Chapter 30 – Advanced Techniques

Background

Production servers disconnect themselves from the terminal and run in background.

Method 1) `mutd &`
Advantage: no extra code
Disadvantage: if you forget the `&`

Method 2) The server backgrounds itself.

```
/* This code backgrounds the server */
i = fork ();
if (i < 0 ) {
  exit(1); /* Error occurred in fork */
}
if (i != 0 ) {
  /* parent exits */
  exit(0);
}
/* Child continues */
```

Forks a child.
The child is in the background.
The parent is in the forground.
The parent exits.
The child continues.

More techniques

1) A good idea to close all unused descriptors.
Did you start from the command line or with an execv.
With execv you inherit open descriptors, close them all just incase.

```
nfds = getdtablesize();
for (i=0; i < nfds ; i++ ) close(i);
```

Technically you fail if the descriptor isn't open, but you don't care. WARNING: if you are printing debugging information, do not close descriptor 2.

2) A good idea to disconnect from the tty.

```
fd = open("/dev/tty", O_RDRW);
ioctl(fd, TTIOCNOTTY, 0);
close(fd);
```

Open the terminal that controls the process.
Use `ioctl` to disconnect from that terminal.
close the file descriptor (see 1 above)

3) A good idea to run in a known directory.

If you crash, that is where the core dump goes.

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

4) Process groups

All processes belong to process groups.
Usually the group includes the parent and child processes.

Some signals are sent to all processes in a group.
You can start a new process group.

```
setpgrp(0, getpid());
```

Put me in a new process group.
Don't send me anyone elses signals.

5) Make sure only one server is running.

Use flock.

```
lf = open("/usr/mydir/my.lock", O_RDWR|O_CREAT,
          0640);
if (lf < 0) exit(1); /*file system failure */
if (flock(lf, LOCK_EX|LOCK_NB)) exit(0);
```

Exit if we can't get an flock using the lock file.

Recall flocks are removed if a server dies.

6) Put your PID somewhere so it's easy to find who the server is. (Why not use the lock file.)

```
char pbuf[10]; /*pid buf*/
sprintf(pbuf, "%d\n", getpid());
write(lf, pbuf, strlen(pbuf));
```

Put your pid into a string buffer,
write the string into the lock file.

Any wanting the PID of the server
can look in the lock file

7) If you fork processes don't forget the wait
(see the reaper from browserd)

8) Block signals you don't want

```
signal(SIGINT, SIG_IGN);
```

9) If you have a config file, use a signal to re-read it.

```
signal(SIGHUP, reload);
...
reload() {
  int fd;
  fd = open("mut.conf", O_RDONLY );
  /* Read new config into globals */
  close(fd);
}
```

Locating a Server

If you are looking for a server on the same subnet you can automatically locate it.

Use a combination of the multiprotocol server
(does both TCP and UDP)
and broadcasts.

1) Client sends a broadcast.

2) Server responds to the broadcast.

3) Client receives servers response; it now has the address of the server.

4) Client connects (TCP) to server.

Syslog

Problem: There is no screen for a server to print to.
Solution: Log a message to the system log.
Use a generalized messaging system that allows the administrator to say where the messages go.

```
#include <syslog.h>
openlog("ProgName",options,facility);
syslog(loglevel,"message");
```

`openlog`: called once per program
Establishes syslog defaults.
Name of program.
options–such as include the pid with the message.
facility–default log to use

`syslog`: called for each message to be logged
Sends a message to `syslogd`.
log level–the importance of this message
message–to be recorded.

`syslogd`–gets the message, handles it as defined by the configuration file `/etc/syslog.conf`.

```
openlog("MyServer",LOG_PID,LOG_USER);
syslog(LOG_NOTICE,"Failed");
```

Log the process ID with the message.
Log as facility `user` Level is `notice`