# 6.033
# COMPUTER SYSTEMS ENGINEERING
# RECITATION 3: UNIX

JOHN WANG

## 1. The UNIX Filesystem

1.1. **Timesharing.** There were systems that people were using when UNIX was being developed where the files required 80 characters per line. This is a relic of physical cards. Files looked like punchcards, with a fixed line width. Advantage is that the OS can enforce a set of rules to the filesystem.

In UNIX, a file is just an array of bytes, doesn't impose any structure to the file. Advantage is that you have flexibility.

On other timesharing systems, each user obtained their own block of disk space. Advantage is fewer disk accesses, so better performance since you read memory sequentially. In UNIX, there is a file system tree, and each directory or file is owned by a particular user. This checks permissions on the files.

1.2. **Special Files.** For instance, devices are special files: */dev/mouse* or */dev/sound*. Making everything a file makes it very easy to chain things together. There are also things like */dev/random* which seem like files.

1.3. **Links.** There are two types of links, soft and hard links. Soft link: a human readable path to a file. Hard link: points to an inode number. Hard links cannot point to a directory, but soft links can. Moreover, hard links cannot point across file systems or mounted devices.

Hard links can't point to directories because if they do, then they can point to themselves, so even if a directory has no more external links, it still has a non-zero reference count. This prevents deletion of the directory. This same logic works for any disconnected cycle, so the system will never be able to reclaim it.

## 2. Processes and Pipes

2.1. **Images.** "Image" is the state of a process. It has all of he memory and registers, as well as UNIX specific state, such as process id, working directory, etc. An image also contains file descriptors. Procedures like *open(filename,...)* returns a file descriptor. File descriptors are usually small integers. Standard input, output, and error messages are also file descriptors.

2.2. **Forking.** "fork()" copies the entire process over to a child process. One process will be the child and one will be the parent, determined by the return values of the "fork()" method.

2.3. **Piping.** "pipe()" creates a new open file descriptor where one process can write to it, and one process can read from it.

```
rd, wr = pipe(); \\ get new file descriptors
if (fork()) {
  \\ replace standard out with the write file descriptor
  dup2(wr, stdout);
  close(rd);
} else {
  \\ replace standard in with the read file descriptor
  dup2(rd, stdin);
  close(wr);
}
```

2.4. **Writing a Shell.** "execute()" takes in a program name and runs it. "wait()" pauses the process until one of its children is finished. Once the child is finished, it calls the "exit()" system call. To write a shell, we want the following

```
if (fork() == 0){
  execute(‘‘process_name’’);
} else {
  wait();
}
```