Contents of Lecture 1

- A definition of an Operating System
- Command line interpreters
- Processes
- Implementing a simple command line interpreter in UNIX (Lab 1)

A definition of an Operating System

- An OS is an abstract machine which can run a number of programs.
- The programs can talk to the OS using special functions called *system calls*.
- The OS is also a government/police which controls the hardware so that:
  - the processes do not destroy each other — provide *protection*,
  - the hardware is used efficiently — e.g. using *multiprogramming*, and
  - it is convenient for programmers — e.g. using *virtual memory*.
- An OS is not about user interfaces although all operating systems should have a nice user interface (e.g. a command line interpreter with vi-compatible editing commands (vi is the standard UNIX editor)).

Some terminology

- A program is an executable file stored on disk and when it is running in the computer it is called a **process**.
- There can be hundreds of **system calls** dealing for instance with
  - loading a file from disk and start executing it (`execve`)
  - creating a file (`creat`)
  - reading and writing a file (`read/write`)
  - removing a file (`unlink`)
  - asking what time it is (`gettimeofday`)
  - creating a new process (`fork`)
  - terminating execution (`exit`)
  - killing another process (`kill`)

Some more terminology

- **Protection:** a process can only read and write its own memory. Earlier systems such as MS-DOS did not offer this facility.
- **Multiprogramming:** while one process is waiting for some event such as disk access being completed, instead of just doing nothing, the OS runs another process in the mean time. This is to make more efficient use of the hardware.
- **Virtual memory:** a technique to make the RAM look much larger than it actually is so that programmers don't have to worry (so much) about how much memory is used.

Three aspects of a modern computer

- **Dual mode processor:** user-mode and supervisor-mode. Supervisor-mode is police-mode which can do anything. User-mode is for running normal programs. If something exceptional/illegal is attempted, the processor automatically switches to supervisor-mode. The ”mode” is stored in a one-bit hardware register in the processor.
- A **real-time clock:** the clock interrupts the processor (say, 100 times per second) and lets the OS decide if the current user process has run for too long. By switching user processes often, the user thinks all programs are running in parallel but this is not true (on a uniprocessor).
- A **kernel:** the kernel is a program, often implemented in C and assembler, and acts as the government/police.

More about the kernel

The kernel has four main things to do:

- **Scheduling:** decide which process run next
- React to **system calls** from the running process: the kernel knows e.g. how to open a file
- React to **exceptions** from the processor: e.g. a divide by zero, or, TLB fault (TLB = translation lookaside buffer, a part of the VM system (VM = virtual memory)), or
- React to **interrupts** from peripheral devices: e.g. take care of a network packet and forward it to the proper user process.

The main difficulty in writing a kernel is the same as for much other software: getting it correct, fast, and easy to maintain

But why do we really need a kernel in the first place?

- Without process scheduling and protection, chaos will reign. Bad.
- Without the abstraction of the system calls, user programs would have to deal with e.g. disk access interfaces. Too messy and not portable.
- With centralised control, all processes running e.g. emacs on a machine can share parts of memory in a read-only fashion. All processes using printf can share the same copy in RAM.
- Sharing the memory for instructions in system libraries such as printf is done through **shared libraries** and **dynamic linking**.
- The alternative is **static linking** in which the compiler produces a complete file with all instructions.

Definition of a process

- A process is an entity which can execute a program (note that it can decide to execute another program by loading a different file — registers and memory contents are lost but e.g. opened files remain open).
- Each process has among other things:
    - instruction pointer (PC), integer and floaing-point registers,
    - memory,
    - process state: one of running, ready, waiting, etc
    - credentials: privileges associated with the process owner
- The scheduler in the kernel decides which process should run.

Measurement Hardware

Machine used for performance measurements

- `power.ludat.lth.se`
- Apple PowerMac Quad G5
- 2 x IBM 970MP Power/AltiVec microprocessor, i.e. 4 CPU's
- 2.5 GHz
- 6 GB RAM
- OS: Linux 3.3.4
- Measurements are made using `lmbench-3.0` available at

Getting services from the kernel: system call

- The caller's code for a system call is exactly like a normal function call.
- The details are hidden in a user-level library code in the C library (assembly code).
- In addtion to the normal parameters, a special parameter, the system call number is stored in a CPU register.
- Instead of using the normal function call machine instruction, a special *system call instruction* machine instruction is used in the assembler routine.
- The system call instruction does two things:
    1. switches from user to supervisor mode
    2. jumps to a predefined address in the kernel

More about system calls

- Once the process has exectued the system call instruction, it hands over control to the kernel
- It takes 0.53μs (or 290 clock cycles) to do a ”null” system call (just entering and leaving the kernel) on the PowerBook.
- Avoid system calls if possible.
- How to avoid system calls???
- If possible, use buffered I/O (along with fflush(FILE*) when output is required). If not possible, try at least line-buffered I/O.

Example of using buffered I/O 1(2).

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

#define BUFFER_SIZE     (2<<15) /* 32 KB */

/* Three buffering alternatives provided by the C standard li
 *      _IOFBF -- fully buffered.
 *      _IOLBF -- line buffered.
 *      _IONBF -- no buffering.
 */
```

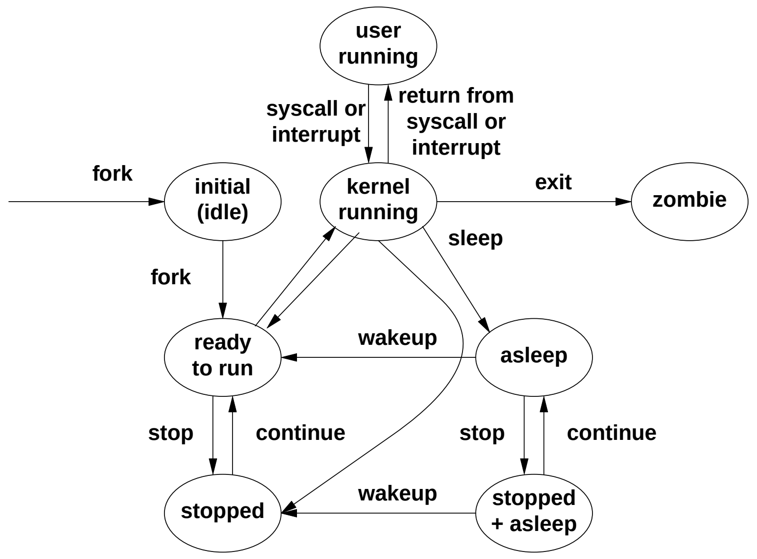
Example of using buffered I/O 2(2).

```
int main(int argc, char** argv)
{
        void*   buf;            /* User-supplied buffer for stdout. */
        char    line[1000];     /* One line of input. */

        buf = malloc(BUFFER_SIZE);
        assert(buf != NULL);

        setvbuf(stdout, buf, _IOFBF, BUFFER_SIZE);
        printf("how are you? ");
        fflush(stdout);  /* flush output to screen. */

        fgets(line, sizeof line, stdin);
        printf("%s? --- good to hear.\n", line);

        /* no fflush needed since exit will do that. */
        return 0;
}
```

Processes in UNIX

- Process states
- Process context
- User credentials
- Data structures: `u area` and `proc structure`
- Signals
- Creating new processes

UNIX Process States



Process context

- Integer and floating-point CPU registers (and vector registers on G4/G5).
- Memory: code, data, stack
- Special hardware registers: MMU data, processor status word, etc
- Credentials: real and effective UID/GID
- `u area` and `proc structure`: where process info is stored
- Environment variables: HOME, PATH, LD_LIBRARY_PATH, etc

Credentials

- Every user has a user id (UID) and a group id (GID)
- Every process has a pair of IDs called *real* and *effective* which are set at login to UID.
- The Superuser, (root), has UID = 0 and GID = 1 and may access any file
- Effective UID/GID are used when a file is opened to check privileges
- A *suid mode* program sets the effective UID/GID to the programs owner

Suid programs 1(2)

- Suppose you have a file `A` which should only be modified in some controlled way.
- Start by setting the access rights to read-only for everyone except yourself:
  `$ chmod 644 A` (results in rw-r–r–).
- Then write a program `P` which the others can use to modify the file (such as adding their name at the end of the file).
- Let the others run your program: `chmod 511 P`. (results in r-x–x–x)
- But, when they run your program, their processes will not have permission to modify your file!

Suid programs 2(2)

- When running your program, the process needs *your* write permission.
- Solution: Make your program SUID: `$ chmod +s P`. (r-s–x–x).
- Now the kernel will set their processes' effective user id to the owner of the program `P`, which is you, and you can modify `A` so they can as well.
- In Kernighan/Pike "The UNIX Programming Environment" 1984 (the classic introduction to UNIX), they suggest protecting high-score files using a SUID program.

U Area

- CPU registers for a process which is waiting for soon being scheduled.
- Real and effective UID/GID
- Pointer to the proc structure
- Argument to system calls and a stack for executing a system call
- Pointer to the current directory
- Information about signals, opened files, and memory

Proc Structure

- Process id (pid), process group, process state, priority, and a pointer to the u area
- Pointers to other processes in a doubly linked list in one of the scheduler's priority queues.
- Sleep channel for a blocked process.
- Information about signals
- Pointers to other procs to make a tree: parent, first child, and sibling.

Process groups

- Every process belongs to a process group.
- A process group is identified with the pid of the leader of the group.
- Every command in a shell creates a new process group, e.g.:

```
$ grep -i unix *.tex | awk -F: ' { print $1; } ' | uniq
```

- The execution of process groups can be controlled with commands such as: Ctrl-C, Ctrl-Z, bg, fg as we will soon see.

UNIX Signals

Signals is a mechanism of notifying a process that something has happened

Signal	**Cause**	**Default effect**
SIGINT	Ctrl-C	Terminate the process
SIGSTOP	Ctrl-Z	Stop the process
SIGSTOP	kill -SIGSTOP <pid>	Stop the process
SIGCONT	kill -SIGCONT <pid>	Resume the process
SIGILL	invalid instruction	Terminate the process
SIGSEGV	eg access of kernel data or addr 0	Terminate the process
SIGBUS	eg non-alignad memory access	Terminate the process
SIGFPU	eg division with zero	Terminate the process
SIGKILL	kill -SIGKILL <pid>	Terminate the process
SIGKILL	kill -9 <pid>	Terminate the process

More about signals

- A process can send a signal to another process with the same real UID.
- kill -9 on others' processes don't work unless you are root.
- A process can register a signal handler which will be executed instead of the default effect (except SIGKILL which cannot be overridden).
- The system call `kill(pid, s)` sends the signal s to pid if pid > 0; and if pid < 0 the signal is sent to all processes in the process group -pid.
- It takes 1.6 μs, or 900 clock cycles, to install a signal handler on the PowerBook.

Job control

- Every terminal window has a process group in the *foreground*.
- Ctrl-Z stops the process group in the foreground.
- The shell command `bg` moves a stopped process group into the *background* and resumes it.
- The shell command `fg` moves a process group into the foreground.
- With multiple process groups in the background, they are named either with *job number* eg %3 or pid.

Creating new processes in UNIX

- A new process is created with the system call fork.
- Time to fork is 460 μs or 253000 clock cycles (on the PowerBook).
- Fork returns -1 on failure, 0 to the child, and the child's pid to the parent.
- The child process is a copy of the parent with a few exceptions:
  - the child gets its own process ID
  - the forking process becomes the parent of the child (in the proc struct).
  - the child gets its own file descriptors (referring to the same files though)
  - waiting signals are removed (ie signals not yet delivered to the parent).

Using the fork System Call

```
int main()
{
        int     child_pid;      /* Child process id. */
        int     child_status;   /* Exit status from the child. */

        if ((child_pid = fork()) < 0)
                error("fork failed");
        else if (child_pid == 0)
                printf("i am the child with pid: %d\n", getpid());
        else if (want_to_wait)
                waitpid(child_pid, &child_status, 0);
}
```

Finding and loading an executable file 1(3)

- The UNIX shell uses the environment variable PATH to search for commands.
- PATH is a colon-separated list of directories. In C/C++:

```
char* s = getenv("PATH");
```

- Suppose the shell is searching for the command `gcc`.
- The system call `access()` tells whether a file exists and the process has suitable rights.
- The system call `execv()` loads an executable file into the process' memory and starts executing it (dropping the previous program).

Finding and loading an executable file 2(3)

```
char*   command;        /* Command we are looking for, eg "gcc". */
char*   directory;      /* Pathname of a directory, eg "/usr/bin". */
char    file[BUFSIZ];   /* File to load. */

/* Produces eg "/usr/bin/gcc" in file. */
sprintf(file, "%s/%s", directory, command);
if (access(file, X_OK) == 0) {
        /* We found the file and we are allowed to exec it. */
        argv[0] = file; /* First element of argument vector is the path. */
        execv(argv[0], argv); /* Path and argument vector to the program. */
        /* Should not come here unless execv fails. */
}
```

File descriptors

- Files in UNIX are referred to by the process using integers.
- These integers are indices into the process table of open files.
- The table elements have references to opened file data structures.
- 0 is `stdin`, 1, `stdout`, and 2 is `stderr`.
- When a new file (or pipe, see next slide) is opened the first free position in the open-file-table is returned to the process as return value from the kernel.

Pipes 1(2)

- To count the number of Java files in a directory, we can do:

```
ls *.java > javafiles
wc -l < javafiles
```

- A pipe is a special anonymous file which is used to let two processes communicate with each other.
- Command to count the number of Java files in a directory: `$ ls *.java | wc -l`
- Here `ls` prints all Java file names on its stdout which becomes the stdin of `wc`, which count the number of lines of input.

Pipes 2(2)

- The system call `pipe(int fd[2])` creates a pipe with `fd[0]` for reading and `fd[1]` for writing.
- To setup the pipeline `ls | wc`, the shell must organise file descriptors as follows:
  - File descriptor 1 of `ls` should be referring to what `fd[1]` refers to.
  - File descriptor 0 of `wc` should be referring to what `fd[0]` refers to.
- The `int dup2(int oldfd, int newfd)` system call copies the pointer in the open-file table at position `oldfd` to position `newfd` (if `newfd` referred to an open file, it is first closed).

A simple command interpreter: Lab 1

- Fetch the `PATH` environment variable using `getenv(PATH")` and make a list of directory names (already done for you in the Lab).
- Read a line from stdin and parse it and split it up into items separated by the pipe symbol `|`. This is also done for you already.
- For each item, the first ”thing” is a command name. Search for it. The rest are arguments.
- The interesting part is setting up the pipes.