UNIX History

1965-1969 Bell Labs participates in the Multics project.

1969 Ken Thomson develops the first UNIX version in assembly for an DEC PDP-7

1973 Dennis Ritchie helps to rewrite UNIX in C for PDP-11

1976 UNIX V6 common among American Universities.

1979 UNIX V7. First portable version of UNIX.

1979 UNIX V7 ported to VAX11 as UNIX/32V

1980-1985 Berkeley university develops 4.1BSD, 4.2BSD and 4.3BSD

1982-1985 Bell that now has become AT&T develops system III and system V

1985 Sun Microsystems uses 4.2BSD as a base for SunOS

1986 4.3BSD complete.

1992 NETBSD 0.8

1993 FreeBSD 1.0. Linus Torvalds puts out first version of Linux on internet.

1994 4.4BSD

1995 4.4BSD Lite-2. The last UNIX distribution from Berkeley.

1996 OpenBSD 2.0

UNIX Kernel

The kernel is the central part of the operating system and is always resident in the primary memory.

The kernel performs among other things the following tasks:

- Process handling
- memory handling
- I/O handling

User processes requests service from the kernel via system calls.

- UNIX systems have many concurrent processes.
- New processes are created with the system call fork. Fork creates a new child process that is an exact copy of the parent process.
- Both the processes (parent and the child process) continues to execute at the instruction after fork.
- Fork returns the value 0 in the child process and a value bigger than 0 in parent process.

UNIX Processes

Memory handling

A UNIX process is divided in three logical segments:

- The code segment begins on virtual address 0. The code segment is (usually) write-protected and can be shared between different processes executing the same program.
- The data segment follows after the code segment and grows upward (against increasing addresses)
- The stack segment begins on the highest virtual address and grows downward (against decreasing addresses)

System Calls for Process Handling

Pid=fork() Create a child process.

exit(status) Terminate the process that called exit.

Pid=wait(status) Wait for a child process to terminate.

execve(name,argv,envp) The code and data segments in the process that calls execve is replaced with data from the file "name". Open files, current directory and other status information is unchanged. The call on execve returns only if the call fails (otherwise there exists nothing to return to)

Input/Output

- All I/O in UNIX is carried out on byte streams.
- All structure in byte streams are defined by application programs.
- The system calls read and write works with unstructured byte streams.
- A byte stream can be linked to an arbitrary external unit. For example a disk memory, a terminal or a tape unit.
- Read and write works in the same way for all types of external units.

External Devices

- External devices can be reached via *special files* in the file system.
- Of convention all the *special files* are located in the directory /dev.
- In the operating system, an external device is identified by it's *device number*.
- A device number can be divided in a *major device number* and a *minor device number*.
- The *major device number* identifies which driver that deals with the device.
- The *minor device number* is used by the driver to distinguish between different logical units, handled by the same driver.
- Each special file corresponds to a unique device number and is actually only a link to a device driver.
- *Special files* are read and is written in the same way as normal files.

Descriptors

- A descriptor is a small positive number, often in the interval 1-32.
- A descriptor identifies an open file or other byte stream within a certain process.
- Descriptors are created with the system calls *open*, *pipe* or *socket*.
- Each PCB contains a table of open files. The system call *open* returns a descriptor that actually is an index in this table. This table also have a pointer to the file's inode.
- The system calls *read* and *write* uses the descriptor to identify which file to read or write.

System calls for Input/Output

All I/O in UNIX is handled mainly by the following system calls: read, write, open, close, lseek and ioctl.

Open returns a descriptor for a new or existing file.

```
fd=open ("fil1", flags, mode)
```

The *flags* parameter specifies the access mode and the *mode* argument specifies permissions.

Reading and writing are done by:

```
bytesread=read (fd, buffer, bytesdesired)
byteswritten=write (fd, buffer, byteswritten)
```

- Each open file has an *I/O pointer* that tells from where in the file the next byte will be read.
- After open the *I/O pointer*, points to byte zero.
- After read or write the *I/O pointer* is positioned at the byte after the last byte read or written.

The *I/O pointer* can be repositioned with the system call lseek ("long seek"):

```
position=lseek (fd, offset, whence)
```

Filters

Output from a process can be sent directly to another process.

A process that reads an input data stream and produces an output data stream is called a *filter*.

Example:

```
ls | sort | wc -l
```

A program that shall be used as a *filter* have to read from “standard input” and write to “standard output”

UNIX File System

- The UNIX file system has a hierarchical tree structure with the top in root.
- Files are located with the aid of directories.
- Directories can contain both file and directory identifiers.
- The user identifies files with absolute or relative path names.
- Example on absolute names: /usr/terry/notes/apr22.txt
- Each user has a login directory. The user can create his own subdirectories within the login directory.
- The system has information about a users *current directory*. At login, the home directory is set as *current directory* but this can be changed with the command *cd*.
- The operating system uses inodes as the internal name for files. An i-number is an index in a table of inodes.
- An entry in the inode table contains complete information about a certain file. A directory only contains a translation from "path name" to i-number.

Pipes

- A pipe is an one way buffered channel between two processes.
- Reading and writing to a pipe is done with the standard *read* and *write* system calls.
- Read and write operations to a pipe is blocking. Read blocks when reading from an empty pipe and write blocks when writing to a full pipe.
- Pipes can only be used between processes that are related with each other.
- A pipe is created with the system call pipe, that returns one file descriptor for reading and one file descriptor for writing.
- After a *pipe* is created, the system call *fork* is used to create a new process to communicate with.
- Current UNIX systems also have other facilities for communication, such as *sockets*.

System calls for Pipe

```
int pipe(int filedes[2]);
```

Creates a buffered channel (pipe) for communication with another process. Two file descriptors are returned. filedes[0] is for reading, filedes[1] is for writing. The process must use fork to create a child process. The two processes can then use the *pipe* for communication.

```
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

Duplicates a file descriptor.

Shell

The command interpreter in UNIX is called the *shell*.

The shell is a normal user program that reads command lines entered by the user and interprets them as commands to be executed.

Several different shells are in common use:

sh (Bourne shell) exists on almost all UNIX systems.

csh (C shell) developed at Berkeley.

tcsh An extended version of csh.

ksh (Korn shell) is used above all on UNIX systems V.

bash (Bourne Again Shell) from GNU is the most common shell on Linux.

There also exists several graphical user interfaces for UNIX. The most common is the X window system.

Shell

The simplest form of command comprises a command name with arguments.

For example: ls - l

If the command name begins with / the shell tries to execute the file with that name.

If the command does not begin with / the shell looks for the command in a number directories.

Which directories to search and in which order is determined by the environment variable PATH.

```
Example: PATH=/local/bin /bin /usr/bin /usr/ucb
```

Programs executed by the shell has three open files:

Standard_input (file descriptor 0) reads from the terminal.

Standard_outputs (file descriptor 1) writes to the terminal.

Standard_error (file descriptor 2) writes to the terminal.

Shell

Commands

- Simple commands.
  - → for example: ls -l
- Redirection of stdin and/or stdout.
  - → pgm <file1 >file2. Data is read from file1 and written to file2.
- Redirecting stdout and stderr to the same file.
  - → pgm >& file1
- Background jobs.
  - → pgm& Give prompt for the next job without waiting for *pgm* to terminate. Background jobs can be moved to the foreground by the command *fg* in csh.
- Pipes.
  - → sort file1 | head -20 | tail -5
- Built in commands
  - → cd (change directory). Some commands need to be executed internally by the shell (Normal commands are executed by a child process)