered to a parent process whenever one of its children terminates, and optionally when a child is stopped by a signal.
- Alternatively, but somewhat less portable, a process may elect to set the disposition of SIGCHLD to SIG_IGN , in which case the status of terminated children is immediately discarded (and thus can’t later be retrieved by the parent), and the children don’t become zombies.

Executing a New Program: *execve()*

- The *execve()* system call loads a new program into a process's memory. During this operation, the old program is discarded, and the process's stack, data, and heap are replaced by those of the new program.
- The most frequent use of *execve()* is in the child produced by a *fork()*, although it is also occasionally used in applications without a preceding *fork()*.
- Various library functions, all with names beginning with *exec*, are layered on top of the *execve()* system call. Each of these functions provides a different interface to the same functionality. The loading of a new program by any of these calls is commonly referred to as an exec operation, or simply by the notation *exec()*.

evecve()

```
#include <uninst.h>

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

                                                Never returns on success; returns –1 on error
```

The *pathname* argument contains the pathname of the new program to be loaded into the process's memory.

The *argv* argument specifies the command-line arguments to be passed to the new program. The value supplied for *argv[0]* corresponds to the command name.

The final argument, *envp*, specifies the environment list for the new program. The *envp* argument corresponds to the environ array of the new program; it is a NULL - terminated list of pointers to character strings of the form *name=value*

evecve()

Since it replaces the program that called it, a successful *execve()* never returns. We never need to check the return value from *execve()*; it will always be –1. The very fact that it returned informs us that an error occurred. Among the errors that may be returned in errno are

the following:

EACCES : The pathname argument doesn't refer to a regular file, the file doesn't have execute permission enabled, or one of the directory components of pathname is not searchable

ENOENT : The file referred to by pathname doesn't exist.

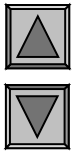
ENOEXEC : The file referred to by pathname is marked as being executable, but it is not in a recognizable executable format (possibly, it is a script that doesn't begin with a line specifying a script interpreter).

ETXTBSY : The file referred to by pathname is open for writing by another process

E2BIG : The total space required by the argument list and environment list exceeds the allowed maximum

```
#include "tlpi_hdr.h"

int main(int argc, char *argv[])
{
   char *argVec[10];                                    /* Larger than required */
   char *envVec[] = { "GREET=salut", "BYE=adieu", NULL };

    if (argc != 2 || strcmp(argv[1], "--help") == 0)
         usageErr("%s pathname\n", argv[0]);

     argVec[0] = strrchr(argv[1], '/');               /* Get basename from argv[1] */
     if (argVec[0] != NULL)
         argVec[0]++;
    else
        argVec[0] = argv[1];
        argVec[1] = "hello world";
        argVec[2] = "goodbye";
        argVec[3] = NULL;                   /* List must be NULL-terminated */

     execve(argv[1], argVec, envVec);
     errExit("execve");                    /* If we get here, something went wrong */

}
```

```c
#include "tlpi_hdr.h"
extern char **environ;

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

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

    for (ep = environ; *ep != NULL; ep++)
      printf("environ: %s\n", *ep);

exit(EXIT_SUCCESS);

}
```



$ **./t_execve ./envargs**

argv[0] = envargs
argv[1] = hello world
argv[2] = goodbye
environ: GREET=salut
environ: BYE=adieu

Example Program : envargs.c

The exec() library functions

```
#include <uninst.h>

int execle(const char * pathname , const char * arg , ...
                /* , (char *) NULL, char *const envp [] */ );
int execlp(const char * filename , const char * arg , ...
                /* , (char *) NULL */);

int execvp(const char * filename , char *const argv []);
int execv(const char * pathname , char *const argv []);
int execl(const char * pathname , const char * arg , ...
                /* , (char *) NULL */);
```

None of the above returns on success; all return –1 on error

Summary of differences between the *exec()* functions

Function	Specification of program file (–, *p*)	Specification of arguments (*v*, *l*)	Source of environment (*e*, –)
execve()	pathname	array	*envp* argument
execle()	pathname	list	*envp* argument
execlp()	filename + PATH	list	caller's *environ*
execvp()	filename + PATH	array	caller's *environ*
execv()	pathname	array	caller's *environ*
execl()	pathname	list	caller's *environ*

Signals and *exec( )*

- During an *exec()*, the text of the existing process is discarded. This text may include signal handlers established by the calling program. Because the handlers disappear, the kernel resets the dispositions of all handled signals to SIG_DFL .
- The dispositions of all other signals are left unchanged by an *exec()*.
- Signals should not be blocked or ignored across an *exec()* of an arbitrary program (Here, “arbitrary” means a program that we did not write) It is acceptable to block or ignore signals when *exec*ing a program we have written or one with known behavior with respect to signals.

Executing a Shell Command: *system()*

- The *system()* function allows the calling program to execute an arbitrary shell command.
- Since one of our assignments will be a "shell" implementation, we will concentrate on how to implement ***system()*** using *fork()*, *exec()*, *wait()*, and *exit()*.

```
#include <stdlib.h>

int system(const char *command);
```

The *system()* function creates a child process that invokes a shell to execute command.

system ()

The principal advantages of *system()* are simplicity and convenience:

- We don't need to handle the details of calling *fork()*, *exec()*, *wait()*, and *exit()*.
- Error and signal handling are performed by *system()* on our behalf.
- Because *system()* uses the shell to execute command, all of the usual shell processing, substitutions, and redirections are performed on command before it is executed. This makes it easy to add an "execute a shell command" feature to an application.

system ()

- The main cost of *system()* is inefficiency.
- Executing a command using *system()* requires the creation of at least two processes—one for the shell and one or more for the command(s) it executes—each of which performs an *exec()*.
- If efficiency or speed is a requirement, it is preferable to use explicit *fork()* and *exec()* calls to execute the desired program.

```
#include <sys/wait.h>
#include "print_wait_status.h"
#include "tlpi_hdr.h"
#define MAX_CMD_LEN 200

int main(int argc, char *argv[])
{
   char str[MAX_CMD_LEN];          /* Command to be executed by system() */
   int status;                     /* Status return from system() */

   for (;;) {                      /* Read and execute a shell command */
        printf("Command: ");
        fflush(stdout);
        if (fgets(str, MAX_CMD_LEN, stdin) == NULL)
            Break;                 /* end-of-file */

        status = system(str);
        printf("system() returned: status=0x%04x (%d,%d)\n",
                  (unsigned int) status, status >> 8, status & 0xff);

         if (status == -1) {
                errExit("system");
           }
         else {
          if (WIFEXITED(status) && WEXITSTATUS(status) == 127)
               printf("(Probably) could not invoke shell\n");
            else                          /* Shell successfully executed command */
                printWaitStatus(NULL, status);
             }
          }
exit(EXIT_SUCCESS);
}
```

Sample run:

```
$ ./t_system
Command: whoami
mtk
system() returned: status=0x0000 (0,0)
child exited, status=0
Command: ls | grep XYZ                    Shell terminates with the status of...
system() returned: status=0x0100 (1,0)    its last command (grep), which...
child exited, status=1                    found no match, and so did an exit(1)
Command: exit 127
system() returned: status=0x7f00 (127,0)
(Probably) could not invoke shell         Actually, not true in this case
Command: sleep 100
Type Control-Z to suspend foreground process group
[1]+  Stopped                 ./t_system
$ ps | grep sleep                         Find PID of sleep
29361 pts/6    00:00:00 sleep
$ kill 29361                              And send a signal to terminate it
$ fg                                      Bring t_system back into foreground
./t_system
system() returned: status=0x000f (0,15)
child killed by signal 15 (Terminated)
Command: ^D$                              Type Control-D to terminate program
```

!! Important !!

- Set-user-ID and set-group-ID programs should never use *system()* while operating under the program's privileged identifier.
- Even when such programs don't allow the user to specify the text of the command to be executed, the shell's reliance on various environment variables to control its operation means that the use of *system()* inevitably opens the door for a system security breach.

Implementing *system()*

To implement *system()*, we need to use *fork()* to create a child that then does an *execl()* to implement a command like

```
execl("/bin/sh", "sh", "-c", command, (char *) NULL);
```

To collect the status of the child created by *system()*, we use a *waitpid()* call that specifies the child's process ID.

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

int system(char *command)
{
  int status;
  pid_t childPid;

   switch (childPid = fork()) {
   case -1:                          /* process creation error */
      return -1;

   case 0: /* Child */
    execl("/bin/sh", "sh", "-c", command, (char *) NULL);
    _exit(127);                      /* Failed exec */

   default: /* Parent */
     if (waitpid(childPid, &status, 0) == -1)
        return -1;
     else
        return status;
    }
}
```

Using *wait()* would not suffice, as *wait()* waits for any child, could accidentally collect the status of some other child created by the calling process.

simple_system.c

Treating signals correctly inside system()

- What adds complexity to the implementation of *system()* is the correct treatment with signals.
- The first signal to consider is SIGCHLD . Suppose that the program calling *system()* is also directly creating children, and has established a handler for SIGCHLD that performs its own *wait()*. In this situation, when a SIGCHLD signal is generated by the termination of the child created by *system()*, it is possible that the signal handler of the main program will be invoked—and collect the child's status—before *system()* has a chance to call *waitpid()*.
- Therefore, *system()* must block delivery of SIGCHLD while it is executing

Treating signals correctly inside system()

- The other signals to consider are those generated by the terminal interrupt (usually Control-C) and quit (usually Control-\) characters, SIGINT and SIGQUIT , respectively
- SIGINT and SIGQUIT should be ignored in the calling process while the command is being executed.

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

int system(const char *command)
{
   sigset_t blockMask, origMask;
   struct sigaction saIgnore, saOrigQuit, saOrigInt, saDefault;
   pid_t childPid;
   int status, savedErrno;

    if (command == NULL)               /* Is a shell available? */
     return system(":") == 0;

   sigemptyset(&blockMask);            /* Block SIGCHLD */
   sigaddset(&blockMask, SIGCHLD);
   sigprocmask(SIG_BLOCK, &blockMask, &origMask);

   saIgnore.sa_handler = SIG_IGN;      /* Ignore SIGINT and SIGQUIT */
   saIgnore.sa_flags = 0;
   sigemptyset(&saIgnore.sa_mask);
   sigaction(SIGINT, &saIgnore, &saOrigInt);
   sigaction(SIGQUIT, &saIgnore, &saOrigQuit);
```

```
    switch (childPid = fork()) {
    case -1:                          /* fork() failed */
           status = -1;
           Break;                    /* Carry on to reset signal attributes */

    case 0: /* Child: exec command */
           saDefault.sa_handler = SIG_DFL;
           saDefault.sa_flags = 0;
           sigemptyset(&saDefault.sa_mask);

          if (saOrigInt.sa_handler != SIG_IGN)  sigaction(SIGINT, &saDefault, NULL);
          if (saOrigQuit.sa_handler != SIG_IGN) sigaction(SIGQUIT, &saDefault, NULL);

          sigprocmask(SIG_SETMASK, &origMask, NULL);

          execl("/bin/sh", "sh", "-c", command, (char *) NULL);
          _exit(127);                     /* We could not exec the shell */

     default: /* Parent: wait for our child to terminate */
            while (waitpid(childPid, &status, 0) == -1) {
                 if (errno != EINTR) {              /* Error other than EINTR */
                     status = -1;
                     break;                         /* So exit loop */
                    }
                 }
                 break;
        }
    /* Unblock SIGCHLD, restore dispositions of SIGINT and SIGQUIT */
   savedErrno = errno;                       /* The following may change 'errno' */
    sigprocmask(SIG_SETMASK, &origMask, NULL);
    sigaction(SIGINT, &saOrigInt, NULL);
    sigaction(SIGQUIT, &saOrigQuit, NULL);
   errno = savedErrno;
  return status;
}
```

Daemons

A daemon is a process with the following characteristics:

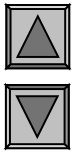
- It is long-lived. Often, a daemon is created at system startup and runs until the system is shut down.
- It runs in the background and has no controlling terminal. The lack of a controlling terminal ensures that the kernel never automatically generates any job-control or terminal-related signals (such as SIGINT , SIGTSTP , and SIGHUP) for a daemon.
- It is a convention (not universally observed) that daemons have names ending with the letter *d*.

Creating a Daemon

To become a daemon, a program performs the following steps:

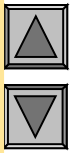
- Perform a *fork()*, after which the parent exits and the child continues.
- The child process calls *setsid()* to start a new session and free itself of any association with a controlling terminal
- If the daemon might later open a terminal device, then we must take steps to ensure that the device does not become the controlling terminal
- Clear the process umask to ensure that, when the daemon creates files and directories, they have the requested permissions.
- Change the process's current working directory, typically to the root directory
- Close all open file descriptors that the daemon has inherited from its parent
- After having closed file descriptors 0, 1, and 2, a daemon normally opens */dev/null* and uses *dup2()* (or similar) to make all those descriptors refer to this device

```
#ifndef BECOME_DAEMON_H                 /* Prevent double inclusion */
#define BECOME_DAEMON_H


/* Bit-mask values for 'flags' argument of becomeDaemon() */
#define BD_NO_CHDIR             01      /* Don't chdir("/") */
#define BD_NO_CLOSE_FILES       02      /* Don't close all open files */
#define BD_NO_REOPEN_STD_FDS    04      /* Don't reopen stdin, stdout, and stderr to
                                         *   /dev/null */
#define BD_NO_UMASK0               010  /* Don't do a umask(0) */
#define BD_MAX_CLOSE               8192 /* Maximum file descriptors to close if
                                         * sysconf(_SC_OPEN_MAX) is indeterminate */
int becomeDaemon(int flags);


#endif
```



```
#include  <sys/stat.h>
#include  <fcntl.h>
#include  "become_daemon.h"
#include  "tlpi_hdr.h"

int becomeDaemon(int flags)                     /* Returns 0 on success, -1 on error */
{
 int maxfd, fd;

   switch (fork()) {                    /* Become background process */
     case -1: return -1;
     case  0: break;                    /* Child falls through... */
    default: _exit(EXIT_SUCCESS);       /* while parent terminates */
    }

    if (setsid() == -1)             /* Become leader of new session */
        return -1;

     switch (fork()) {              /* Ensure we are not session leader */
     case -1: return -1;
     case 0: break;
     default: _exit(EXIT_SUCCESS);
     }

     if (!(flags & BD_NO_UMASK0))
     umask(0);                                  /* Clear file mode creation mask */
     if (!(flags & BD_NO_CHDIR))
     chdir("/");                                /* Change to root directory */
     if (!(flags & BD_NO_CLOSE_FILES)) {        /* Close all open files */
          maxfd = sysconf(_SC_OPEN_MAX);
           if (maxfd == -1)             /* Limit is indeterminate... */
          maxfd = BD_MAX_CLOSE;         /* so take a guess */

          for (fd = 0; fd < maxfd; fd++)
          close(fd);
}
```

```
    if (!(flags & BD_NO_REOPEN_STD_FDS)) {
        close(STDIN_FILENO);                /* Reopen standard fd's to /dev/null */

        fd = open("/dev/null", O_RDWR);

        if (fd != STDIN_FILENO)             /* 'fd' should be 0 */
            return -1;

        if (dup2(STDIN_FILENO, STDOUT_FILENO) != STDOUT_FILENO)
            return -1;

        if (dup2(STDIN_FILENO, STDERR_FILENO) != STDERR_FILENO)
            return -1;
    }
    return 0;
}
```



become_daemon.c

Daemon

- Daemons perform specific tasks, such as providing a network login facility or serving web pages. To become a daemon, a program performs a standard sequence of steps, including calls to *fork()* and *setsid()*.
- Where appropriate, daemons should correctly handle the arrival of the SIGTERM and SIGHUP signals. The SIGTERM signal should result in an orderly shutdown of the daemon, while the SIGHUP signal provides a way to trigger the daemon to reinitialize itself by rereading its configuration file and reopening any log files it may be using.