Contents of Lecture 8 on UNIX History

- Before UNIX
- Origin of UNIX
- Commerical wars and other threats
- Linux
- Game over. UNIX won.
- Continued work by the original UNIX team: Plan 9 and Inferno
- Microkernels vs Monolithic kernels and other issues.

CTSS

- The Compatible Time-Sharing System was designed at MIT in the early 1960s.
- Supported up to 32 simultanously logged in users: proof of the time-sharing idea.
- Ran on IBM 7090 with 32K 36 bit words, of which the monitor used 5K.
- User programs were swapped in from a magnetic drum (disk).
- Multilevel feedback scheduling queue. Higher priorities have shorter time quantum
- Initial priority (quantum) reflects size of the executable file so it can be swapped in and run.
- CTSS was extremely successful and used as late as 1972: basis for MULTICS.

MULTICS

- Multiplexed Information and Computing Ser vice designed by MIT, AT&T, and GE.
- AT&T and GE provided telephone and electricity ser vices as utilities.
- GE also produced computers, eg the GE-645 used in MULTICS development.
- Why not do the same with computing power? The goal was one computer for 100,000s users in Boston (on hardware equivalent to about 386 or 486).
- Designed as an extension to CTSS with virtual memory and sophisticated protection (more levels than UNIX user vs superuser).
- MULTICS was written in PL/1 and consisted of about 300,000 lines.

End of MULTICS

- Lots of people were very enthusiastic about MULTICS, including the newly graduated Ken Thompson from Berkeley.
- MULTICS never worked very well. It was too complex, the compiler was buggy, and the hardware was too slow.
- The first place it started to work was at Bell Labs (then part of AT&T), but
- ”three persons could overload it” according to Doug McIlroy.
- It had cost AT&T a huge amount of money but produced no real results.
- AT&T dropped out in March 1969: ”a traumatic change” for some of the staff at Bell Labs: ”the toy was gone”.

Space Travel

- After AT&T had withdrawn from MULTICS in March 1969, Ken Thompson continued to work on a Space Travel simulation game but the GE-645 machine he used had no good OS.
- Thompson wanted to continue working on the game and needed a file system.
- At a meeting with Ken Thompson, Dennis Ritchie, and Rudd Canaday, the UNIX file system was discussed and designed. The inode term was introduced and other things.
- Ken had found an unused PDP-7 and used the GE machine to write a rudimentary kernel and assembler for it (paper tapes output by GE and input by PDP-7).

Birth of UNIX

- Ken could allocate one week for each of the
    - the UNIX kernel,
    - the editor,
    - the shell, and
    - the assembler.
- This was during the summer of 1969 when his wife and one-year old son was with his parents in California for a month.
- This was a *real* one-month of work.
- If you haven't yet done so, please read *The Mythical Man-Month* by Fred Brooks.
- Ritchie then also contributed and the origin of UNIX was created.
- Thompson and Ritchie worked with this new toy for about a year and then others joined them.

The PDP-7

- The PDP-7 had 4K words of memory for user programs.
- A PL/1 compiler system would have been too large.
- Ken Thompson decided that a serious system must have a FORTRAN compiler.
- Ken sat down and worked on it for one day before realising he didn't want to write a FORTRAN compiler and instead came up with the B programming language.
- B was a stripped down version of BCPL (Basic Combined Programming Language) from 1967.
- B was an interpreted language.

The B Programming Language

- Only one type of object: the machine word.
- A pointer simply contains the address of an object which also is a number, ie which word the object is in.
- To get the address of a variable, the lv operator is used (as in BCPL): lv x gives the address of x. This was later abbreviated to & in C.
- B (or actually, a B program) was used until 1989 in a type-setting system.

The NB Programming Language

- When some software changed behaviour, Bell Labs usually put an N is put in front of it to indicate this is new.
- Problems with B led to the language NB:
- Pointers referred to word addresses but new machines were byte addressed,
- There were pointers and arrays but no structs in B.
- After a short time, NB was changed to C.

A New Machine: the PDP-11

- The PDP-7 was too small but a PDP-11 cost around USD 100,000 which nobody wanted to spend.
- So, a need for AT&T was identified and a new proposal written: a text processing system.
- The request was granted.
- The UNIX team came up with the troff/nroff typesetting system, which still is in widespread use (eg UNIX manual pages are processed with nroff)

UNIX and the C Programming Language

- Initially system programs and applications were written in C.
- In 1973 the UNIX kernel itself was rewritten in C, which was the first time ever an operating system successfully was written in a high-level language.
- The C language was then implemented on other platforms eg an IBM OS.
- Some applications turned out to rely too much on UNIX and to run these machines, and in 1977 it was decided to port the UNIX system itself to many other machines.
- The base for this project was the Portable C Compiler written by Steve Johnson (most early commercial C compilers were based on pcc), and it was this project which enabled the widespread use of UNIX and C.

Monopoly and AT&T

- Monopolies are legal but a company which has a monopoly in one industrial sector (eg telephony) is strictly forbidden to exploit one monopoly to invade another industrial sector (eg computers).
- AT&T had a monopoly in telephony and was not allowed to sell computers.
- This turned out to be extremely good for us.
- Since they could not sell computers (or software) they gave away UNIX to universities for a nominal fee (essentially only the distribution cost).
- Among other universities Berkeley installed UNIX as early as 1973.

Berkeley UNIX

- Berkeley received funding for doing research on UNIX.
- Bill Joy and others implemented virtual memory, demand paging, and page replacement in 1979 for 3BSD.
- The success of 3BSD convinced DARPA (Defense Advanced Research Project Agency) to fund the development of 4BSD to support TCP/IP (TCP was outlined by Kahn and Cerf in 1974).
- BSD UNIX was very succesful and many companies including Sun Microsystems (co-founded by Bill Joy) used it for their workstations.

Split of AT&T

- AT&T was split into several ”small” companies and was now allowed to sell UNIX.
- The main commercial version was called UNIX System V.
- Some companies preferred BSD UNIX while others preferred AT&T UNIX.
- AT&T had repeatedly failed to sell computer hardware and instead bought a huge amount of Sun stocks. Sun and AT&T joined forces to unify BSD UNIX and AT&T UNIX in UNIX System V Release 4.
- Several companies including IBM, HP and DEC were worried that Sun will take advantage of its collaboration with AT&T
- War begins.

Mach

- Rochester Intelligent Gateway 1975 is a microkernel project
- One of the designers, Rashid, moves to Carnegie Mellon U.
- Rashid builds Accent, ready 1981 but UNIX becomes more popular than Accent
- Rashid starts to design Mach as a UNIX compatible microkernel based OS
- DARPA wants a multiprocessor OS and funds Mach

More about Mach

- Mach and 4.2BSD are combined into one kernel to guarantee binary compatibility
- First multiprocessor version ready 1986
- Open Software Foundation (HP, DEC, IBM, ...) propose Mach as an alternative to UNIX System V Release 4
- BSD moved out from the kernel to a server process: Mach 3.0

New start for UNIX

- The war between AT&T and OSF initally only benefitted one person: Bill Gates.
- Prof Tanenbaum made an educational OS called Minix to teach UNIX.
- Linus Torvalds used Minix when he started to write Linux.
- Linus released the first version in 1991 and after a while Linux implemented the UNIX System V Release 4 specification
- IBM has put huge resources into developing and promoting Linux, especially on Power.
- Solaris 10 runs Linux executables natively.

Plan 9

UNSPEAKABLE HORRORS FROM OUTER SPACE PARALYZE THE LIVING AND RESURRECT THE DEAD!
PLAN 9 FROM OUTER SPACE
BELA LUGOSI
VAMPIRA
LYLE TALBOT

Plan 9 Background

- Plan 9 is the continuation of the work by the original UNIX team.
- Work on Plan 9 started in the late 1980's, and Plan 9 was put in production use in 1989 at Bell Labs (by its developers).
- The original UNIX was a time-sharing OS for many logged-in users.
- Among its advantages include centralized administration and file systems.
- With personal networked UNIX workstations and distributed file systems some of this was lost.
- With NFS the client machines can mount exported directories and have roughly a normal UNIX file hierarchy, resulting in "building a system out of little UNIXes".
- The Plan 9 goal was to "build a UNIX out of little systems".
- Forget about X11-servers, ssh, scp, and the wrong architecture of executable files..., at least when working within the LAN.

Plan 9 Organization

- Plan 9 distinguishes between terminals and servers.
- A terminal is a graphics workstation and a server is typically a high-end multiprocessor.
- All resources are described as files.
- A low-level protocol called 9P is used to access the resources.
- For instance, when a user types `/bin/date`, which particular executable file is fetched depends on the architecture of the terminal.
- In NFS the machine client must mount the proper directory.
- In Plan 9 you can open some windows at a terminal, run commands on different machines, and the resources seen in each window are the same for a particular user — but other users can see a different environment.

C Programming on Plan 9

- The `#if` preprocessor directive has been removed.
- Use of `#ifdef` is discouraged.
- Their ANSI C dialect supports Unicode (ie before C99).
- Instead of 16-bit characters, non-ASCII characters are represented as a byte stream.
- This byte stream representation is UTF-8 which was invented for Plan 9 by Ken Thompson.
- There is an `rfork` system call with which the Linux `clone` system call is similar to, ie to specify which resources should be shared between parent and child.

Design of Plan 9

- UNIX represents devices as a file name — some other things are outside the file system such as sockets. Special `ioctl` commands control some hardware.
- In Plan 9 everything is regarded as a file, including network protocols, devices, computers, and users.
- An extension of the UNIX directory in Plan 9 is the **union directory**.
- The union directory and the 9P protocol allow a remote application to access the local mouse or console — `/dev/cons` is mapped to the local console, ...
- ... or, a device, eg a printer `/dev/lp` on a remote machine can be accessed as a local device in the `/dev` directory.
- Plan 9 is a refinement of UNIX for the network age — but similar functionality can be provided for the user in normal UNIX anyway.

Union Directories

- A union directory allows, essentially, multiple directories to be ”mounted” on the same ”mount point” and the directory then contains the union of the contents of each such directory.
- In addition to `mount` and `unmount`, `bind` adds a subtree of a mounted file system to a union directory.
- Although called a union, the contents is ordered when searched.
- This way the `PATH` environment variable is no longer needed, since you only need one bin-directory.

Inferno

Inferno

- The Inferno Operating System is the successor to Plan 9.
- Inferno is a commercial operating system for distributed systems.
- Portability of applications is ensured by using a virtual machine (Dis).
- Dis was introduced at about the same time as the JVM but is not a stack machine but rather a register machine (ie much better).
- The main difference with Plan 9 is that the Inferno kernel contains both the virtual machine and a just-in-time compiler.
- It has been ported to ARM, MIPS, PA-RISC, PowerPC, SPARC, and X86.
- It can be run as an application on UNIX or Windows or as a real OS.

A new page table for 64-bit address spaces

- Paper by Madhusudan Talluri, Mark D. Hill, and Yousef A. Khalidi
- University of Wisconsin and Sun

We have seen different page table designs

- A **hashed page table** is simply a hash table with of virtual page numbers — disadvantage of space overhead (PowerPC)
- A **linear page table** is an array in virtual memory (VAX/VMS)
- A **forward mapped page table** ie a multi-level page table with physical addresses and stored in RAM (Used by many CPUs and OSes)

Forward mapped page tables

- At a TLB miss for the forward mapped page table, all levels must be used to find the translation, starting with the top-level and using some bits from the virtual page number for each level
- A forward mapped page table thus traverses the tree **top-down**
- A TLB-miss can easily lead to seven memory reads to find the translation
- For a 64-bit architecture the number of levels make this impractical

Linear page tables

- Since the linear page tables are stored in virtual memory, we can be lucky
- We got a TLB miss for page P_1
- We need to read **linear_page_table[P_1].inmemory** and **linear_page_table[P_1].page**
- Now comes the chance/risk: if the TLB has a translation for the virtual page P_2 with our page table entry, then we fetch it and find the translation for P_1
- If there is a new TLB fault we must process that first
- Thus, we access the ”levels” bottom-up
- This design is acceptable for 64-bit architectures

Effects of sparseness of address space

- Forward mapped are a poor option for 64-bit archtectures
- Linear page tables are good at dense address spaces since then it's more likely we find the translation of the translation in the TLB
- For sparse address spaces the linear page table performs poorly since there will be many nested TLB faults
- Hashed page tables performs the same regardless of whether the address space is sparse or dense but have a significant storage overhead

Clustered Page Tables

- A **clustered page table** is like a hashed page table
- The difference is that instead of mapping one virtual page, each entry maps eg 16 contiguous pages
- For sparse address spaces, this performs much better than linear page tables
- If the address space has many ”random” blocks of data which consist of significantly fewer than 16 pages (eg 1 or 2) then clustered page tables are inefficient

The performance of μ-kernel-based systems

- Paper from 1997 by Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter
- Dresden University of Technology and IBM T.J. Watson Research Center

Microkernels

- A microkernel should provide address spaces, threads, and IPC
- Previous research using Mach has showed that microkernels are inherently slower and in practise significantly slower
- Liedtke realized that the IPC implementation in Mach is the main reason for its poor performance
- The IPC implementation cannot be machine-independent but must be targeted and tuned for a particular architecture
- The paper compares a monolitic Linux, L4 with a Linux server, and Mach with a Linux server (MkLinux)

L4 Linux

- Linux runs as a user level server and all device drivers and APIs could be reused without any change
- L4 Linux is completely compatible with the monolitic kernel (2.0)
- The system call mechanism used depends on how an application was linked
- For libc.so the system call results in IPC to the Linux server
- For libc.a the system call instruction is emulated using a trampoline (a way for a kernel to call user level code by changing the return address) which is somewhat slower

Performance numbers of null system call

System	Time	Cycles
Linux	1.68 μs	223
L4 Linux	3.95 μs	526
L4 Linux (trampoline)	5.66 μs	753
MkLinux in-kernel	15.41 μs	2050
MkLinux user	110.60 μs	14710

- It does help Mach to move the server to the kernel but it's still slow.
- Depending on application behaviour, L4 Linux is 5-10% slower than a monolitic kernel (N.B. application performance — not null system call performance).
- Next paper will compare Mach and a monolitic kernel

The impact of OS structure on memory system performance

- Paper from 1993 by J. Bradley Chen (Carnegie-Mellon University) and Brian N. Bershad (Univesity of Washington)
- Mach comes from Carnegie-Mellon University

Comparison between Ultrix and Mach

- Ultrix is a Digital Equipment's version of BSD UNIX and is monolitic
- Mach has a BSD server
- The study measures the cache behaviour of user memory and kernel memory

Performance numbers

Program	**Mach time**	**Ultrix time**
sed	0.58	0.57
egrep	2.01	1.90
yacc	1.75	1.82
gcc	3.70	4.20
compress	1.32	1.32
ab	112.18	98.96
espresso	6.23	6.46
lisp	56.46	54.97
eqntott	66.05	65.85
fpppp	25.20	16.78
doduc	22.94	24.56
liv	1.24	1.22
tomcatv	139.42	155.44

Findings

- System code has less locality than system data and user code/data
- Mach has less locality than Ultrix
- Mach suffers more from write buffer stalls than Ultrix and user
- Page mapping can (independent of OS) affect user locality since one choice can lead to unnecessary conflict misses

Development of UNIX on multiprocessors

- Synchronisation
- Master-slave kernel
- Spin-locked kernels

Synchronisation in the UNIX kernel on a uniprocessor 1(3)

- Short term mutual exclusion: problem avoided with a non-preemptive kernel
- Interrupt handler mutual exclusion: increase the processor interrupt level to block interrupts

```
s = splhi(); /* set priority level to high. */
counter += 1; /* access shared resource. */
splx(s); /* restore previous level. */
```

Synchronisation in the UNIX kernel on a uniprocessor 1(3)

- Long-term mutual exclusion: eg for UNIX semantics when writing to a file (the system call is non-preemptible but may need several I/O operations)
- An ordinary variable can be used as a lock:

```
void lock(int* flag)
{
        while (*flag == LOCKED)
                sleep(flag);
        *flag = LOCKED;
}
```

Synchronisation in the UNIX kernel on a uniprocessor 1(3)

- Sleep() puts the process in the BLOCKED state and wakeup() makes RUNNABLE all (which is unnecessary) processes which slept on that lock.

```
void unlock(int *flag)
{
        *flag = UNLOCKED;
        wakeup(flag);
}
```

Problems with a uniprocessor kernel on a multiprocessor

- The primary problem is that short-term mutual exclusion does not come ”automatically”: there can be several CPUs in the kernel.
- Unless all processors can execute in the kernel, we will soon suffer performance problems
- It is not enough to turn off interrupts on one CPU.
- The implementation of the long-term mutual exclusion does not work either — why?

Master-Slave kernels

- This is the easiest way to use a multiprocessor with a uniprocessor kernel.
- One processor is called the master and only it may execute kernel code.
- Slave processors only execute user code
- Process switch from slave to kernel occurs at a system call or exception
- Implementation: two run queues: master queue and slave queue
- dequeue() removes the process with the highest priority
- enqueue(queue, proc) puts a process in a queue

Buggy enqueue(queue, proc)

-
  ```
  void enqueue(queue_t *queue, proc_t *proc)
  {
          lock(&queue->flag);
          put process in the queue
          unlock(&queue->flag);
  }
  ```
- What will happen if the master has taken queue->flag and an interrupt handler calls enqueue()?

Correct enqueue(queue, proc)

-
  ```
  void enqueue(queue_t *queue, proc_t *proc)
  {
          int prev; /* previous priority level. */
          prev = splhi();
          lock(&queue->flag);
          put process in the queue
          unlock(&queue->flag);
          splx(prev);
  }
  ```
- What happens if a slave has taken `queue->flag` and an interrupt handler at the master calls enqueue()?

Performance of master-slave kernels

- With compute-bound processes, master-slave kernels are quite OK
- With many file accesses or page faults etc (kernel activities) the master gets overloaded
- Some system calls can be executed by the slaves, eg getpid, time etc but since they are simple they would not take much time on the master anyway.

Spin Locked Kernels

- The step after master-slave kernels is to enable some parallelism using spin locks
- Long-term mutual exclusion can still be done using sleep/wakeup (instead of with semaphores and monitors) but see below how to do this properly
- Design alternatives: varying spin lock granularity (how much to lock, ie which data structures)
- A giant lock can lock the entire kernel.
- So, `lock(&kernel_lock)` is called both at system calls and by interrupt handlers
- Can you see any pros/cons with master-slave or giant-lock kernel?

Coarse-grained locks

- A spin lock for each kernel subsystem: process table, files, virtual memory
- System calls can execute in parallel if they use different subsystems
- But how can we avoid deadlocks now?
- Before sleep() the spin locks must first be released and the subsystem must be in a consistent state.

Long-term mutual exclusion: correct sleep/wakeup

- Since sleep/wakeup uses normal variables (and not mutexes) to access eg an i node, we can easily get race conditions on a multiprocessor
- Long-term solution: rewrite the kernel and use normal synchronisation primitives such as mutexes, semaphores, monitors, and condition variables.
- Short term solution: introduce a spin lock to protect modifications of the old synchronization method

sleep/wakeup continued

```
void lock_object(int *flag)
{
        lock(&object_locking);
        while (*flag)
                sleep(flag);
        *flag = 1;
        unlock(&object_locking);
}
```

sleep/wakeup continued

```
void unlock_object(int *flag)
{
        lock(&object_locking);
        *flag = 0;
        wakeup(flag);
        unlock(&object_locking);
}
```

DEC OSF/1 Version 3.0 SMP implementation

- Synchronisation primitives
- Kernel preemption
- Example synchronisations
- Data structure improvements
- Lock package
- Scheduler

Product goals

- High performance for file servers, databases etc
- Same kernel binary for both uniprocessors and multiprocessors
- Lock package which can detect bugs
- Improved parallelism

Synchronisation primitives

- Interrupt levels
- Simple lock: spin lock
- Complex locks: blocking read/write locks with sleep-queue
- Funnelling: execution on a master processor

Real-time kernel preemption

- A kernel thread may be preempted if the following conditions are satisfied:
  1. Interrupt priority level (SPL) is zero
  2. The thread owns no spin locks
  3. The thread does not execute funneled on the master
- Example uses of different synchronisation primitives
  - Timer queue is protected by a spin lock and the disabling of interrupts
  - A directory structures is protected with a complex lock
  - CD-ROM file system is handled only by the master processor since it’s slow anyway

Performance improvements of processes

- Initially a statically allocated proc[] array
- Two lists: allproc and zomproc (all processes and zombie processes)
- Initially protected by funneling it gave poor performance
- New structure: dynamically allocated proc structs and a hash table instead of the lists and the array
- A spin lock in each proc gives good parallelism
- No funneling

Access to the current thread state

- `U_ADDRESS[]` indexed by CPU number
- `U_ADDRESS[]` points to the thread's uarea and is set at a context switch
- Problem: To read the CPU number (on an Alpha machine) cost > 20 clock cycles and is too expensive
- False sharing when different CPUs write to the array.
- Solution: put all thread state in the beginning of the thread stack and use the stack pointer and a mask to find it.

CPU Scheduling

- Soft-affinity (SA) scheduling
- Periodic load-balancing (LB): CPU load average = average local run queue length
- A thread may be moved only if it has waited long enough (since then its cache state has disappeared anyway) Performance on a four-processor DEC 7000 SMP

# job	time with SA/LB	time without SA/LB
1	25.9 s	26.0 s
4	25.9 s	26.0 s
16	106.5 s	141.9 s

- A small change in the scheduler can make a big difference!

Game Over UNIX Won

- With Google's Android the future of UNIX has never been brighter, ever.
- Android is a software stack with a modified Linux kernel.
- This is not an ad, but, with an Android "phone", you buy a UNIX computer that you can hold in your hand and which lets you make phone calls as well.
- You hold approximately 40 years of UNIX history and 20 years of Linux history in your hand.

Background

- In 2005 Google purchased the company Android Inc which made software for mobile phones.
- In 2007 the Open Handset Allicance presented their goal of developing software for mobile devices based on open standards and the Linux 2.6 kernel — a relatively new Sony phone (Xperia V) uses Linux 3.4
- In 2008 Google open sourced the entire software under the Apache License
- Apache License means companies can extend the sources without submitting back the new code.
- Android does not support Java ME but Google's Java libraries
- Android SDK runs on any modern Linux/X86 machine, MacOS X or Windows.

Booting Linux

- After the Power-On-Self-Test the Master Boot Record (ie the first sector) of the boot device is read and the boot loader, e.g. GRUB, on the active partition is fetched which fetches the kernel.
- The startup code of the kernel is written in assembler and performs:
  - identifies the CPU type
  - disables interrupts
  - enables the MMU (memory management unit)
  - calls the **main** C function of the kernel
- **main** continues the initialization and allocates data structures and probes devices

The First Processes

- After initialization the kernel ”manually” creates the first process (manually since it cannot create a process using fork yet)
- Process 0 then mounts the root file system and programs the real time clock
- It then forks to create the **init** process and the **page daemon**
- The init process either enters single-user mode by forking a shell or multiuser, and forks the program **getty** for each attached terminal
- Getty makes initializations and then prints **login:**. After somebody has written a user name it execs the **login** program which reads and checks the password
- If the password is correct login execs the shell specified in **/etc/passwd** for the user

Dynamically Loading Device Drivers

- Linux, and many other OSes including MS-DOS, can load device drivers dynamically while the kernel is running
- This is convenient but unpleasant in security sensitive environments such as banks
- At a bank which uses dynamically loaded device drivers, a system administrator might be able to insert a module which accidently (or intentionally) corrupts the system.

Linux Kernel Modules

- To avoid enforcing static linking of all device drivers, Linux can dynamically load kernel modules
- While usually used for device drivers, they can be used for any kernel extension such as a performance measurement tool
- To insert a module the command **insmod** is used
- The first step to do is to **link-edit** the module: the addresses of the symbols used by the module from the rest of the kernel must be set to the actual values (for the module, they are just undefined symbols before the module is inserted)

More about Inserting Kernel Modules

- After link-editing, the resources needed by the module must be examined to check that they are available (including interrupt request level)
- Interrupt vectors are set up to map the interrupt to code in the module
- A device driver table entry for a new major device number is set up to refer to code in the module
- Then the module is invoked to let it initialize itself
- After this, the module is a part of the running Linux kernel
- Other modern versions of UNIX have similar features