Unix Daemon Server Programming

Introduction

Unix processes works either in foreground or background. A process running in foreground interacts with the user in front of the terminal (makes I/O), whereas a background process runs by itself. The user can check its status but he doesn't (need to) know what it is doing. The term 'daemon' is used for processes that performs service in background. A server is a process that begins execution at startup (not neccessarily), runs forever, usually do not die or get restarted, operates in background, waits for requests to arrive and respond to them and frequently spawn other processes to handle these requests.

Readers are suppossed to know Unix fundamentals and C language. For further description on any topic use "man" command (I write useful keywords in brackets), it has always been very useful, trust me :)) Keep in mind that this document does not contain everything, it is just a guide.

1) Daemonizing (programming to operate in background) [fork]

First the fork() system call will be used to create a copy of our process(child), then let parent exit. Orphaned child will become a child of init process (this is the initial system process, in other words the parent of all processes). As a result our process will be completely detached from its parent and start operating in background.

```
i=fork();
if (i<0) exit(1); /* fork error */
if (i>0) exit(0); /* parent exits */
/* child (daemon) continues */
```

2) Process Independency [setsid]

A process receives signals from the terminal that it is connected to, and each process inherits its parent's controlling tty. A server should not receive signals from the process that started it, so it must detach itself from its controlling tty.

In Unix systems, processes operates within a process group, so that all processes within a group is treated as a single entity. Process group or session is also inherited. A server should operate independently from other processes.

```
setsid() /* obtain a new process group */
```

This call will place the server in a new process group and session and detach its controlling terminal. (setpgrp() is an alternative for this)

3) Inherited Descriptors and Standart I/0 Descriptors [gettablesize,fork,open,close,dup,stdio.h]

Open descriptors are inherited to child process, this may cause the use of resources unneccessarily. Unneccesarry descriptors should be closed before fork() system call (so that they are not inherited) or close all open descriptors as soon as the child process starts running.

```
for (i=getdtablesize();i>=0;--i) close(i); /* close all descriptors */
```

There are three standart I/O descriptors: standart input 'stdin' (0), standart output 'stdout' (1), standart error 'stderr' (2). A standard library routine may read or write to standart I/O and it may occur to a terminal or file. For safety, these descriptors should be opened and connectedthem to a harmless I/O device (such as /dev/null).

```
i=open("/dev/null",O_RDWR); /* open stdin */
dup(i); /* stdout */
dup(i); /* stderr */
```

As Unix assigns descriptors sequentially, fopen call will open stdin and dup calls will provide a copy for stdout and stderr.

4) File Creation Mask [umask]

Most servers runs as super-user, for security reasons they should protect files that they create. Setting user mask will pre vent unsecure file priviliges that may occur on file creation.

```
umask(027);
```

This will restrict file creation mode to 750 (complement of 027).

5) Running Directory [chdir]

A server should run in a known directory. There are many advantages, in fact the opposite has many disadvantages: suppose that our server is started in a user's home directory, it will not be able to find some input and output files.

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

The root "/" directory may not be appropriate for every server, it should be choosen carefully depending on the type of the server.

6) Mutual Exclusion and Running a Single Copy [open,lockf,getpid]

Most services require running only one copy of a server at a time. File locking method is a good solution for mutual exclusion. The first instance of the server locks the file so that other instances understand that an instance is already running. If server terminates lock will be automatically released so that a new instance can run. Recording the pid of the running instance is a good idea. It will surely be efficient to make 'cat mydaamon.lock' instead of 'ps -ef|grep mydaemon'

```
        lfp=open("exampled.lock",O_RDWR|O_CREAT,0640);
        if (lfp<0) exit(1); /* can not open */
        if (lockf(lfp,F_TLOCK,0)<0) exit(0); /* can not lock */
        /* only first instance continues */

        sprintf(str,"%d\n",getpid());
        write(lfp,str,strlen(str)); /* record pid to lockfile */
```

7) Catching Signals [signal,sys/signal.h]

A process may receive signal from a user or a process, its best to catch those signals and behave accordingly. Child processes send SIGCHLD signal when they terminate, server process must either ignore or handle these signals. Some servers also use hang-up signal to restart the server and it is a good idea to rehash with a signal. Note that 'kill' command sends SIGTERM (15) by default and SIGKILL (9) signal can not be caught.

```
        signal(SIG_IGN,SIGCHLD); /* child terminate signal */
```

The above code ignores the child terminate signal (on BSD systems parents should wait for their child, so this signal should be caught to avoid zombie processes), and the one below demonstrates how to catch the signals.

```
        void Signal_Handler(sig) /* signal handler function */
        int sig;
        {
                switch(sig){
                        case SIGHUP:
                                /* rehash the server */
                                break;
                        case SIGTERM:
                                /* finalize the server */
                                exit(0)
                                break;
                }
        }

        signal(SIGHUP,Signal_Handler); /* hangup signal */
        signal(SIGTERM,Signal_Handler); /* software termination signal from kill */
```

First we construct a signal handling function and then tie up signals to that function.

8) Logging [syslogd,syslog.conf,openlog,syslog,closelog]

A running server creates messages, naturally some are important and should be logged. A programmer wants to see debug messages or a system operator wants to see error messages. There are several ways to handle those messages.

Redirecting all output to standard I/O : This is what ancient servers do, they use stdout and stderr so that messages are written to console, terminal, file or printed on paper. I/O is redirected when starting the server. (to change destination, server must be restarted) In fact this kind of a server is a program running in foreground (not a daemon).

```
        # mydaemon 2> error.log
```

This example is a program that prints output (stdout) messages to console and error (stderr) messages to a file named "error.log". Note that this is not a daemon but a normal program.

Log file method : All messages are logged to files (to different files as needed). There is a sample logging function below.

```
        void log_message(filename,message)
        char *filename;
        char *message;
        {
        FILE *logfile;
                logfile=fopen(filename,"a");
                if(!logfile) return;
                fprintf(logfile,"%s\n",message);
                fclose(logfile);
        }

        log_message("conn.log","connection accepted");
        log_message("error.log","can not open file");
```

Log server method : A more flexible logging technique is using log servers. Unix distributions have system log daemon named "syslogd". This daemon groups messages into classes (known as facility) and these classes can be redirected to different places. Syslog uses a configuration file (/etc/syslog.conf) that those redirection rules reside in.

```
        openlog("mydaemon",LOG_PID,LOG_DAEMON)
        syslog(LOG_INFO, "Connection from host %d", callinghostname);
        syslog(LOG_ALERT, "Database Error !");
        closelog();
```

In openlog call "mydaemon" is a string that identifies our daemon, LOG_PID makes syslogd log the process id with each message and LOG_DAEMON is the message class. When calling syslog call first parameter is the priority and the rest works like printf/sprintf. There are several message classes (or facility names), log options and priority levels. Here are some examples :

Message classes : LOG_USER, LOG_DAEMON, LOG_LOCAL0 to LOG_LOCAL7
Log options : LOG_PID, LOG_CONS, LOG_PERROR
Priority levels : LOG_EMERG, LOG_ALERT, LOG_ERR, LOG_WARNING, LOG_INFO

About

This text is written by Levent Karakas mailto:levent@mis.boun.edu.tr. Several books, sources and manual pages are used. This text includes a sample daemon program (compiles on Linux 2.4.2, OpenBSD 2.7, SunOS 5.8, SCO-Unix 3.2 and probably on your flavor of Unix). You can also download plain source file : exampled.c. Hope you find this document useful. We do love Unix.

```
/*
UNIX Daemon Server Programming Sample Program
Levent Karakas <levent@mis.boun.edu.tr> May 2001

To compile:     cc -o exampled examped.c
To run:         ./exampled
To test daemon: ps -ef|grep exampled (or ps -aux on BSD systems)
To test log:    tail -f /tmp/exampled.log
To test signal: kill -HUP `cat /tmp/exampled.lock`
To terminate:   kill `cat /tmp/exampled.lock`
*/

#include <stdio.h>
#include <fcntl.h>
#include <signal.h>
#include <unistd.h>

#define RUNNING_DIR    "/tmp"
```

```
#define LOCK_FILE       "exampled.lock"
#define LOG_FILE        "exampled.log"

void log_message(filename,message)
char *filename;
char *message;
{
FILE *logfile;
        logfile=fopen(filename,"a");
        if(!logfile) return;
        fprintf(logfile,"%s\n",message);
        fclose(logfile);
}

void signal_handler(sig)
int sig;
{
        switch(sig) {
        case SIGHUP:
                log_message(LOG_FILE,"hangup signal catched");
                break;
        case SIGTERM:
                log_message(LOG_FILE,"terminate signal catched");
                exit(0);
                break;
        }
}

void daemonize()
{
int i,lfp;
char str[10];
        if(getppid()==1) return; /* already a daemon */
        i=fork();
        if (i<0) exit(1); /* fork error */
        if (i>0) exit(0); /* parent exits */
        /* child (daemon) continues */
        setsid(); /* obtain a new process group */
        for (i=getdtablesize();i>=0;--i) close(i); /* close all descriptors */
        i=open("/dev/null",O_RDWR); dup(i); dup(i); /* handle standart I/O */
        umask(027); /* set newly created file permissions */
        chdir(RUNNING_DIR); /* change running directory */
        lfp=open(LOCK_FILE,O_RDWR|O_CREAT,0640);
        if (lfp<0) exit(1); /* can not open */
        if (lockf(lfp,F_TLOCK,0)<0) exit(0); /* can not lock */
        /* first instance continues */
        sprintf(str,"%d\n",getpid());
        write(lfp,str,strlen(str)); /* record pid to lockfile */
        signal(SIGCHLD,SIG_IGN); /* ignore child */
        signal(SIGTSTP,SIG_IGN); /* ignore tty signals */
        signal(SIGTTOU,SIG_IGN);
        signal(SIGTTIN,SIG_IGN);
        signal(SIGHUP,signal_handler); /* catch hangup signal */
        signal(SIGTERM,signal_handler); /* catch kill signal */
}

main()
{
        daemonize();
        while(1) sleep(1); /* run */
}

/* EOF */
```