COMP421
Unix Environment for Programmers
Lecture 14: Daemons

Jeff Wiegley, Ph.D.
Computer Science

jeffw@csun.edu

10/17/2005

> ‘‘Maxwell’s daemon was an imaginary agent which helped sort molecules of different speeds and worked tirelessly in the background.’’
>
> –*Fernando J. Corbato, 1963*

Services

Unix is traditionally a server operating system.

- Held huge market share in the server market until the mid to late '90s. (Windows products are now competitive in abilities and market share is slipping.)[a]
- Two main functions:
  1. Multiuser/process batch handling for large, dumb terminal farms. (VT100)
  2. Constant availability of services such as database, web, mail, DNS and printing.

To provide "services", Unix uses processes termed *daemons*.

[a] Due to the popularity and cost of Linux, FreeBSD and OpenBSD however, Unix is gaining market share on the desktop.

Daemons

A Daemon process is not really any different from a normal process. A few programming procedures are simply done to make the process behave in a certain manner:

- The process shouldn't create files (or other resources) with unacceptable/insecure privileges.
- The process cannot have a "controlling" terminal attached to it (such as a shell) because we want it to run in the "background"
- The process should not be a member of any other process group.
- The process should have a known current working directory.
- The process shouldn't have any unneeded file descriptors open.
- If desired, the process should provide some sort of logging of actions or errors that it encountered while running.
- The process should enforce *least privileges* whenever possible.

Insecure privileges:

Unix processes have a *umask* associated with them that determine the default permissions that will be created for new files.

The umask is a negation of privileges.

A umask of 0777 for instance would create files that nobody (expect) root could do anything with.

0077 On the other hand would create files/directories with permissions set to "-wrx——".

The umask is inherited from the parent process when the `fork()` is performed.

It will be necessary for the daemon process to change its umask to insure least-privileges.

`umask(0077);` will do the trick. This umask should be as strict as possible without preventing the daemon from establishing permissions that should be allowed.

Eliminating the Controlling terminal:

The controlling terminal is the terminal device attached to a process.

It's a bit different than just stdin and stdout. Closing stdin and stdout does not detach the process from the controlling terminal. This has to be done manually.

```
pid = fork();
if (pid<0)  exit(1); // fork failed. exit with an error of some kind.
if (pid!=0) exit(0); // parent exits, child is adopted by init
// child continues and has no controlling TTY attached.
```

But that's not quite enough...

Process group leader:

The child still keeps the inherited process group from the fork and this is not acceptable.

```
setsid();
```

creates a new session with this process as the session leader.

Now the problem is that a future `open()` could deliver a controlling TTY. So we need to take care of that possibility.

Future Controlling TTYs:

We fork, yet again, to guarantee that no future TTY can be a controlling TTY.

```
  sa.sa_handler = SIG_IGN;
sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;
if (sigaction(SIGHUP, &sa, NULL) < 0)
   exit(2);  // ARGH! HUP is not ignorable, fatal error.
pid = fork();
if (pid<0)  exit(1); // fork failed. exit with an error of some kind.
if (pid!=0) exit(0); // parent exits, child is again adopted by init
// child continues can never have a controlling TTY attached.
```

Current Working Directory:

All process entries have a current working directory. Relative pathnames for `open()` or other path aware functions function relative to the current working directory.

Servers and daemons stay up for very, very long times. (One of mine has been up for 293 consecutive days.)

But directories, like files, can change.

Therefore daemons should establish a directory as the current working directory that is known not to change.

```
chdir("/");
```

Other, more specific, directory selections would be acceptable.
"/var/www" for an HTTP daemon, for example.

Close all unused file descriptors:

We don't need any files (stdin, stdout and stderr) don't really have a terminal to send or receive anything from.

The following loop insures that all file descriptors are closed. (including stdin, stdout and stderr.

```
for (fd=getdtablesize()-1;fd>=0;fd--)
  close(fd);
```

Not all Unix-like systems have `getdtablesize()` functions.

Other functions exist or you can manually just close 0, 1 and 2 since they will probably be the only open file descriptors.

A good idea for stdin, stdout and stderr...:

Your daemon may later fork (like a shell) and execute other commands that would die if there was no stdin, stdout or stderr attached.

You can use `/dev/null` as a handy file to attach these to:

The following will handle this nicely.

```
int fd = open("/dev/null",O_RDWR);
if (fd<0)
   exit(3); //OOOPS!
if (fd!=0)
  {
    dup2(fd, 0);
    close(fd);
  }
dup(0,1);
dup(0,2);
```

Logging:

Daemons run in the background but provide essential services.

It is important to keep track of what they are doing.

There are two basic methods of handling logging:

1. The daemon handles it all itself.

   Pros: complete control of log location and information.

   Cons: You have to handle everything yourself. Logs can fill up.

2. The daemon utilizes the *syslog* daemon to handle the logging.

   Pros: File descriptor/read/writes are handled for you.
   log locations can be changed easily.
   various information can be ignored based on service class.

   Cons: Limited logging facilities.
   Requires a seperate daemon to handle to logging.

handling logging yourself:

Straight forward:

```
FILE *log = fopen("/var/log/mydaemon","r");
```

Now just simply use `fprintf(log,"...",...)` to print out whatever information you wish to log.

Handling logging through syslog:

```
#include <syslog.h>


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

`openlog()` and `closelog()` are optional but should be used if for no other reason than without `openlog()` the `ident` of the logged messages will be NULL.

Syslog shuttles messages to various system log files based on two parameters: The *facility* and the *priority*.

Facility is a general item like: LOG_AUTH, LOG_CRON, LOG_DAEMON, LOG_FTP, etc.

The priority is a disjunction of LOG_EMERG, LOG_ALERT, LOG_CRIT, LOG_ERR, LOG_WARNING, LOG_NOTICE, LOG_INFO or LOG_DEBUG.

Syslog Behavior:

a syslog.conf configuration file dictates what facilities and priorities should be logged or ignored and to what files they should be appended.

The configuration file also determines the location of files and which files ceratin facility/priority messages should be appended.

Logging can even be configured to be sent out to be collected by a different machine entirely. (So for instance, all logging information for all of IBM's servers could be collected and stored on a single machine.)

This allows a daemon to issue copious information and the system administrator can configure syslog to log or ignore certain classes of information.

This makes debugging both:

- effective: The debugging level can be turned up by simply reconfiguring syslog.
- Economical: The debugging level can be turned down to conserve disk space when the daemon is running as expected.

Least privileges:

Processes have three user(group) ids associated with them:

1. The *real* user id, which is the ID of the user that executed the program.
2. The *effective* user id, which is the user id that should be considered in effect (sort of). This user id is established by the setuid bit belonging to the program.[a]
3. *saved set*-user id. which is a copy of what the effective UID was to start with.

You need elevated privileges for certain things such as opening privileges network ports (like port 80) or for creating log files in the `/var/log/` directory.

But you don't want elevated privileges all the time because you don't want virus to gain these privileges if they take over your program.

[a] This is why it is very, very bad to have setuid programs. They establish an effective user id of root!

Dropping privileges (when you are root):

Programs like apache start life with a real, effective and saved set user id of "root" and have god-level privileges.

When you are done setting up stuff that requires such power you should relinquish control:

```
struct passwd pwbuf;
struct passwd *pwbufp;
getpwnam_r("nobody", &pwbuf, buf, 1024, &pwbufp);
setuid(pwbuf.pw_uid);
```

The same thing would have to be done with the group id as well.

Since the process was the superuser, after the call to `setuid()` all three user ids are set to the user "nobody" and can **never** be switched back to root ever again.

Dropping privileges (when you are not root):

When the process begins life as a user other than root the code is the same for changing effective user and group IDs but the rules change a bit:

- `setuid()` only modifies the effective UID.
- The effective UID can only be changed to either:
  1. The real UID, or
  2. The saved set-user id.

Therefore, a normal user can only switch the effective UID between his real user id and the user id of the owner of the setuid program in question.

Because it didn't start as user root, the program is free to drop and regain its privileges as many times as it want and whenever it wants.

But an advantage to starting life as root is that you can drop privileges to any other user. So for instance, apache2 can run as the user "www" or "nobody".

Putting it all together:

```
int daemonize(const char *ident)
{
  int fd;
  struct sigaction sa;
  struct passwd pwbuf;
  struct passwd *pwbufp;
  struct group grbuf;
  struct group *grbufp;
  char buf[1024];

  // set a decent umask.
  umask(0); // for more security use 0700 or whatever you need.


  // switch to a reasonable current working directory
  chdir("/");
```

```
// close all file descriptors
for (fd=getdtablesize()-1;fd>=0;fd--)
  close(fd);

// eliminate the controlling terminal
pid = fork();
if (pid<0)  return(1); // fork failed. return from daemon that failed.
if (pid!=0) exit(0); // parent exits, child is adopted by init

// establish self as new seesion group leader.
setsid();

//Make sure we can never have a controlling terminal
sa.sa_handler = SIG_IGN;
sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;
if (sigaction(SIGHUP, &sa, NULL) < 0)
```

```
     return(2);  // ARGH! HUP is not ignorable, fatal error.
pid = fork();
if (pid<0)  return(1); // fork failed. return from daemon that failed.
if (pid!=0) exit(0); // parent exits, child is again adopted by init

//optional: attach stdin, stdout, stderr to /dev/null
fd = open("/dev/null",O_RDWR);
if (fd<0)  return(3); //OOOPS!
if (fd!=0) dup2(fd, 0), close(fd);
dup(0,1);
dup(0,2);

openlog(ident, 0, LOG_DAEMON);
// now the program can call
// syslog(LOG_INFO|LOG_DEBUG, "hey I want to show you something");
// whenever it wants to tell us something.

//This should be done elsewhere/later once elevated privileges are
```

```
  // no longer required but is shown here for completeness:
  if (getpwnam_r("nobody", &pwbuf, buf, 1024, &pwbufp)==NULL)
    return 4;
  setuid(pwbuf.pw_uid);

  if getgrnam_r("nobody", &grbuf, buf, 1024, &grbufp)==NULL)
    return 5;
  setgid(grbuf.pw_uid);
}
```