Lecture 9 - Daemon Processes (Chapter 13)

- Daemon Characteristics

  Daemons are processes that live for a long time.
  They are often started when the system is booted, and terminate only on system shutdown
  They general run in the background (i.e. have no controlling terminal)

  Examples of daemons: syslogd, cron, sendmail, update (syncer), inetd, etc.

- Coding Rules

  1. Call **fork** and have the parent exit.

     This makes the shell assume the command has finished.

     In addition, the child inherits the PGID of the parent but has a different PID. This means it cannot be the process group leader.

  2. Call **setsid** to create a new session.

     This makes our new process the session leader of the new session, and the process group leader of a new process group, and it will have no controlling terminal.

  3. Change the current working directory to the root directory.

     Since daemons run for long periods of time, we do not want to prevent somone from unmounting an secondary filesystem just because our daemon has its current working directory set there.

     This also helps encourage good coding practices since many people rely on relative paths for their programs to work correctly. By change directories to "/", we are forced to think through these issues.

  4. Set the file creation mask to 0.

     The file mode creation mask that's inherited could be set to deny certain permissions, and we may want to be specific about the permissions of created files.

  5. Close all unneeded file descriptors

     This prevents the daemon from holding open files for long periods of time.

Example:

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

        if ((pid = fork()) < 0)
                return(-1);
        else if (pid > 0)
                exit(0);                /*  exit the parent */

        setsid();                       /*  the child creates a new session */

        chdir("/");                     /*  change our working directory */

        umask(0);                       /*  Reset the file creation mask */

        return(0);                      /*  Return to the caller */
}
```

- Error Logging

  We cannot just write error messages for our daemons to stderr since there is no controlling terminal.

  We could redirect our error messages to a specific file for each daemon, but this can be cumbersome to administer if there is a large number of daemons to monitor.

- 4.3+ BSD "syslog" facility (see picture on page 429 - Figure 13.2)

  Methods to generate log messages:

  1. Kernel routines can call the "log" function.
  2. Most user processes call the syslog(3) function to generate log messages. This causes messages to be sent to the datagram socket /dev/log.
  3. A user process on this host, or some other host, can send log messages to UDP port 514.

  Normally, the syslogd daemon reads all three forms of log messages.

  This daemon reads a configuration file on start-up (/etc/syslog.conf) that determines where different classes of messages are to be sent.

  For example, urgent messages can be sent to the system administrator via email and printed on the console, while warnings may be logged to a file.

```
void    openlog(char *ident, int option, int facility);
void    syslog(int priority, char * format, …);
void    closelog(void);
```

Calling openlog and closelog is optional. If openlog is not used, it is called automatically the first time syslog is called.

Calling openlog let's us specify "ident" which is added to each log message. This is often the name of the program.

Options:

LOG_CONS	If log message cannot be sent to syslogd, write to console.
LOG_NDELAY	Open connection to syslogd immediately
LOG_PERROR	Write log message to syslog AND stderr
LOG_PID	Log the PID with each messages

The "facility" argument to openlog is taken from Figure 13.4 (page 423). This is used as an indicator in the configuration file. For example, the facility might be LOG_AUTH, LOG_CRON, LOG_KERN, LOG_MAIL, etc.

When calling syslog, the "priority" argument is a combination of the "facility", and the "level" (see Figure 13.5). For example, the "level" might be LOG_EMERG, LOG_ALERT, LOG_CRIT, LOG_WARNING, LOG_DEBUG, etc.

The "format" argument and any remaining arguments are passed to vsprintf for formatting. Any occurrence of "%m" is the format are first replaced with the error message string (stderr) corresponding to the value of errno.

The "logger" program is also provided as a way to send log messages to the syslog facility. It is intended to allow shell scripts to generate log messages.

Example:

```
openlog("lprps",  LOG_PID, LOG_LPR);
syslog(LOG_ERR, "open error for %s: %m", filename);
```

which is equivalent to

```
syslog(LOG_ERR | LOG_LPR, "open error for %s: %m", filename);
```

Advanced I/O (Chapter 14)

- Non-blocking I/O

"Slow" system calls are those that can block forever:

- Reads from files that can block if data isn't present (pipes, terminals, network)
- Writes to those same files if data cannot be accepted immediately
- Opens of files under certain conditions (i.e. when waiting on modem, etc)
- Reads and writes of files under mandatory record locking
- Certain ioctl operations
- Some of the IPC functions (chapter 15)

Non-blocking I/O allows us to issue an I/O operation (open, read, write) and not have it block forever. If operation cannot be completed, return is made immediately with error.

Methods for using non-blocking I/O:

1. Call open with O_NONBLOCK.
2. Use fcntl on an open file descriptor to turn on O_NONBLOCK

Example (non-blocking on stdout, see page 444 for output):

```
int main(void)
{
        char    buf[100000], *ptr;
        int     ntowrite, nwrite;

        ntowrite = read(0, buf, sizeof(buf));
        fprintf(stderr, "read %d bytes\n", ntowrite);
        set_fl(1, O_NONBLOCK);              /* set_fl on page 81 */

        for (ptr = buf; ntowrite > 0; ) {
                errno = 0;

                nwrite = write(1, ptr, ntowrite);
                fprintf(stderr, "nwrite = %d, errno = %d\n", nwrite, errno);

                if (nwrite > 0) {
                        ptr += nwrite;
                        ntowrite -= nwrite;
                }
        }
        clr_fl(1, O_NONBLOCK);
        exit(0);
}
```

- Record locking

  Record locking is the ability of a process to prevent other processes from modifying a region of a file, while the first process is reading or modifying that portion of the file.

  UNIX really does not have a concept of records in files, so record locking should really be called "range locking".

  Note: See Figure 14.2 for different locking mech. supported under different UNIX's

  fcntl record locking:

```
int     fcntl(int filedes, int cmd, …  /* struct flock *flockptr */ );

struct flock {
        short   l_type;         /*  F_RDLCK, F_WRLCK, or F_UNLCK */
        off_t   l_start;        /*  starting offset */
        short   l_whence;       /*  SEEK_SET, SEEK_CUR, or SEEK_END */
        off_t   l_len;          /*  length, 0 means to lock to EOF */
        pid_t   l_pid;          /*  returned with F_GETLK */
};
```

Types of locks:

F_RDLCK	shared read lock
F_WRLCK	exclusive write lock
F_UNLCK	unlock

Has / request	**Read lock**	**Write lock**
No locks	OK	OK
1+ read locks	OK	Denied
1 write lock	Denied	denied

Types of commands:

Command	Description
F_GETLK	Determine if lock described by "flockptr" is blocked by some other lock. If a lock exists that would prevent ours from being created, it replaces "flockptr".
F_SETLK	Set the lock described by "flockptr".
F_SETLKW	Blocking version of F_SETLK.

Implied inheritance and release of locks:

1. Locks are associated with a process and a file. A) When a process terminates, all its locks are released. B) Whenever a descriptor is closed, any locks on this file referenced by that descriptor for that process are released.

2. Locks are never inherited by the child across fork.

3. Locks may be inherited by a new program across an exec.

Advisory versus mandatory locking:

Advisory locking does not prevent some process from violating the lock; therefore, it only works well in a cooperating process environment.

Mandatory locking causes the kernel to check every open, read, and write to verify that the calling process is not violating a lock on the file being accessed.

Mandatory locking is enabled for a particular file by turning on SGID and turning off group execute.

- Streams

A stream provides a general way to provide a full-duplex path between a user process and a device driver (or pseudo device driver).

Beneath the stream head, we can push processing modules onto the stream (by using ioctl). Any number of processing modules can be pushed onto a stream.

- I/O multiplexing

In some cases, we might want to read or write to multiple file descriptors concurrently (i.e. not to block on any one file descriptor).

One way to resolve this problem is to fork copies of our process to handle I/O for each file descriptor, but this can get messy.

Another way to resolve the problem would be to use non-blocking I/O and loop through the file descriptors one at a time checking for data, then sleep some amount of time, and loop through them again (called "polling").

Polling wastes lots of CPU time since we are constantly calling read even though there will only be data ready a fraction of the time.

The real solution is to use "select" or "poll".

- Select function

The select function lets us do I/O multiplexing, by passing as arguments:

The file descriptors we are interested in.
What conditions we are interested in (read, write, or exception).
How long we are willing to wait.

On return, we get:

The total count of ready file descriptors
Which descriptors are ready for each condition

```
int     select(int maxfd, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
                Struct timeval *tvptr);
```

Maxfd:

The first argument is the maximum file descriptor we are interested in plus 1.

Timeouts:

```
struct timeval {
        long    tv_sec;
        long    tv_usec;
};
```

tvptr == NULL	wait forever (or until signal) when signal caught, errno = EINTR
tvptr == 0	don’t wait at all
tvptr != 0	wait specified number of seconds and microsec (timeout returns 0)

File Descriptor Sets:

```
FD_ZERO(fd_set *fdset);
FD_SET(int fd, fd_set *fdset)
FD_CLR(int fd, fd_set *fdset)
FD_ISSET(int fd, fd_set *fdset)
```

Return values:

-1	error (i.e. signal caught)
0	no descriptors ready (timeout)
>0	number of file descriptors ready

- Example (selpipe.c):

```
struct pipetype {
    char    *name;
    int     fd;
    int     num_char;
} p[MAXPIPES];

int     maxpipes = MAXPIPES;
int     numpipes = MAXPIPES;

int process(int argc, char **argv)
{
    char    line[MAXLINE];              /*  input line */
    int     ofd[MAXPIPES];              /*  filedes for named pipe */
    int     maxfd = 0;                  /*  maximum filedes */
    int     fd, num, i, status;
    fd_set  readfs, exceptfs, savefs;   /*  filedes sets for testing */

    FD_ZERO(&savefs);

    for (i = 1; i < argc; i++)
    {
        /*  Open the named pipe */

        if ((fd = open(argv[i], O_RDONLY, 0)) < 0)
        {
            perror("process: open");
            return FALSE;
        }

        p[fd].name = (char *) malloc(strlen(argv[i])+1);
        strcpy(p[fd].name, argv[i]);
        fprintf(stderr, "opening pipe %s\n", p[fd].name);

        if (fd > maxfd)
            maxfd = fd;

        FD_SET(fd, &savefs);
    }
```

```
        while (numpipes > 0)
        {
                readfs = savefs;
                exceptfs = savefs;

                status = select(maxfd+1, &readfs, NULL, NULL, NULL);

                if (status < 0)
                {
                        if (errno == EINTR)
                                continue;

                        perror("process: select");
                        return FALSE;
                }

                fprintf(stderr, "select returned %d\n", status);

                for (i = 0; i < maxfd + 1; i++)
                {
                        if (FD_ISSET(i, &readfs))
                        {
                                num = read(i, line, MAXLINE);

                                if (num == 0)
                                {
                                        numpipes--;
                                        close(i);
                                        FD_CLR(i, &savefs);
                                        continue;
                                }

                                p[i].num_char += num;

                                line[num] = 0;
                                fprintf(stderr, "%s: %s\n", p[i].name, line);
                        }
                }
        }

        for (i = 1; i < argc; i++)
                close(p[i].fd);

        return TRUE;
}
```

- Poll function

  int poll(struct pollfd fdarray[], unsigned long nfds, int timeout);

  Returns: count of ready descriptors, 0 on timeout, -1 on error

```
struct pollfd {
        int             fd;
        short           events;         /*  events of interest */
        short           revents;        /*  events that occurred */
};
```

  Events can be any value in Figure 12.15 on page 401:

POLLIN	read
POLLOUT	write
POLLERR	error (exception)

  Time values:

timeout == INFTIM	infinite timeout (until signal is caught)
timeout == 0	don't wait
timeout > 0	wait "timeout" milliseconds

- Readv and writev

  Readv and writev let us read into and write from multiple non-contiguous buffers in a single function call. These are also called "scatter read and gather write".

```
ssize_t         readv(int filedes, const struct iovec iov[], int iocnt);
ssize_t         writev(int filedes, const struct iovec iov[], int iocnt);

struct iovec {
        void    *iov_base;
        size_t  iov_len;
};
```

  Note: These use a good deal less CPU time since we are making less system calls.

- Readn and writen

```
ssize_t writen(int fd, const void *vptr, size_t n)
{
        size_t          nleft;
        ssize_t         nwritten;
        const char      *ptr;

        ptr = vptr;
        nleft = n;

        while (nleft > 0)
        {
                if ((nwritten = write(fd, ptr, nleft)) <= 0)
                        return(nwritten);

                nleft -= nwritten;
                ptr += nwritten;
        }

        return(n);
}
```

Note: See also readn on page 408

- Memory mapped I/O

  Memory mapped I/O lets us map a file on disk into a buffer in memory so that when we fetch bytes from the buffer, the corresponding bytes of the file are read. Also, when we store data in the buffer, the corresponding bytes are automatically writen to the file.

```
caddr_t         mmap(caddr_t addr, size_t len, int prot, int flag, int filedes, off_t off);
int             munmap(caddr_t addr, size_t len);
```

  Returns starting address of mapped region if OK, -1 on error

  "addr" is where we want the region to start (0 lets the system pick).
  "filedes" is the file descriptor specifying the file that is to be mapped.
  "len" is the number of bytes to map, "off" is the starting offset in the file

  "prot" specifies the protection of the mapped region:

PROT_READ	region can be read
PROT_WRITE	region can be written
PROT_EXEC	region can be executed
PROT_NONE	region cannot be accessed

Example (mmap_test.c):

```
int main(int argc, char **argv);
{
        int             fdin, fdout;
        char            *src, *dst;
        struct stat     statbuf;

        if (argc != 3)
        {
                fprintf(stderr, "usage: %s, <fromfile> <tofile>\n", argv[0]);
                exit(1);
        }

        if ((fdin = open(argv[1], O_RDONLY)) < 0)
                exit(1);

        if ((fdout = open(argv[2], O_RDWR | O_CREAT | O_TRUNC,
                FILE_MODE)) < 0)
                exit(1);

        if (fstat(fdin, &statbuf) < 0)
                exit(1);

        if (lseek(fdout, statbuf.st_size - 1, SEEK_SET) == -1)
                exit(1);

        if (write(fdout, "", 1) != 1)
                exit(1);

        if ((src = mmap(0, statbuf.st_size, PROT_READ, MAP_FILE | MAP_SHARED,
                fdin, 0)) == (caddr_t) -1)
                exit(1);

        if ((dst = mmap(0, statbuf.st_size, PROT_READ | PROT_WRITE,
                MAP_FILE | MAP_SHARED, fdout, 0)) == (caddr_t) -1)
                exit(1);

        memcpy(dst, src, statbuf.st_size);

        exit(0);
}
```