## Daemons

- Daemon = A program does something useful without interacting directly with the user (sits in the background)
  - Technically not part of the kernel, but still part of the OS
  - Good example: network (TCP) servers
- A (Unix) daemon is different from a normal program
  - In particular, a server does not interact with a user
  - It communicates instead with other programs maybe 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 = Background

- 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 daemon runs in background
  - 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



## Programming a Daemon

- The easy way: put the daemon in the background explicitly

```
bbserv -c bb.conf -f bbfile &
```

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

```
int main (...) {
  Initialize stuff (network 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 daemon
}
```



## The Hard Way (cont'd)

- OK, but why?
  - A daemon is started up by the init process
  - init starts the daemons in a specific order
    - e.g., remote file system access should be started before anybody needs it
  - init cannot put everything into the background from the very start
    - it has to make sure that the daemon actually started before moving forward
  - On the other hand, if the daemon never gets to the background, init never gets a chance to go ahead and start the other services
  - Ergo, a daemon that expects to be launched by init (they all should!)
    - sits in the foreground until it makes sure that the startup succeeded
    - goes then into background for the actual work

## Talking to Daemons

- We have first to find the process id of the daemon 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 bbserv bbb
```

    and then we hunt for our daemon between them
  - We do `ps aux | grep` name, we get only the lines that contain name
  - We already have the pid (useful!)
    - But how?
- We could then send signals to the daemon

| | | |
|---|---|---|
| `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 normally restarts) |



## Lonely Daemons

- Daemons are lonely. It does not make sense to run multiple copies of a daemon on the same machine
  - How do we prevent multiple copies to run?



## Lonely Daemons

- Daemons are lonely. It does not make sense to run multiple copies of a daemon on the same machine
  - How do we prevent multiple copies to run?
- Each daemon has a well-known associated lock file
  - Different daemons use different lock files, but a daemon will always use the same lock file
- Immediately upon startup the daemon 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 daemon exits, it releases the lock on the file and deletes the file
  - Loosely speaking, each daemon runs in a critical region



## Lonely Daemons

- Daemons are lonely. It does not make sense to run multiple copies of a daemon on the same machine
  - How do we prevent multiple copies to run?
- Each daemon has a well-known associated lock file
  - Different daemons use different lock files, but a daemon will always use the same lock file
- Immediately upon startup the daemon 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 daemon exits, it releases the lock on the file and deletes the file
  - Loosely speaking, each daemon runs in a critical region
- The lock file is also a good place to hold the process id of the daemon!

## 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 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 daemon from consuming resources unnecessarily
  - But note that we close them before opening back those descriptors we actually need (such as who knows what file on which the daemon does its stuff)
- Closing descriptor 0 does not make our daemon happy though! (why?)



## Grumpy Daemons (cont'd)

- The daemon 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 you write to `/dev/null` is simply 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, daemons should not receive signals generated by the process that started it
  - Signaling from the tty to the piece of code that starts the daemon is acceptable (sometimes desired), signaling to the daemon itself is not
  - A daemon must 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?

## Detached Daemons and Their Output

- OK, so we have now no terminal, where do we put the output?
  - Initialization code outputs to whatever is inherited from the parent process
  - We then redirect standard output (descriptor 1) and standard error (descriptor 2) to files

```
// We close everything!!
for (int i = getdtablesize() - 1; i >= 0 ; i--)
  close(i);
int fd = open("/dev/null", O_RDWR); // Descriptor 0
// We now re-open descriptors 1 and 2, in this order:
```

    – Same file:

```
fd = open("global-output-file", O_RDWR);
dup(fd);
```

    – Different files:

```
fd = open("output-file", O_RDWR);
fd = open("error-file", O_RDWR);
```



## Daemons Don't Like Signals

- There is no signal from the controlling tty, but nonetheless a daemon may receive signals (e.g., from you when you use the command `kill`)
- Some signals (e.g., `SIGHUP`, maybe) have some meaning to the daemon
  - 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 a network server when a client quits unexpectedly (without shutting down the socket)
- When unhandled a `SIGPIPE` brings down the whole process
- A server must not die when a client misbehaves
- Ergo, 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 daemon inherits membership in a process group
- But: usually, a daemon operates independently from any process group
  - E.g., it should not receive signals sent to its parent's group
  - The daemon 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:

```
setpgrp(getpid(),0);
```

## Secure Daemons

- Daemons 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)
  - This is of course a complex problem
- In addition, you should be careful about what daemons write to disk and where



## Daemons and Their Directories

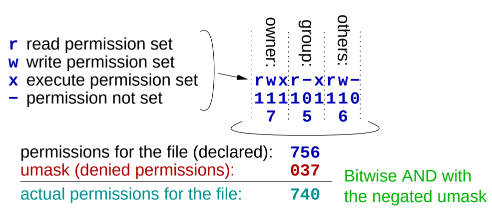
- When a program is launched, it inherits an environment variable called the current working directory
- When a program creates or opens a file, it looks in this current working directory
- Daemons are launched by the init script, which works in a directory whose content should not be modified
- Daemons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

```
chdir("/");
```



## Confidential Daemons

- Some data that is written to files is log data, which should be inspectable 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
  - You can 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)






## 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 daemon 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