Linux System Programming Part 10 - Daemons

IBA Bulgaria
2018

What are Daemons?

A **daemon** is a process that runs in the background, not connecting to any controlling terminal.

Daemons are normally started at boot time, are run as root or some other special user (such as **apache** or **postfix**), and handle system-level tasks. As a convention, the name of a daemon often ends in **d** (as in **crond** and **sshd**), but this is not required, or even universal.

A daemon has two general requirements:

- it must run as a child of **init**;
- it must not be connected to a terminal.

Our chat Daemon

- Check the arguments, exit on error
- Daemonize the program, exit on error
- Prepare the listening socket
- Do until **graceful exit** is signalled
- Accept and process a new connection
- Reset the **graceful exit** and **HUp** flags
- Clean up the resources and exit

```c
int main (int argc, char *argv[]) {
  int result;
  pid_t daemonPID;
  if (argc > 1) {
    // Process the arguments...
  if((result = BecomeDaemonProcess()) < 0) {
    // Daemonization error...
  if((result = ConfigureSignalHandlers())<0) {
    // Signal handlers error...
  if((result = BindPassiveSocket(gListenPort,
                                 &gMasterSocket))<0) {
    // Socket binding error...
  do {
    if(AcceptConnections(gMasterSocket)<0) {
    // Connection accept error...
    if((gGracefulShutdown==1) && (gCaughtHupSignal==0))
      break;
    gGracefulShutdown = gCaughtHupSignal = 0;
  } while(1);
  TidyUp();
  return 0;
}
```

Daemonize

- Fork the process, exit the parent and on error
- Open the application log
- Create the **lock file** and lock it, exit on error
- Write the **pid** of the process in **lock file**
- Take the lead in a new session, exit on error
- Change the process directory to '*/*'
- Close all possible file descriptors, except for the **lock file** one
- Open the standard streams (0 - 2), redirect to '/dev/null' and exit with success

```
int BecomeDaemonProcess (void)    {
  ...
  switch (iCurrentPID = fork ())    {
      case 0: break; // The child continues execution
      case -1: // Error forking...
      default: exit (0);
  ...
  openlog (cApplicationName, LOG_NOWAIT, LOG_LOCAL0);
  if ((gLockFileDesc = creat (cLockFilePath, 0644)) < 0)
  // Error creating the lock file ...
  if ((LockResult = fcntl (gLockFileDesc, F_SETLK,
                                  &Lock)) < 0)
  // Error locking the lock file...
  if (write (gLockFileDesc, cPIDString,
                 strlen(cPIDString)) <= 0)
  // Error writing in lock file...
  if (setsid () < 0) {
  // Error changing the session ID...
  if (chdir ("/") < 0){
  // Error changing the process folder...
  long numFiles = sysconf(_SC_OPEN_MAX);
  for(i = numFiles-1; i >= 0; --i) {
      if(i != gLockFileDesc)close(i); }
  int stdioFD = open("/dev/null", O_RDWR);
  dup(stdioFD); dup(stdioFD);
  return EXIT_SUCCESS;
```

Singlas handling

Define signals **handler**:

On **SIGUSR1** - set the **graceful** flag

On **SIGHUP** - set **graceful** and **HUp** flags

On **SIGTERM** - cleanup and exit process

Block **SIGUSR2**, **SIGALRM**... flags (total 7)

Set the handler for **SIGUSR1**, **SIGHUP**, **SIGTERM** and other fatal signals (total 16)

```
void Handler (int Signal) {
    switch (Signal)    {
        case SIGUSR1: gGracefulShutdown = 1;break;
        case SIGHUP: gGracefulShutdown =
                    gCaughtHupSignal = 1;break;
        case SIGTERM:TidyUp();exit (EXIT_SUCCESS); ...
        default: ...
            TidyUp(); exit (0);
...
int ConfigureSignalHandlers (void)    {
    int i, j, k;
    int BlockSignals[] = {SIGUSR2, SIGALRM, SIGPIPE, ...};
    int HandleSignals[] = {SIGUSR1, SIGHUP, SIGTERM, ...
    sigemptyset (&SigMask);
    for (i = 0; i < 7; i++)    {
        sigaddset (&SigMask, *(BlockSignals + i));  }
    ...
    for (i = 0; i < 16; i++)    {
        sigaction (*(HandleSignals + i), &SignalAction, NULL);
    }
    return EXIT_SUCCESS;
}
```

Set listening Socket

Initialize socket configuration for INET family, provided port and all interfaces

Create the socket, exit on error

Set the socket to reuse the address

Bind the socket, exit on error

Listen on the socket, exit on error

Exit with success

```c
int BindPassiveSocket(const int portNum,
                      int *const boundSocket)
{
    struct sockaddr_in sin;
    int   newsock, optval;
    size_t optlen;
    memset(&sin.sin_zero, 0, 8);
    sin.sin_port = htons(portNum);
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = htonl(INADDR_ANY);
    if((newsock= socket(PF_INET, SOCK_STREAM, 0))<0)
      return -1;
    optval = 1;
    optlen = sizeof(int);
    setsockopt(newsock, SOL_SOCKET, SO_REUSEADDR,
               &optval, optlen);
    if(bind(newsock, (struct sockaddr*) &sin,
        sizeof(struct sockaddr_in))<0) return -1;

    if(listen(newsock,SOMAXCONN)<0) return -1;
    *boundSocket = newsock;
    return 0;
}
```

Connections handling

- Define connections **handler**:
- Read text line from the passed socket
- If no error - write the line back to socket
- Return the result of the read/write

- While connections accepting is ok:
- Accept new connection, on **EINTR** try again, on other error exit the loop
- Handle the accepted connection
- Close the accepted socket

```
int HandleConnection(const int slave) {
  ...
  retval = ReadLine(slave, readbuf, buflen, &bytesRead);
  if(retval == 0)
      WriteToSocket(slave, readbuf, bytesRead);
  return retval;
}
int AcceptConnections(const int master){
  int    proceed = 1, slave, retval = 0;
  struct sockaddr_in client;
  socklen_t   clilen;
  while((proceed==1) && (gGracefulShutdown==0)) {
    clilen = sizeof(client);
    slave = accept(master,(struct sockaddr *)&client,
                   &clilen);
    if(slave < 0) {
        if(errno == EINTR) continue;
        proceed = 0; retval = -1;
    } else {
        retval = HandleConnection(slave);
        if (retval) proceed = 0;
    }
    close(slave);
  }
  return retval;
```

Tidy Up

If we have open lock file descriptor:

Close the file descriptor

Delete the lock file

If we have open listening socket:

Close the listening socket

```c
int gLockFileDesc=-1;
int gMasterSocket=-1;
const char *const gLockFilePath =
                    "/var/run/chatdaemon.pid";

void TidyUp(void)
{
    if(gLockFileDesc!=-1) {
            close(gLockFileDesc);
            unlink(gLockFilePath);
            gLockFileDesc=-1;
    }
    if(gMasterSocket!=-1) {
            close(gMasterSocket);
            gMasterSocket=-1;
    }
}
```

Startup arguments

If arguments are passed, this first one should be ‘**stop**’ or ‘**restart**’ command

Open **lock file**, exit on error

Read and convert the **pid** from the **lock file**

If the command is ‘**stop**’, send **SIGUSR1** to the **pid** process and exit with success

If the command is ‘**restart**’, send **SIGHUP** to the **pid** process and exit with success

If the command is not correct, print the usage and exit with error

```c
int main (int argc, char *argv[]) {
  ...
  if (argc > 1) {
    int fd, len;
    pid_t pid;
    char pid_buf[16];
    if ((fd = open(gLockFilePath, O_RDONLY)) < 0) {
      //ERROR ...
    }
    len = read(fd, pid_buf, 16);
    pid_buf[len] = 0;
    pid = atoi(pid_buf);
    if(!strcmp(argv[1], "stop")) {
          kill(pid, SIGUSR1);
          exit(EXIT_SUCCESS);
    }
    if(!strcmp(argv[1], "restart")) {
          kill(pid, SIGHUP);
          exit(EXIT_SUCCESS);
    }
    printf ("usage %s [stop|restart]\n", argv[0]);
    exit (EXIT_FAILURE);
  }
```