CSC 4304 - Systems Programming
Fall 2008

LECTURE - XVII

DAEMON PROCESSES & ADVANCED I/O

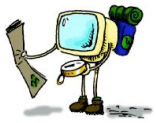
Tevfik Koşar

Louisiana State University

November 6th, 2008

Roadmap

- Simple Web Server
- Daemon Processes
- Advanced I/O



Simple Web Server

Logic of a Web Server

- Setup the server
  - socket, bind, listen
- Accept a connection
  - accept, fdopen
- Read a request
  - fread
- Handle the request
  - 1. directory --> list it: opendir, readdir
  - 2. regular file --> cat the file: open, read
  - 3. .cgi file --> run it: exec
  - 4. not exist --> error message
- Send a reply
  - fwrite

An HTTP Request

- <command> <argument> <HTTP version>
- <optional arguments>
- <blank line>

- GET /index.html HTTP/1.0

Server Response

- <HTTP version> <status code> <status message>
- <aditional information>
- <a blank line>
- <content>

- HTTP/1.1 200 OK
  Date: Thu, 06 Nov 2008 18:27:13 GMT
  Server: Apache

  <HTML><HEAD><BODY> ....

Example

```
$ telnet www.cnn.com 80
Trying 64.236.90.21...
Connected to www.cnn.com.
Escape character is '^]'.
GET /index.html HTTP/1.0

HTTP/1.1 200 OK
Date: Thu, 06 Nov 2008 18:27:13 GMT
Server: Apache
Accept-Ranges: bytes
Cache-Control: max-age=60, private
Expires: Thu, 06 Nov 2008 18:28:14 GMT
Content-Type: text/html
Vary: Accept-Encoding,User-Agent
Connection: close

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN""http://
www.w3.org/TR/html4/loose.dtd"><html lang="en"><head><title>CNN.com -7
```

Writing a Simple Web Server

```
int init_socket(int portnum)
{   ...
    gethostname( hostname , 256 );          /* where am I ?         */
    hp = gethostbyname( hostname );         /* get info about host  */
    ...
    bzero( (void *)&saddr, sizeof(saddr) ); /* zero struct & fill host addr*/
    bcopy( (void *)hp->h_addr, (void *)&saddr.sin_addr, hp->h_length);
    saddr.sin_family = AF_INET ;            /* fill in socket type  */
    saddr.sin_port = htons(portnum);        /* fill in socket port  */

    sock_id = socket( AF_INET, SOCK_STREAM, 0 );    /* get a socket */
    ...
    rv = setsockopt(sock_id, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
    ...
    bind(sock_id, (struct sockaddr *) &saddr, sizeof(saddr));
    ...

    listen(sock_id, 1) != 0 );
    ...
    return sock_id;
 }
```



```
int main(int ac, char *av[])
{
  ...
  sock = init_socket(portnum);
  ...
  /* main loop here */
  while(1){
       /* take a call and buffer it */
       fd    = accept( sock, NULL, NULL );
        ...
       fpin  = fdopen(fd, "r" );
       fpout = fdopen(fd, "w" );

       /* read request */
       fgets(request,BUFSIZ,fpin);
       ...
        while( fgets(buf,BUFSIZ,fp) != NULL && strcmp(buf,"\r\n") != 0 ) ;

      /* do what client asks */
       process_rq(request, fpout);
        ...
       fclose(fpin);
       fclose(fpout);
  }
  return 0;
  /* never end */
 }
```



```
void process_rq( char *rq, FILE *fp)
{
    ...

    /* create a new process and return if not the child */
    if ( fork() != 0 ) return;

    if ( sscanf(rq, "%s%s", cmd, arg) != 2 ) return;

    ...

    if ( strcmp(cmd,"GET") == 0 )
    {
     if ( not_exist( item ) )
         do_404(item, fp );
     else if ( isadir( item ) )
         do_ls( item, fp );
     else
         do_cat( item, fp );
    }
    ...
    exit(0);
}
```



```
void do_cat(char *f, FILE *fpsock)
{
    char *extension = file_type(f);
    char *content = "text/plain";
    FILE *fpfile;
    int c;

    if ( strcmp(extension,"html") == 0 )
        content = "text/html";
    else if ( strcmp(extension, "gif") == 0 )
        content = "image/gif";
    else if ( strcmp(extension, "jpeg") == 0 )
        content = "image/jpeg";

    fpfile = fopen( f , "r");
    if ( fpfile != NULL )
    {

         fprintf(fpsock, "HTTP/1.0 200 OK\r\n");
        fprintf(fsock, "Content-type: %s\r\n", content );
        fprintf(fpsock, "\r\n");
        while( (c = getc(fpfile) ) != EOF )
            putc(c, fpsock);
        fclose(fpfile);
    }
}
```


Daemon Processes

Daemon Characteristics

Commonly, dæmon processes are created to offer a specific service.

Dæmon processes usually

- live for a long time
- are started at boot time
- terminate only during shutdown
- have no controlling terminal

The previously listed characteristics have certain implications:

- do one thing, and one thing only
- no (or only limited) user-interaction possible
- consider current working directory
- how to create (debugging) output

Writing a Daemon

- fork off the parent process
- change file mode mask (umask)
- create a unique Session ID (SID)
- change the current working directory to a safe place
- close (or redirect) standard file descriptors
- open any logs for writing
- enter actual dæmon code

Example Daemon Creation

```
int daemon_init(void)
{
  pid_t pid;
  if ((pid=fork())<0)  return (-1);
  else if (pid!=0) exit (0); //parent goes away
  setsid(); //becomes session leader
  chdir("/"); //cwd
  umask(0);   //clear file creation mask
return (0)
}
```


Daemon Logging

A daemon cannot simply print error messages to the terminal or standard error. Also, we would not want each daemon writing their error messages into separate files in different formats. A central logging facility is needed.

There are three ways to generate log messages:

- via the kernel routine `log(9)`
- via the userland routine `syslog(3)`
- via UDP messages to port 514

Syslog()

`openlog(3)` allows us to set specific options when logging:

- prepend *ident* to each message
- specify logging options (`LOG_CONS | LOG_NDELAY | LOG_PERRO | LOG_PID`)
- specify a *facility* (such as `LOG_DAEMON`, `LOG_MAIL` etc.)

`syslog(3)` writes a message to the system message logger, tagged with *priority*. A *priority* is a combination of a *facility* (as above) and a *level* (such as `LOG_DEBUG`, `LOG_WARNING` or `LOG_EMERG`).

Example

- openlog("myhttpd", LOG_PID, LOG_MAIL);

Advanced I/O

Non-blocking I/O

Certain system calls can block forever:

- `read(2)` from a particular file, if data isn't present (pipes, terminals, network devices)
- `write(2)` to the same kind of file
- `open(2)` of a particular file until a specific condition occurs
- `read(2)` and `write(2)` of files that have mandatory locking enabled
- certain `ioctls(2)`
- some IPC functions (such as `sendto(2)` or `recv(2)`)

Nonblocking I/O lets us issue an I/O operation and not have it block forever. If the operation cannot be completed, return is made immediately with an error noting that the operating would have blocked.

Example

```
    int main(void)
{
    int     ntowrite, nwrite;
    char *ptr;

    ntowrite = read(STDIN_FILENO, buf, sizeof(buf));
    fprintf(stderr, "read %d bytes\n", ntowrite);

    set_fl(STDOUT_FILENO, O_NONBLOCK); /* set nonblocking */

    for (ptr = buf; ntowrite > 0; ) {
        errno = 0;
        nwrite = write(STDOUT_FILENO, ptr, ntowrite);
        fprintf(stderr, "nwrite = %d, errno = %d\n", nwrite, errno);
        if (nwrite > 0) {
            ptr += nwrite;
            ntowrite -= nwrite;
        }
    }

    clr_fl(STDOUT_FILENO, O_NONBLOCK); /* clear nonblocking */
    exit(0);
}
```


Acknowledgments

- Advanced Programming in the Unix Environment by R. Stevens
- The C Programming Language by B. Kernighan and D. Ritchie
- Understanding Unix/Linux Programming by B. Molay
- Lecture notes from B. Molay (Harvard), T. Kuo (UT-Austin), G. Pierre (Vrije), M. Matthews (SC), B. Knicki (WPI), M. Shacklette (UChicago), J.Kim (KAIST), and J. Schaumann (SIT).