Contents of Lecture 2

- OS implementation languages
- Monolitic kernels
- Microkernels
- Threads
- Signals
- Interprocess communication

Preferred implementation language for operating systems

- A long time ago (before UNIX) operating systems were written in assembler code to improve speed.
- Multics was written in IBM's PL/1 during the 1960's but their compiler was buggy.
- MPC was written in Algol for a Burroughs machine.
- The first successful OS written in a high-level language, C, was UNIX.
- MS-DOS was written in 8086 assembler.
- BSD UNIX, Linux and Windows XP were written in C.

Why C?

- Advantages over assembler language:
  - Easier to write in C than in assembler, of course.
  - Portable — that was in fact the reason for using C for UNIX.
  - Faster code in reasonable time. The programmer should not waste time on things the compiler can do faster and often usually better.
  - Some things have no support in C and must be written in assembler.
- Advantages over other high-level languages:
  - Reliable mature compilers are available.
  - Easier to debug if you write carefully (because you have complete control).
  - Easier to write fast code because you have complete control.

Structure of operating systems

- In MS-DOS, applications have access to everything including BIOS.
- BIOS stands for Basic I/O System, and for MS-DOS was used to do I/O.
- Nowadays, BIOS is used to initialise the hardware and to fetch the so called boot loader (a small piece of software) which will fetch and start the kernel.

Monolitic UNIX

- The UNIX kernel was written as one C program.
- The advantage is that it ***potentially*** can be faster.
- A disadvantage is that it ***potentially*** can be less well structured.
- Linux is monolitic but supports dynamic loading of modules, eg to load a device driver into kernel memory.

Microkernels

- In a microkernel only the most essential parts are left in the kernel and the remaining system calls are supported by servers, eg file systems.
- Typically interprocess communication, processes, and memory are handled by the microkernel.
- The advantages include
  - Improved robustness (a crash in a server does not need to crash the system — the microkernel can restart the server)
  - Faster porting to other hardware — see eg what Apple did with Mach for the iPhone (new hardware).
  - A microkernel can support multiple operating systems' interfaces simultaneously — eg one server for MSDOS, one for Windows 7, and one for UNIX.

More about microkernels

- For an application program, the familiar UNIX system calls look the same.
- A system call instruction causes the microkernel to execute.
- If the microkernel should not service that system call, it will compose and send a message to the appropriate server.
- So a disadvantage of microkernels is that they imply a lot of message passing which makes them slower, but if your program mostly computes instead of makes system calls, that will not be noticable for most persons.
- In my limited experience Linux is often faster than MacOS X but not enough to affect which OS I decide to boot.
- Windows NT started out as a microkernel design but became more and more monolitic.

Modules

- An alternative to a monolitic kernel with everything needed compiled into the kernel at build time, is the concept of modules.
- In Linux and Solaris, one can add code to a running kernel.
- Such code pieces are called modules.
- The advantage of this approach over a microkernel is that the message passing overhead is removed — ie a faster kernel.
- The reason modules are faster is that communication between the different subsystems is through normal function calls instead messages.

MacOS X

- The MacOS X uses both a microkernel and modules.
- On top of the microkernel **mach** is the FreeBSD kernel.
- In addition MacOS X has **kernel extensions** (Apple term) which are loadable modules. These are used for device drivers.
- As expected, Apple is the company which has sold most UNIX systems.

Virtual machines

- Suppose we wish to run different operating systems concurrently on the same machine.
- One way to do so is to have lower level software below the kernel which provide a virtual machine.
- The kernels (and the rest of each OS) think they have their own machine.
- For decades, IBM has had such a system in their IBM VM operating system.
- One issue to deal with is disk space. The disks must also be virtual and each OS is given a part of a disk — IBM calls these minidisks.

More about virtual machines

- In IBM VM user instructions are executed on the real machine and privileged instructions are simulated.
- Virtual machines are perfect for OS development since a crash does not require rebooting the real machine.
- A popular VM is VMware which abstracts an X86 machine.
- Suppose you are testing an application on Linux, FreeBSD, and a Windows version. Either buy three machines, or boot each OS one after the other, or run all three OSes on VMware virtual machines and test the application much cheaper and efficiently.

Limitations of Processes

- Traditional UNIX (and other OSes) processes are single-threaded.
- Many important applications contain parts which can run concurrently.
- With only processes, it's difficult and inefficient to parallelize them.
- With multicores it becomes even more important to have threads.
- Modern UNIXes support threads.

Writing Parallel Servers without Threads

- A server, eg for a database, had the following structure.
  - The server listens to client requests.
  - At a request, the server forks a new process.
  - The new process services the client request.
- Advantage: concurrent I/O for different processes
- Disadvantages:
  - fork is an expensive system call
  - more work to set up sharing of memory between processes
  - sharing eg network connections between processes is not supported

Using threads

- Threads are units of execution which share an address space.
- Disadvantages:
  - data must be protected from other threads
  - and (therefore) not all compiler optimizations can be performed
  - for details read about memory consistency models — see eg EDAN25
- Advantages:
  - sometimes easier programming
  - concurrency among threads especially on multicores
  - for uniprocessors: while one thread is waiting another can execute

More about performance benefits of using threads

- **Responsiveness**: Multithreaded applications can proceed while one part is blocked waiting for I/O, eg web browsers or servers.
- **Efficiency**:
  - On Solaris, creating a process is 30 times slower than creating a thread, and
  - also on Solaris, context-switching between processes is 5 times slower than between threads.
- **Parallelism on a multicore**: To achieve high-performance on a multicore we need one thread per processor. Useful eg in numerical computations but also eg in VLSI simulation, and many other areas — but often you must parallelise the program by hand.

A multithreaded server

- A process contains resources and threads.
- The process' credentials are shared by all threads.
- To serve a client C_1 with certain permissions, the server must use system calls `setuid`, `setgid` and `setgroups` to match C_1.
- To concurrently serve another client C_2, the server must therefore switch credentials to match C_2
- Therefore the security checking operations must be serialized.

Kernel threads

- The kernel has threads which need not be associated with a user's thread.
- They can access the kernel text and global memory, and each has a separate stack.
- A kernel thread can be used for a specific task eg asynchronous I/O
- Asynchronous I/O means nonblocking reads and writes for which the request is queued and a `SIGPOLL` signal is delivered when the request is completed.
- Older UNIX kernels had separate processes for eg taking back page frames from processes (the pagedeamon) — this task is better implemented using kernel threads.

Lightweight processes (LWP)

- Based on kernel threads.
- Kernel-supported user thread.
- Initially a lightweight process supported multiple user threads but the trend is to have a one-to-one mapping of user thread to lightweight process (or kernel thread).
- Linux uses a one-to-one mapping and Solaris has switched to it.

User threads

- Based on lightweight processes.
- There is a user level scheduler which is responsible for scheduling user threads without kernel knowledge.
- This is invoked when a user thread blocks eg waiting for a lock.
- Asynchronous I/O is complex and avoided by providing a synchronous interface (the usual read and write) which uses asynchronous I/O and switches user threads while waiting for the queued I/O request.

Upcalls

- The kernel schedules the LWP but knows nothing about what the user thread is doing
- The user thread might own a spin lock.
- An **upcall** is a call from the kernel to the user thread library and is used to let the thread library schedule a new thread at a blocking system call.

Interrupt handling using threads

- Interrupt handlers may modify critical kernel data which must be protected.
- Traditionally the **interrupt priority level** (IPL) was used to block interrupts (or other interrupts) when such data was modified.
- On multicores, this is particularly inefficient — since the IPL must be changed on all CPUs.
- Blocking urgent interrupts can slow down the system.
- In Solaris interrupts are handled by kernel threads which have the highest priority and are synchronized with the rest of the kernel using normal facilities such as mutexes.
- For this, Solaris has a pool of interrupt threads ready to be used.
- Interrupts are relatively infrequent but changing the IPL was frequent, so using threads optimizes the common case.

Signals and Threads

- Solaris and other systems distinguishes between two types of signals:
  - Synchronous signals called **traps**, and
  - Asynchronous signals called **interrupts**.
- Synchronous signals include SIGSEGV, SIGILL, SIGBUS, and SIGFPU.
- Asynchronous signals include SIGPOLL, SIGTERM and SIGKILL.
- Synchronous signals are delivered to the thread causing it, while asynchronous are delivered to the first thread with them enabled.
- A thread can install its own handler for synchronous signals.
- Asynchronous signals have a common set of handlers.

UNIX Signals

- A signal is a simple way for the kernel to tell a process that an event has occured.
- When a signal is delivered to the process either a default action of killing the process, or a function in the process is called.
- The original design had some problems making them unreliable (see below) which were fixed in an incompatible (with System V from AT&T) way Berkeley in 4.2BSD.
- Today all UNIX implementations are POSIX-compliant and portable.

Generating and Delivering a Signal

- A signal is generated when an important event occurs.
- It is said to be **pending** until it is **delivered** to the process.
- The kernel calls `issig` to check whether a process has a pending signal:
    - Before the process blocks on an interruptable event.
    - Immediately after the process wakes up from an interruptable event.
    - Before returning to user mode from kernel mode.
- If there is a pending signal, the kernel calls `psig` to either terminate the process or to call the signal handler using `sendsig`.
- The function `sendsig` manipulates the process' stack so that it ***almost*** looks as if the signal handler was called from the program.

Leaving a Signal Handler

- Normally a signal handler returns when it is finished but there are other options.
- The signal that was delivered is blocked while executing the handler.
- The previous slide said it ***almost*** looks as if the signal handler was called from the program.
- The delivered signal should be unblocked when the handler returns.
- ***If*** the signal handler returns, execution is resumed where the process was stopped.
- If the signal handler instead calls `longjmp/exit/abort` the signals blocked for the duration of the handler remain blocked — see `sigaction` below.
- If the process was in a system call it usually fails and `errno` is set to `EINTR`.

Reasons for Generating a Signal

- Some hardware exceptions — eg trying to access an invalid address
- Other process uses `kill` or `sendsig`
- Terminal interrupts — user sends CTRL-C to the foreground process
- Job control — a back ground process doing terminal I/O gets a signal
- Quotas — when a quota is exceeded the process is notified with a signal
- Notification — of eg a device being ready for I/O
- Alarms — the time for an alarm has been reached (see `alarm`).

Example Signal Delivery

- The user hits CTRL-C. The terminal driver is run to service the interrupt and sees it was CTRL-C.
- The terminal driver of this terminal then sends the `SIGINT` signal to the foreground process (or processes if its eg a pipeline).
- When a receiving process is scheduled to run and returns to user mode from the kernel, the `issig` function discovers the signal.
- Previous point also works properly if the receiving process was the interrupted process.

Two Kinds of UNIX Sleep

- Sleep means a process being blocked and UNIX distinguishes between two kinds:
    - Interruptable — long term sleep such as waiting for terminal input
    - Uninterruptable — short term sleep such as waiting for a disk block
- The kernel wakes up the process if it was interruptable and delivers the signal
- The signal becomes pending in the other case.

Unreliable Signals

- Before 4.2BSD signals were unreliable.
- The problem was that the signal handler was removed at delivery.
- Signal handlers therefore usually reinstalled the handler.
- If there should be a handler for `SIGINT` and a user hits CTRL-C very quickly the result could be a terminated process!
- Signal handling as specified by POSIX is used in Linux:
  - Handlers are persistent — they remain until another is installed
  - Masking (or blocking) — a signal can be temporarily blocked
  - Some signal information is moved from the `u area` to the `proc structure` to avoid having to wake up a process to see if it has a handler for a signal.
  - The `sigpause` system call atomically unmasks signals and waits for a signal.

UNIX System V Release 4 Signal Functions — used in Lab 2

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

- `sigprocmask` manipulates the set of blocked signals
- The values of `how` determine the behavior:
  - `SIG_BLOCK` — the signals specified in the set are added to the set
  - `SIG_UNBLOCK` — the signals specified are removed from the mask
  - `SIG_SETMASK` — the set becomes the new mask.
- the old set is saved in `oldset` if non-null.

More SVR4 Signal Functions: sigaction

```
int sigaction(
        int                       signum,
        const struct sigaction*   act,
        struct sigaction*         oldact);
```

- The `sigaction` function is used to specify a signal handler for the signal `signum`.
- The handler is specified as a field of `act`.
- A mask for this signal number: which additional signals should be temporarily blocked until the handler returns.
- Thus these signals remain blocked if the handler calls `longjmp` instead of returns.
- A set of flags to specify other actions (see Lab 2).
- Hint: declare `act` as `struct sigaction act = { 0 };`
- Otherwise, since `act` usually is a stack variable, it will initially contain garbage which, unless properly cleared, is likely to annoy the kernel — and the kernel to annoy you.

More Signal Macros and Functions

```
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum);
```

- The macro `sigemptyset` clears a set of signals.
- The macro `sigfillset` adds all signals to a set.
- The macros `sigaddset` and `sigdelset` modify a set.
- The macro `sigismember` can then be used to see if a specific signal is pending:
  - returns 0 if `signum` is valid but not a member of `set`.
  - returns 1 if `signum` is valid and a member of `set`.
  - returns -1 otherwise — useful in Lab 2.
- The above are macros for manipulating signal sets (`sigset_t`).
- The system call `sigpending` fills in a set of pending signals.

Interprocess Communication

- Shared memory
- Issues in message passing
- Sockets
- Remote Procedure Calls (RPC)

Shared Memory

- This is the most convenient and efficient method of communication.
- It is useful for either multiprocessors or uniprocessors, and
- either for multithreaded programs or for separate processes on the same machine.
- With shared memory, communication is through writing and reading from memory, and this must be protected using synchronisation such as locks.
- With a multithreaded program, no special action has to be taken by the programmer — the memory is shared from the start.
- With separate processes, the shared memory must be attached to the program eg using the UNIX `shmat` system call (shared memory attach).

Issues in Message Passing

- **Direct vs indirect communication**: send a message to a process or to a mailbox (the latter is more convenient),
- **Blocking vs nonblocking**: if both send and receive are blocking we have the equivalent of the Ada **rendezvous** primitive, and
- **Buffering:**
  - **Zero-capacity**: forces the sender to block
  - **Bounded-capacity**: forces the sender to block if queue is full
  - **Unbounded capacity**: the sender never blocks.

Message System Example 1: Mach (MacOS X)

- Mach was developed at Carnegie-Mellon University in a project on a microkernel-based UNIX for multiprocessors.
- A mailbox is called a **port** in Mach and has exactly one receiver.
- The receive rights of a port can be given away to another process.
- The contents of a message is a fixed header with a reply port, and **typed data** (ie, not just a stream of bytes).
- The sender can send port rights by referring to a process-local port number, and then the kernel translates this port number to another process-local port number in the receiver.

Message System Example 2: Windows

- Two types of ports: **connection ports** and **communication ports**.
- Connection ports are known to all processes and are used to initiate communication with a subsystem by creating two communication ports, one for the client and another for the server.
- The maximum size of the data to be sent must be known before starting the communication: short messages are sent by copying the data into the message while long messages instead send a pointer to data to avoid the copying.

Sockets

- A pair of **sockets** are used to communicate **a stream of bytes** between two processes.
- A socket is identified by an IP address and a port number (this port number is local to the machine, as opposed to Mach-process port numbers mentioned before).
- Port numbers $\leq$ 1024 are regarded as **well-known** eg: ssh is 22 and http is 80.

More structured than sockets: RPC

- The main problem with sockets is that there is no type system.
- Data is just a stream of bytes, but representing eg a `short int` in binary is not portable!
  - `short int s = 0x1234;` Assume `s` is 16-bits, or two bytes.
  - Think now of `s` as an array of two bytes: `char s[2];`
  - Which element contains 0x12 and which contains 0x34?
  - On a big-endian machine eg Power `s[2] = { 0x12, 0x34 }`.
  - On a little-endian machine eg X86: `s[2] = { 0x34, 0x12 }`.
  - A common representation is required, eg the **External Data Representation**, **XDR** from Sun (in which they defined data should be transmitted as big-endian).
- Sockets are messy to program with — normal function calls are easier.

Remote Procedure Calls

The goal is to hide the IPC from the source code and let programmers think of normal procedure calls.

Some issues:

- Data representation (eg endian-ness and structure layout).
- Finding the port of the server (eg port 111 is a **portmapper** which is asked)
- Complicated data structures: pointers and eg graphs are difficult to support