Video Lecture # 20
Designing, Coding and Managing Daemon Processes / Services

Course: SYSTEM PROGRAMMING

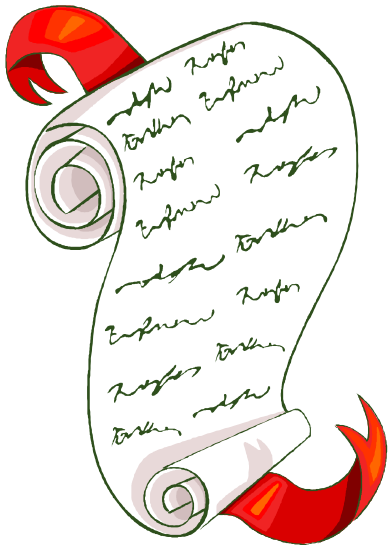
Instructor: Arif Butt

Punjab University College of Information Technology (PUCIT)
University of the Punjab

Source Code files available at: `https://bitbucket.org/arifpucit/spvl-repo/src`
Lecture Slides available at: `http://arifbutt.me`

Agenda

- Overview of daemon processes
- Writing your own daemon (programmatically)
- Introduction to system daemon (`systemd`)
- Controlling daemons using `systemctl` utility
- Steps to write your own service/daemon
  - Writing a long lived program
  - Writing a service unit file for it
  - Copying program and unit file to appropriate directories
  - Managing service using `systemctl`



Overview of
Daemons

Introduction to Daemon Processes

A daemon is a system process that provides services to system admins w/o human intervention (`cron`), provide services to OS Kernel (`paged`), or communicate with other processes (`httpd`). It has following characteristics:

- A daemon is long-lived, often 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 terminal-related signals (such as SIGINT, SIGQUIT, SIGTSTP, and SIGHUP) for a daemon

Introduction to Daemon Processes (cont...)

Daemons are written to carry out specific tasks, for example:

- **`crond`:** a daemon that executes commands at a scheduled time
- **`sshd`:** the secure shell daemon, which permits logins from remote hosts using a secure communications protocol
- **`httpd`:** the HTTP server daemon (Apache), which serves web pages
- **`xinetd`:** the Internet super server daemon, which listens for incoming network connections on specified TCP/IP ports and launches appropriate server programs to handle these connections

Writing a Daemon process

Creating a Daemon: Step - I

Perform a **`fork()`**, after which the parent exits and the child continues. The child process inherits the PGID of the parent ensuring that the child is not a process group leader. Moreover, the `daemon` process becomes the child of `/lib/systemd/systemd` process having PID of 992, which is further the child of `/sbin/init` process having a PID of 1

```
pid_t cpid = fork();
if(cpid > 0)
   exit(0);
while(1);
```

Creating a Daemon: Step - II

Close all open file descriptors that the daemon may have inherited from its parent. (A daemon may need to keep certain inherited file descriptors open, so this step is open to variation)

```
. . .
struct rlimit r;
getrlimit(RLIMIT_NOFILE, &r);
for(i=3; i<r.rlim_max; i++)
      close(i);
while(1);
```

Creating a Daemon: Step - III

Only a single instance of a daemon process should run. For example, if multiple instances of `cron` start running, each would run a scheduled operation. We can use following logic to ensure that if one instance of a program is running, no user should be able to run another instance

```
int fd = open("f1.txt", O_CREAT | O_TRUNC | O_RDWR, 0666);
struct flock lock;
lock.l_start = 0;
lock.l_len = 0;
lock.l_type = F_WRLCK;
lock.l_whence = SEEK_SET;
int rv = fcntl(fd, F_SETLK, &lock);
if(rv == -1){
   printf("This process is already running\n");
   close(fd);
   exit(1);
}
while(1);
```

Creating a Daemon: Step - IV

Make the file descriptors 0, 1, and 2 of PPFDT point to the file `/dev/null`. This is done to ensures that if the daemon calls library functions that perform I/O on these descriptors, those functions won't unexpectedly fail

```
. . .
//assume that all the descriptor are closed
int fd = open("/dev/null", O_RDWR);
dup2(fd, 1);
dup2(fd, 2);
while(1);
```

Creating a Daemon: Step - V

Since daemons run in the background, there is no need to have any terminal attached to them. To detach the terminal from a daemon we use **`setsid()`** system call, which also makes the daemon process the process group leader and session leader

```
. . .
setsid();
while(1);
```

Creating a Daemon: Step - VI

Set the file mode creation mask to 0 by calling `umask(0)`, to ensure that, when the daemon creates files and directories, they have exactly the same access privileges as mentioned in the mode specified in an `open()` or `creat()` system call

Change the process's current working directory, typically to the root directory (/). This is necessary because a daemon usually runs until system shutdown; if the daemon's current working directory is on a file system other than the one containing /, then that file system can't be unmounted

```
. . .
setsid();
umask(0);
chdir("/");
while(1);
```

Creating a Daemon: Step - VII

Handle the SIGHUP signal, so that when this signal arrives, the daemon should ignore it

```
. . .
signal(SIGHUP,SIG_IGN);
while(1);
```

Writing a Daemon Process
Proof of concept
mydaemon.c

Managing Services using `systemd`

Introduction to systemd

- The `systemd` is a system and service manager for Linux operating systems. When run as first process on boot (as PID 1), it acts as `init` system that brings up and maintains user space services
- The `/sbin/init` and `/bin/systemd` are both a soft link to `/lib/systemd/systemd`
- The system daemon has replaced the old traditional SYS-V `init` daemon as well as the short lived `upstart` daemon since Ubuntu15.04, RHEL7, CentOS7, Debian7 onwards
- The two key advantages of `systemd` over SYS-V `init` daemon are:
  - Even Driven: Stuff is started at the moment when it is needed, not before and not after that
  - Fast Booting: `systemd` displays the login prompt within a couple of seconds, no matter if a service on a remote socket is not available

Introduction to systemctl

The `systemctl(1)` is a program that is used to introspect and control the state of the `systemd` system and service manager. It uses unit files to describe services that should run along with other elements of the system configuration. A `systemd` unit file is basically a configuration file that describes how to manage a specific resource, which can be a

- **Service** units are daemons that can be started, stopped, restarted, reloaded, enabled, disabled
- **Target** units group units together and represent state of the system at anyone time, similar to run levels in older systems
- **Socket** units describes a NW, IPC socket, or a FIFO buffer. These socket unit files will always have an associated service unit file that will be started when some activity is seen on the socket that this unit defines
- **Mount** units defines a mount point on the system to be managed by systemd
- **Automount** units configure a mount point that will be automatially mounted
- **Timer** unit defines a timer managed by systemd like cornjob. A matching unit is activated when timer is triggered

Managing a Service using `systemd`

Step 1: Write a long running program in your favorite programming language that you want `systemd` to manage

Step 2: Write a service unit file to manage the above executable

Step 3: Copy the executable in `/usr/local/bin/` directory and the service unit file in the `/etc/systemd/system/` directory

Step 4: Start/stop the service using `systemctl` and enjoy

Step:1

Writing an echo server in Python

`myechoserver.py`

Anatomy of a Unit File

All unit files are organized with sections, and a section is denoted by a pair of square brackets with the section name enclosed within:

- **[Unit]:** This is the first section found in most of the files. Although section order does not matter, but mostly this is placed at the top. It defines metadata of the unit file and is used to configure the relationship of this unit with other units
- **[Service]:** This is applicable to service unit files only. You can find socket, mount, automount sections instead of service section in other unit files
- **[Install]:** This is often the last optional section and is used to define it this unit is enabled or disabled. Only units that can be enabled will have this section

To get help about service unit file:

```
$ man systemd.service
```

To display the contents of a unit file:

```
$ systemctl cat ssh.service
```

Step:2
Writing a service unit file
`myecho.service`

Step:3

Copy the program and service unit file to appropriate directories

Step:4
Use the service using `systemctl`

Things To Do

If you have problems visit me in counseling hours. . . .