Daemons

- Understanding Daemons
- Creating a Daemon
- Communicating with a Daemon

Introduction

What is a Daemon?

A *Daemon* is a background process that runs without user input and usually provides some service, either for the system as a whole or for user programs.

Common daemon programs include:

- **System Logger (syslogd);**
- **Web Servers (httpd);**
- **Mail Servers (smtpd);**
- **Database Servers (mysqld).**
- **FTP Servers (ftpd)**

Introduction

Understanding Daemons

- Normally, daemons are started when a system boots and, unless forcibly terminated, run until system shutdown.
- Because a daemon does not have a controlling terminal, any output, either to stderr or stdout, requires special handling.
- Daemons often run with superuser privilege because they use privileges ports (1 - 1024) or because they have access to some sort of privileged resource.
- Daemons generally are process group leaders and session leaders.
- Finally, a daemon's parent is the init process, which has the PID of 1. (daemon is an orphan process inherited by init)

Creating a Daemon

Although a daemon may seems mysterious and difficult to program, it is really very simple if you keep a few rules in mind and know the key function calls to make.

Simple steps to follow to create a daemon:

1. fork and exit in the parent process;
2. Create a new session in the child using the setsid call;
3. Make the root directory, /, the child process's working directory;
4. Change the child process's umask to 0;
5. Close any unneeded file descriptor the child inherited.

Creating a Daemon

1 . Fork and exit in the parent process

A daemon is started from a shell script or the command line.

Daemons are unlike application programs because they are not interactive, i.e., they run in the background and, as a result, do not have the controlling terminal.

The parent forks and exits as the first step toward getting rid of the controlling terminal (they only need a terminal interface long enough to get started).

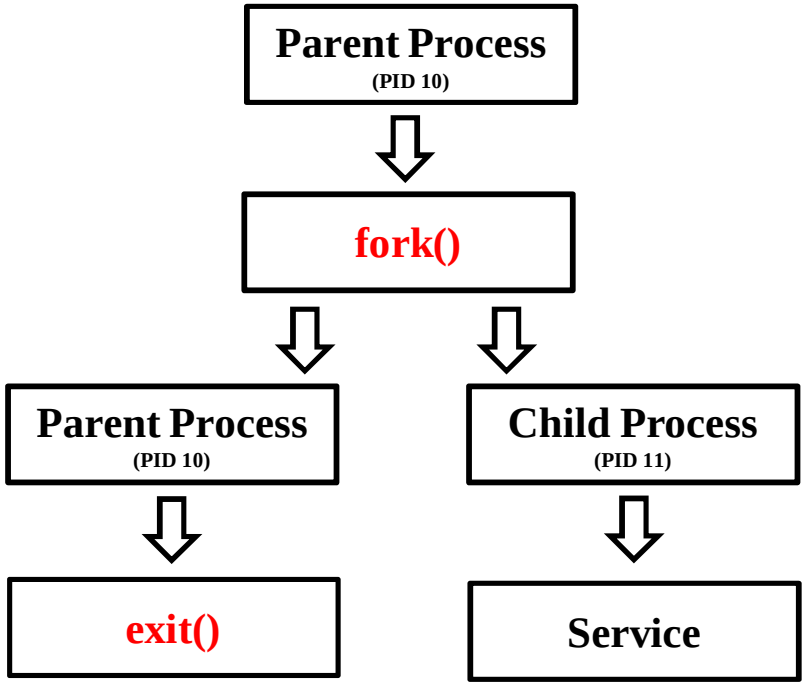
```
pid_t pid;

// create a new process
pid = fork();
if (pid < 0) { // error trying to execute fork
   ERROR("fork failure");
   exit(EXIT_FAILURE);
}

if (pid > 0) // parent process (exit)
exit(EXIT_SUCCESS);
// child process continues the execution
...
```

Creating a Daemon

1 . Fork and exit in the parent process



Creating a Daemon

2 . Create a new session in the child

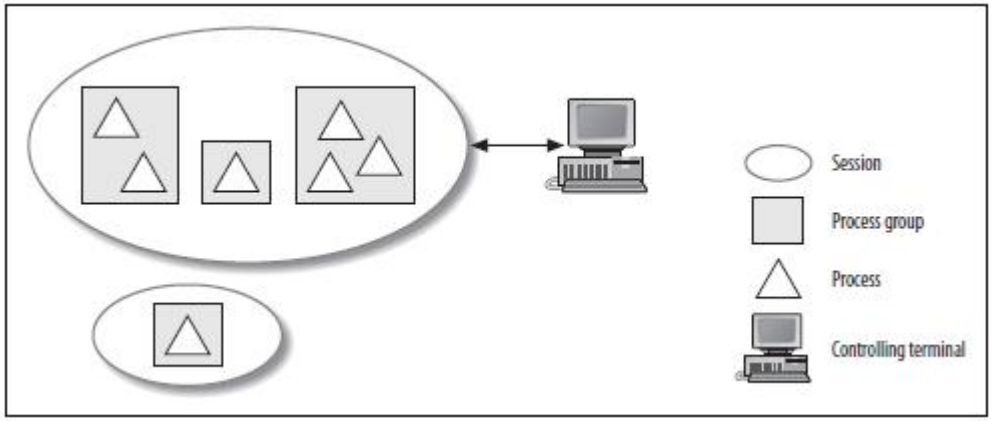
Calling setsid accomplishes several things:

- It creates a new session if the calling process is not a process group leader, making the calling process the session leader of the new session;
- It makes the calling process the process group leader of the new process group;
- It sets the process group ID (PGID) and the session ID (SID) to the process ID (PID) of the calling process;
- It dissociates the new session from any controlling tty.

```
pid_t sid;
sid = setsid(); // create a new session
if (sid < 0) {
   ERROR("setsid failure");
   exit(EXIT_FAILURE);
}
```

Creating a Daemon

2 . Create a new session in the child



- Each process is a member of a process group, which is a collection of one or more processes generally associated with each other for the purposes of job control (`# cat ship-inventory.txt | grep booty | sort`).
- When a new user first logs into a machine, the login process creates a new session that consists of a single process, the user's login shell. The login shell functions as the session leader.

Creating a Daemon

3 . Make “/” the root directory

This is necessary because any process whose current directory is on a mounted file system will prevent that file system from being unmounted.

Making “/” a daemon’s working directory is a safe way to avoid this possibility.

```
// make '/' the root directory
if (chdir(”/”) < 0) {
   ERROR(”chdir failure”);
   exit(EXIT_FAILURE);
}

// continue executing the child process

...
```

Creating a Daemon

4 . Change the child process's umask to 0

This step is necessary to prevent the daemon's inherited umask from interfering with the creation of files and directories.

Consider the following scenario:

- **A daemon inherits a umask of 055, which masks out read and execute permissions for group and other.**
- **Resetting the daemon's umask to 0 prevents such situation.**

```
// resetting umask to 0
umask(0);

// continue executing the child process

...
```

Creating a Daemon

5 . Close any unneeded file descriptor

This is simply a common sense step.

There is no reason for a child to keep open descriptors inherited from parent.

The list of potential file descriptors to close includes at least stdin, stdout and stderr.

```
// close unneeded file descriptors
close(STDIN_FILENO);
close(STDOUT_FILENO);
close(STDERR_FILENO);

// continue executing the child process

...
```

Example 1

```
...
int main(int argc, char *argv[])
{
   pid_t pid, sid;

   pid = fork(); // create a new process
   if (pid < 0) { // on error exit
      perror("fork");
      exit(EXIT_FAILURE);
   }

   if (pid > 0) // parent process (exit)
      exit(EXIT_SUCCESS);

   sid = setsid(); // create a new session
   if (sid < 0) { // on error exit
      perror("setsid");
      exit(EXIT_FAILURE);
   }

   if (chdir("/") < 0) { // on error exit
      perror("chdir");
      exit(EXIT_FAILURE);
   }

   umask(0); // make '/' the root directory
   close(STDIN_FILENO); // close standard input file descriptor
   close(STDOUT_FILENO); // close standard output file descriptor
   close(STDERR_FILENO); // close standard error file descriptor
   // service implementation
}
```

Example 1

```
int main(int argc, char *argv[])
{
   ...
   // service implementation
   len = strlen(ctime(&timebuf));
   while (1) {
      char *buf = malloc(sizeof(char) + len + 1);
      if (buf == NULL) {
         perror("malloc");
         exit(EXIT_FAILURE);
      }

      if ((fd = open("/var/log/example1.log",
                O_CREAT | O_WRONLY | O_APPEND, 0600)) < 0) {
         perror("open");
         exit(EXIT_FAILURE);
      }

      time(&timebuf);
      strncpy(buf, ctime(&timebuf), len + 1);
      write(fd, buf, len + 1);
      close(fd);
      sleep(60);
   }

   exit(EXIT_SUCCESS);
}
```

Example 1

Using example1

Compile example1:

```
# gcc example1.c -o example1
```

Run example1:

```
# sudo ./example1
```

Stop the program:

```
# kill {PID of example1}
```

Open the log file /var/log/example1.log:

```
# sudo tail -f /var/log/example1.log
```

Daemon Problem

Problem:

Once a daemon calls **setsid**, it no longer has the controlling terminal and so it has nowhere to send output that would normally go to **stdout** or **stderr** (such as error messages).

Solution:

Fortunately, the standard utility for this purpose is the **syslog** service, provided by the system logging daemon, **syslogd**.

Using syslog

Handling Errors with syslog

syslogd is a daemon that allow to save log messages from other daemons or applications.

The relevant interface is defined in <syslog.h> header file.

The API is simple, openlog opens the log, syslog writes a message to it, and closelog close the log.

The function prototypes are listed here:

```
#include <syslog.h>

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

Using syslog

void openlog(char *ident, int option, int facility);

openlog initiates a connection to the system logger.

ident is a string added to each message and typically is set to the name of the program.

The **option** argument is a logical OR of one or more of the default values listed here:

- **LOG_CONS** - write to the console if the system logger is unavailable.
- **LOG_NDELAY** - open the connection immediately. Normally, the connection is not opened until the first message is sent.
- **LOG_PERROR** - print to stderr.
- **LOG_PID** - include the process's PID with each message.

Using syslog

void openlog(char *ident, int option, int facility);

facility specifies the type of program sending the message and can be one of the following values.

LOG_AUTHPRIV	**Security/authorization messages**
LOG_CRON	**Clock daemons; cron and at**
LOG_DAEMON	**Other system daemons**
LOG_KERN	**Kernel messages**
LOG_LOCAL[0-7]	**Reserved for local use**
LOG_LPR	**Line printer subsystem**
LOG_MAIL	**The mail subsystem**
LOG_NEWS	**Usenet news subsystem**
LOG_SYSLOG	**Messages generated by syslog**
LOG_USER	**Default**
LOG_UUCP	**UUCP**

Using syslog

void closelog(void)

closelog close the system logger connection.

The use of openlog and closelog is optional, because syslog will open the log file automatically the first time it is called, and close the log file when your application is terminated.

void syslog(int priority, chat *format, ...);

syslog write a message to system logger.

priority specifies the importance of the message.

LOG_EMERG – System is unusable
LOG_ALERT – Take action immediately
LOG_CRIT – Critical condition
LOG_ERR – Error condition
LOG_WARNING – Warning condition
LOG_NOTICE – Normal but significant condition
LOG_INFO – Informational message
LOG_DEBUG – Debugging message

Example 2

```
...
int main(int argc, char *argv[])
{
   pid_t pid, sid;

   pid = fork(); // create a new process
   if (pid < 0) { // on error exit
      syslog(LOG_ERR, "%s\n", "fork");
      exit(EXIT_FAILURE);
   }

   if (pid > 0) // parent process (exit)
      exit(EXIT_SUCCESS);

   sid = setsid(); // create a new session
   if (sid < 0) { // on error exit
      syslog(LOG_ERR, "%s\n", “setsid");
      exit(EXIT_FAILURE);
   }

   if (chdir("/") < 0) { // on error exit
      syslog(LOG_ERR, "%s\n", “chdir");
      exit(EXIT_FAILURE);
   }

   umask(0); // make '/' the root directory
   close(STDIN_FILENO); // close standard input file descriptor
   close(STDOUT_FILENO); // close standard output file descriptor
   close(STDERR_FILENO); // close standard error file descriptor
   // service implementation
}
```

Example 2

Using example2

Compile example2:

 # gcc example2.c -o example2

Run example2:

 # sudo ./example2

Stop the program:

 # kill {PID of example2}

Open the log file /var/log/example1.log:

 # sudo tail -f /var/log/example2.log

Open the syslog:

 # sudo tail –f /var/log/syslog (messages)

Communicating with a Daemon

To communicate with a daemon, you send a it signals that cause it to respond in a given way.

For example, it is typically necessary to force a daemon to reread its configuration file. The most common way to do this is to send a SIGHUP signal to the daemon.

When you execute the command “kill PID” on command line, the signal SIGINT is sent to daemon to terminate the daemon execution.

```
#include <syslog.h>

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

Signal Example

Communicating with a Daemon

To communicate with a daemon, you send a it signals that cause it to respond in a given way.

For example, it is typically necessary to force a daemon to reread its configuration file. The most common way to do this is to send a SIGHUP signal to the daemon.

When you execute the command “kill PID” on command line, the signal SIGINT is sent to daemon to terminate the daemon execution.

Example 3

```c
#include <signal.h>

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

int main(int argc, char *argv[])
{
   pid_t pid, sid;

   pid = fork(); // create a new process
   if (pid < 0) { // on error exit
      syslog(LOG_ERR, "%s\n", "fork");
      exit(EXIT_FAILURE);
   }

   ...

   signal(SIGHUP,signal_handler); /* catch hangup signal */
   signal(SIGTERM,signal_handler); /* catch kill signal */
   // service implementation
}
```

Example 3

Using example3

Compile example3:

 # gcc example3.c -o example3

Run example3:

 # sudo ./example3

Stop the program:

 # kill {PID of example3}

Send SIGHUP signal:

 # kill –HUP {PID of example3}

Open the log file /var/log/example1.log:

 # sudo tail -f /var/log/example2.log

Open the syslog:

 # sudo tail –f /var/log/syslog (messages)