CS 33

More Network Programming

The source code used in this lecture is on the course web page.

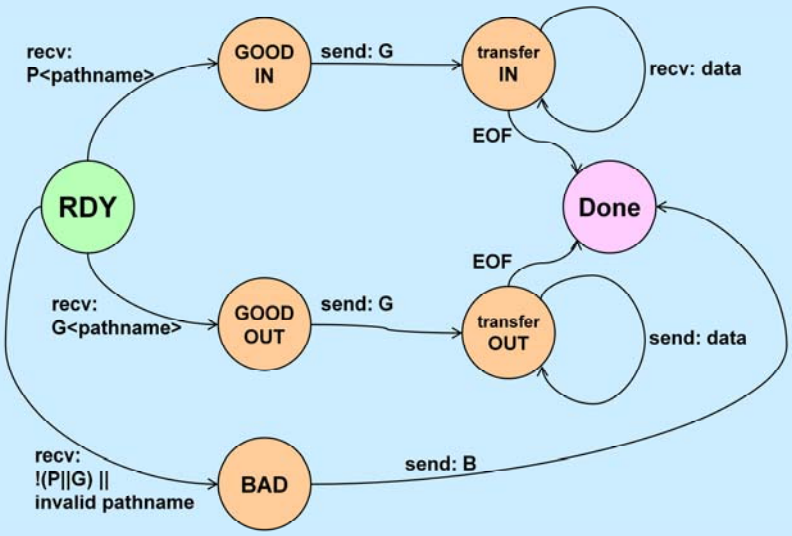
A Really Simple Protocol

- Transfer a file
  - layered on top of TCP
    - » reliable
    - » indicates if connection is closed
- To send a file

  P<null-terminated pathname><contents of file>
- To retrieve a file

  G<null-terminated pathname><contents of file>

Server State Machine



Keeping Track of State

```
typedef struct client {
  int fd;     // file descriptor of local file being transferred
  int size;   // size of out-going data in buffer
  char buf[BSIZE];
  enum state {RDY, BAD, GOOD, TRANSFER} state;
  /*
     states:
        RDY: ready to receive client's command (P or G)
        BAD: client's command was bad, sending B response + error msg
        GOOD: client's command was good, sending G response
        TRANSFER: transferring data
    */
  enum dir {IN, OUT} dir;
  /*
     IN: client has issued P command
     OUT: client has issued G command
   */
} client_t;
```

Main Server Loop

```
while(1) {
  select(maxfd, &trd, &twr, 0, 0);
  if (FD_ISSET(lsock, &trd)) {
    // a new connection
    new_client(lsock);
  }
  for (i=lsock+1; i<maxfd; i++) {
    if (FD_ISSET(i, &trd)) {
      // ready to read
      read_event(i);
    }
    if (FD_ISSET(i, &twr)) {
      // ready to write
      write_event(i);
    }
  }
  trd = rd; twr = wr;
}
```


Note that *trd*, *twr*, *rd* and *wr* are allof type *fd_set*. *rd* and *wr* are initialized so that *rd* contains just the file descriptor for the listening socket and *wr* is empty. *trd* and *twr* are copied from *rd* and *wr* respectively before the loop is entered.

New Client

```
// Accept a new connection on listening socket
// fd. Return the connected file descriptor

int new_client(int fd) {
  int cfd = accept(fd, 0, 0);
  clients[cfd].state = RDY;
  FD_SET(cfd, &rd);
  return cfd;
}
```


Read Event (1)

```
// File descriptor fd is ready to be read. Read it, then handle
// the input
void read_event(int fd) {
  client_t *c = &clients[fd];
  int ret = read(fd, c->buf, BSIZE);
  switch (c->state) {
  case RDY:
    if (c->buf[0] == 'G') {
      // GET request (to fetch a file)
      c->dir = OUT;
      if ((c->fd = open(&c->buf[1], O_RDONLY)) == -1) {
        // open failed; send negative response and error message
        c->state = BAD;
        c->buf[0] = 'B';
        strncpy(&c->buf[1], strerror(errno)+1, BSIZE-2);
        c->buf[BSIZE-1] = 0;
        c->size = strlen(c->buf)+1;
      }
```


Read Event (2)

```
    else {
      // open succeeded; send positive response
      c->state = GOOD;
      c->size = 1;
      c->buf[0] = 'G';
    }
    // prepare to send response to client
    FD_SET(fd, &wr);
    FD_CLR(fd, &rd);
    break;
  }
```

Read Event (3)

```
if (c->buf[0] == 'P') {
  // PUT request (to create a file)
  c->dir = IN;
  if ((c->fd = open(&c->buf[1],
      O_RDWR|O_CREAT|O_TRUNC, 0666)) == -1) {
      // open failed; send negative response and error message
      ...
} else {
    // open succeeded; send positive response
    ...
}
// prepare to send response to client
FD_SET(fd, &wr);
FD_CLR(fd, &rd);
break;
}
```

Read Event (4)

```
case TRANSFER:
  // should be in midst of receiving file contents from client
  if (ret == 0) {
    // eof: all done
    close(c->fd);
    close(fd);
    FD_CLR(fd, &rd);
    break;
  }
  if (write(c->fd, c->buf, ret) == -1) {
    // write to file failed: terminate connection to client
    ...
    break;
  }
  // continue to read more data from client
  break;
}
```


Write Event (1)

```
// File descriptor fd is ready to be written to. Write to it, then,
// depending on current state, prepare for the next action.
void write_event(int fd) {
  client_t *c = &clients[fd];
  int ret = write(fd, c->buf, c->size);
  if (ret == -1) {
    // couldn't write to client; terminate connection
    close(c->fd);
    close(fd);
    FD_CLR(fd, &wr);
    c->fd = -1;
    perror("write to client");
    return;
  }
  switch (c->state) {
```


Write Event (2)

```
case BAD:
  // finished sending error message; now terminate client connection
  close(c->fd);
  close(fd);
  FD_CLR(fd, &wr);
  c->fd = -1;
  break;
```


Write Event (3)

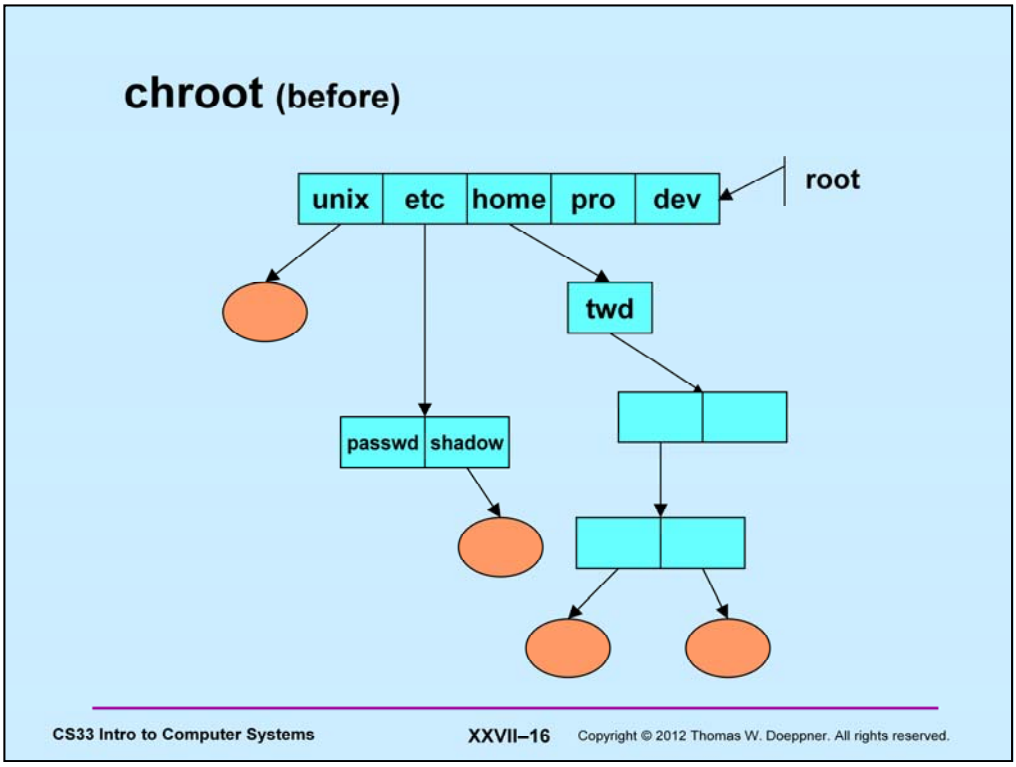
```
case GOOD:
  if (c->dir == IN) {
    // finished response to PUT request
    c->state = TRANSFER;
    FD_SET(fd, &rd);
    FD_CLR(fd, &wr);
    break;
  }
  // otherwise finished response to GET request, so proceed
```

Write Event (4)

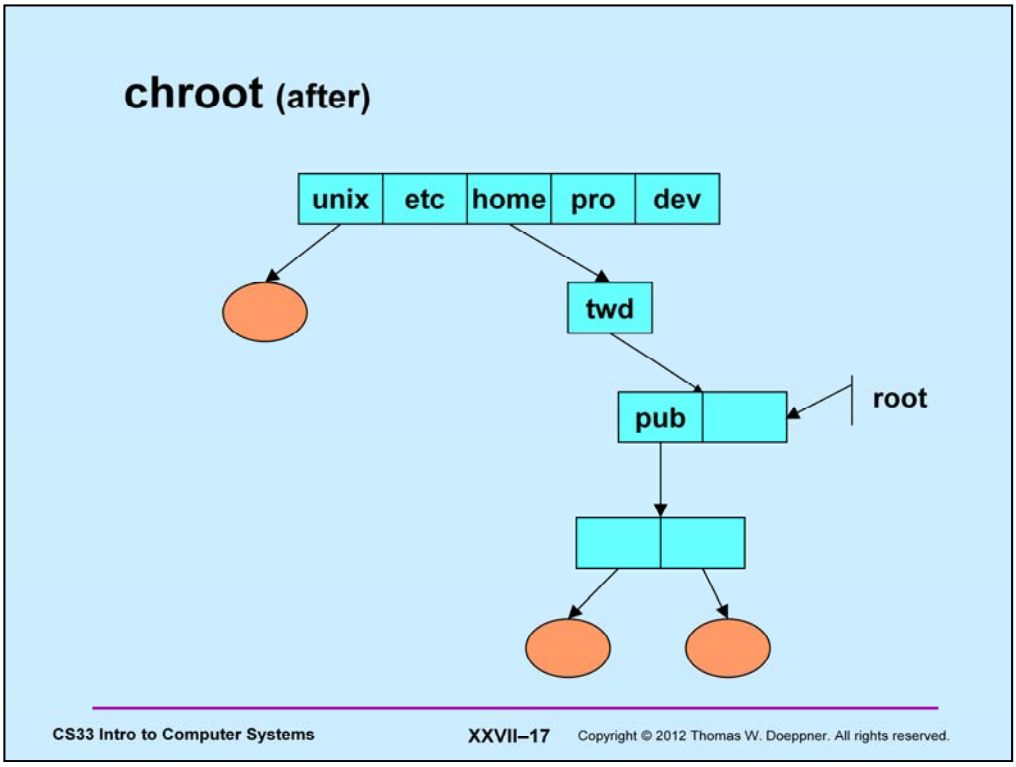
```
 case TRANSFER:
   // should be in midst of transferring file contents to client
   if ((c->size = read(c->fd, c->buf, BSIZE)) == -1) {
     ...
     break;
   } else if (c->size == 0) {
     // no more file to transfer; terminate client connection
     close(c->fd);
     close(fd);
     FD_CLR(fd, &wr);
     c->fd = -1;
     break;
   }
   // continue to write more data to client
   break;
 }
}
```


Problems

- Works fine as long as protocol is followed correctly
  - can client (malicious or incompetent) cause server to misbehave?
- How does server limit file access?



The *chroot* system call allows a process to change the root of its file-system hierarchy to a directory lower down in the tree.



After a successful invocation of *chroot*, paths starting with "/" are followed starting at the new root.

Secure chroot?

- Implementation
  - ".." = "." at process's root
    - » can't *cd* to parent
- Secure?
  - leakproof?

No …

```
chdir("/");
pfd = open(".", O_RDONLY);
mkdir("Houdini", 0700);
chroot("Houdini");
fchdir(pfd);
for (i=0; i<100; i++)
  chdir("..");
chroot(".");
```


The problem is that current open files are not closed, even if they are outside of the subtree to which access is restricted.

The *fchdir* system call changes the current directory to be the directory referred to by the file descriptor passed as an argument.

chroot

- Would work fine for our simple file transfer protocol
  - actually is used in *tftp* (trivial file transfer protocol)
  - however, requires superuser powers (so as to help avoid problems of previous slide)