Practical aspects of server design

Stefan D. Bruda

CS 464/564, Fall 2023

Writing actual code for actual servers

- A (Unix) server is different from a normal program
  - In particular, a server does not interact with a user
  - It communicates instead with other programs over a network
  - It also spawns threads/processes (which are not under immediate user control)
- One is faced thus with a bunch of new issues, including
  - Preventing users to affect server's execution in other ways than the ones specified
  - Providing a mechanism for the server to report status and errors
  - Resource management
  - Access control and other security issues

Daemons everywhere

- A normal program runs in foreground
  - It is attached to a terminal (more general, a "tty")
  - It receives user input from that terminal
  - It prints output (using `cout<<`, `printf`, ...) and error messages (using `cerr<<`, `perror`, ...) to the same terminal
- A server is a daemon i.e., it runs in background
  - A production server is not attached to any terminal
  - Instead, it is launched upon boot, maybe even before terminals are born
  - Thus, it does not accept user input
  - It must send the output to something else than a terminal too

Programming a server as a daemon

- The easy way: you put the server in background explicitly

```
shfd -d -v &
```

- The hard way: the server puts itself into the background
  - You start with a process that does the server initialization
  - It prints whatever messages it wants (to the terminal or something)
  - It then goes in the background for the rest of the job

```
int main (...) {
  Initialize server (socket binding, preparation of the file system)
  int bgpid = fork();
  if (bgpid < 0) {
    perror("startup fork");
    return 1;   }
  if (bgpid)    // parent dies
    return 0;
  Child continues and becomes the server
}
```

Background the hard way (cont'd)

OK, but why?

- A server is usually started up by the init script
- This script starts the servers in a specific order
  - E.g., the database server should be started before the Web server (which needs it)
- The init script cannot put everything into the background from the very start
  - It has to make sure that the server actually started before moving forward
- On the other hand, if the server never gets into background, the init script never gets a chance to go ahead and start the other services
- Ergo, a server that expects to be launched by the init script (they all should!)
  - Sits in foreground until it makes sure that the startup succeeded
  - Goes then into background for the actual work

Debugging a daemon

- A server will eventually need debugging, like any other program
- When this happens, it is much more convenient to run the server in foreground
  - So that we can see the output and maybe stop it by typing `Control-c`
- So it is convenient to have a command line switch that will keep the server in foreground:

```
int main (...) {
  Initialize (socket binding, preparation of the file system)
  if ( strcmp(argv[1],"-d") == 0 ) {
    argc--; argv++;
    int bgpid = fork();
    if (bgpid < 0) {
      perror("startup fork");
      return 1;     }
    if (bgpid)
      return 0;
  }
   Child (or parent) continues with the server code
}
```

- Normal operation: `shfd -f 10000`
- Debug: `shfd -d -f 10000`

Debugging and other verbose output

- Debugging programs is generally difficult
- Debugging servers is even more so (they are concurrent, grumpy, etc.)
- Typical debugging involves verbose logging
- In the process the server usually stays attached so that we can stop (and restart it) as needed
- While attached, it is probably a good idea not to redirect the standard output and standard error streams, as it is often more convenient to have the whole output in the terminal
- These variations in behaviour are best accomplished via command-line switches

Command-line switches for verbose output

- Normal way to obtain the command line arguments:

```
#include <stdio.h>
#include <unistd.h>

int main (int argc, char** argv) {

  for (int i = 1; i < argc; i++) {
    printf("argv[%d] = %s\n", i, argv[i]);  }    }
```

Command-line switches for verbose output

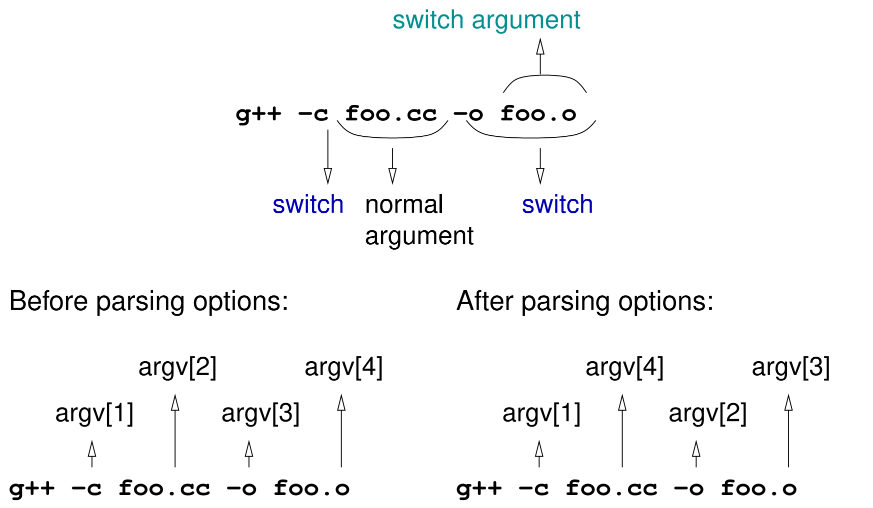
- Obtain command line arguments by identifying switches:

```
#include <stdio.h>
#include <unistd.h>

extern char *optarg;
extern int optind;

int main (int argc, char** argv) {
  int c;
  printf("--------- options: ---------\n");
  while ((c = getopt (argc,argv,"abcd:")) != -1) {
    printf("opt: %c arg %s\n", (char)c, optarg);
  }
  argc -= optind - 1; argv += optind - 1;
  printf("------ remaining args: ------\n");
  for (int i = 1; i < argc; i++) {
    printf("argv[%d] = %s\n", i, argv[i]);  }    }
```


Command-line switches for verbose output



Typical use of switches

- `-d` or `-D` usually stand for "debug"
  - This might make the daemon more verbose but it almost always prevents the daemon from detaching
  - Typically output is produced to standard output (as opposed to log facilities), but this is not always the case (probable cause: laziness)
- `-v` usually stands for "verbose output"
  - It increases the verbosity of the program but does not necessarily keep the program attached and does not necessarily change the destination of program's output
  - Often different levels of verbosity are needed; this is accomplished typically by providing multiple `-v` switches in the command line (the more occurrences of `-v` the more verbose the program)
- As an alternative to the command line debugging behaviour can be changed via configuration options in a configuration file
  - Often both methods are supported

Talking to daemons

- We have first to find the process id of the server process
  - We do `ps aux`, we get a lot of lines like this
    ```
    USER   PID   %CPU %MEM   VSZ  RSS TTY    STAT START TIME COMMAND
    ...
    bruda  13319  0.0  0.1  2572  816 pts/1  S    12:15 0:00 shfd -d -D
    ...
    ```
    and then we hunt for our server between them
  - We do `ps aux | grep` name, we get only the lines that contain name
  - We already have the pid (useful!) — how?
- We could then send a signal to the server

`kill` pid	sends `SIGQUIT` to pid	(which may terminate)
`kill -KILL` pid	sends `SIGKILL` to pid	(which will terminate)
`kill -HUP` pid	sends `SIGHUP` to pid	(which restarts if civilized)

Lonely daemons

- Servers are lonely. It does not make sense to run multiple copies of a server on the same machine
  - How do we prevent multiple copies to run?
- Each server has a well-known associated lock file
  - Different servers use different lock files, but a server will always use the same lock file
- Immediately upon startup the server tries to acquire a lock on this file
  - If it succeeds, it goes ahead with the rest
  - If it fails, it terminates (there is another copy running)
    - An error message would be nice too...
  - When the server exits, it releases the lock on the file and deletes the file
  - Loosely speaking, each server runs in a critical region
- The lock file is also a good place to hold the process id of the server

Grumpy daemons

- Except for the signals they like, daemons do not want to talk to you
- If you leave them in the sate typical for a normal program, they might even get angry and refuse to do the work
  - This happens when they try for some reason to access standard input (descriptor 0)
  - So we have to close descriptor 0
  - What the heck, we close all the descriptors except standard output and standard error!

```
for (int i = 0; i < getdtablesize(); i++)
  if (i != 1 && i != 2)
    close(i);
```

  - Closing descriptors is very important, we thus prevent the server from consuming resources unnecessarily but most importantly we have control over the descriptors (a matter of security)
  - But note that we close them before opening back those descriptors we actually need (so that we positively know what are the files on which the server operates)
- Closing descriptor 0 does not make our server happy though (why?)

Grumpy daemons (cont'd)

- The server may still try to access descriptor 0
  - Many library functions assume that the first three descriptors are open
  - We just exchange one error for another!
- So we open descriptor 0 again
  - This time, descriptor 0 will point to a special device which does nothing ("bit bucket")
  - This device is called, suggestively, `/dev/null`
  - Reading from `/dev/null` always return an end of file
  - Anything written to `/dev/null` is discarded

```
for (int i = 0; i < getdtablesize(); i++)
  if (i != 1 && i != 2)
    close(i);
// We closed descriptor 0 already, so this
// will be the first one available
int fd = open("/dev/null", O_RDWR);
```


Detached daemons

- Each Unix process inherits a connection to its controlling tty
  - A user that started a process can control it by issuing appropriate control commands to that process' controlling tty
- Unlike normal programs, servers should not receive signals generated by the process that started it
  - Signaling from the tty to the piece of code that starts the server is acceptable (sometimes desired), signaling to the server itself is not
- A server must therefore detach itself from the controlling tty

```
#include <sys/ioctl.h>

int fd = open("/dev/tty",O_RDWR);
ioctl(fd,TIOCNOTTY,0);
close(fd);
```

Detached daemons and their output

- OK, so we have now no terminal, where do we put the output?
- We redirect standard output (descriptor 1) and standard error (descriptor 2)
- Using the command line:
  - Redirecting both to the same file:

```
shfd -d  >& global-output-file
shfd -d >>& global-output-file
```

  - Redirecting to different files (bash-like shells):

```
shfd -d 1> output-file  2> error-file
shfd -d 1>> output-file 2>> error-file
```


Detached daemons and their output (cont'd)

- Command line syntax varies
- Not a good idea security-wise to rely on descriptors opened by somebody else
- How about the initializing code? It should print to the terminal
- So we redirect output from inside the program

```
// We close everything!!
for (int i = getdtablesize() - 1; i >= 0 ; i--)
  close(i);
// We closed descriptor 0 already, so this
// will be the first one available
int fd = open("/dev/null", O_RDWR);
// We now re-open descriptors 1 and 2, in this order:
```

Same file:

```
fd = open("global-output-file", O_WRONLY|O_CREAT|O_APPEND);
dup(fd);
```

Different files:

```
fd = open("output-file", O_WRONLY|O_CREAT|O_APPEND);
fd = open("error-file", O_WRONLY|O_CREAT|O_APPEND);
```


Daemons don't like signals

- There is no signal from the controlling tty, but nonetheless a server may receive signals (e.g., from you when you use the command `kill`)
- Some signals (e.g., `SIGHUP`, maybe) have some meaning to the server
  - One signal always has some meaning to any Unix program namely, `SIGKILL`
- Signals with meanings should have associated signal handlers (except `SIGKILL`)

  `signal(signal,handler-function);`
- Some other signals do not have any meaning
  - Signals that are not needed should be ignored
  - There is a predefined function that does exactly this: `SIG_IGN`

  `signal(signal,SIG_IGN);`

SIGPIPE

- Notable signal
- Sent to the server when a client closes the connection
- When unhandled a `SIGPIPE` brings down the whole process
- A server must not die when a client leaves
- Therefore this signal should always be explicitly handled
- Ignoring it is fine for most applications, since the socket also receives an end of file

Daemons are not gregarious

- Unix places each process in a process group
- It can then treat a set of related processes as one entity
- A server inherits membership in a process group
- But usually a server operates independently from any process group
  - E.g., it should not receive signals sent to its parent's group
  - The server must thus leave its parent's group:
    `setpgid(what-process,to-what-group);`
  - The process id of the current process (which is passed to `setpgid`) can be obtained by using the function `getpid`
  - To create a new, private group we pass 0 as second argument of `setpgrp`. So we do:
    `setpgid(getpid(),0);`

Secure daemons

- Servers may run with root privileges
  - In other words, they can do whatever they please with your system
  - So you the programmer have to make sure they do not do things that interfere with normal system operation
- Careful programming is one way of keeping them at bay
  - In particular, it is crucial that you check for array bounds, and that you do not access memory areas you do not own
  - Not checking for these is the most usual cause for issuing security updates (and for people cracking into your system)
  - Obviously a complex problem (to be continued)
- In addition, you should be careful about what servers write to disk and where

Daemons and their directories

- When a program is launched, it inherits an environment variable called the current working directory
- When a program creates or opens a file it looks in this current working directory
- Servers are launched by the init script, which works in a directory whose content should not be modified
- Servers have this habit to write on disk
- You can specify the directory they write into by providing absolute paths to your files
- But a server that encounters an error condition might dump core (write to disk a memory image for debugging purposes. . . in the current working directory!)
- But a server started by the system administrator will have the current directory as the home directory of the administrator
- But a server working in some directory will prevent that directory to be unmounted even if the server does not use the directory for anything
- Conclusion: You should move a server to a known, "safe" directory. Most servers do: `chdir("/run/shfd");`

Confidential daemons

- Some data that is written to files is log data, which should be readable (but not writable) by many people
- Some other data should not be accessible to anybody else (e.g., passwords)
- Each file in a Unix file system has a set of permissions to control access to files
  - You can (and should) specify at creation time the permissions of the file you create
  - You can also specify a set of permissions that will never be set (the umask)

r read permission set
w write permission set
x execute permission set
– permission not set

owner:	group:	others:
rwx	r-x	rw-
111	101	110
7	5	6

permissions for the file (declared):	756
umask (denied permissions):	037
actual permissions for the file:	740

Bitwise AND with the negated umask

Setting a umask

- You do not want to run into the possibility of creating a file owned by the administrator and with all the permissions set (777). Not even by chance!
- So, besides setting suitable permissions for each file you create, it is a very good idea to provide a suitable umask for the server as a whole
- To set a (new) umask, you use the system call `umask`
  - It is very comfortable to work with numbers in octal when you deal with file permissions
    - This way a digit corresponds with a set of permissions for a given group of users
    - In C/C++ a literal integer whose first digit is 0 is considered to be in base 8
    - So when you call `umask`, it is likely that you do not want to write

      ```
      umask(137);
      ```

      but rather

      ```
      umask(0137);
      ```
- Always keep in mind that the umask specifies permissions that are denied

Daemons and zombies (a reminder)

- If the main server exits, no problems will arise
- However, if the server process creates other processes, you may end up with zombie processes
  - So remember to always wait after your children (as we talked about earlier)
  - That is, if your server spawns new processes, it has to have a suitable handler for the `SIGCHLD` signal
- Same issue is applicable to attached threads that are not joined