AGH

AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Introduction to Computer Science Part 5

Version: 2023

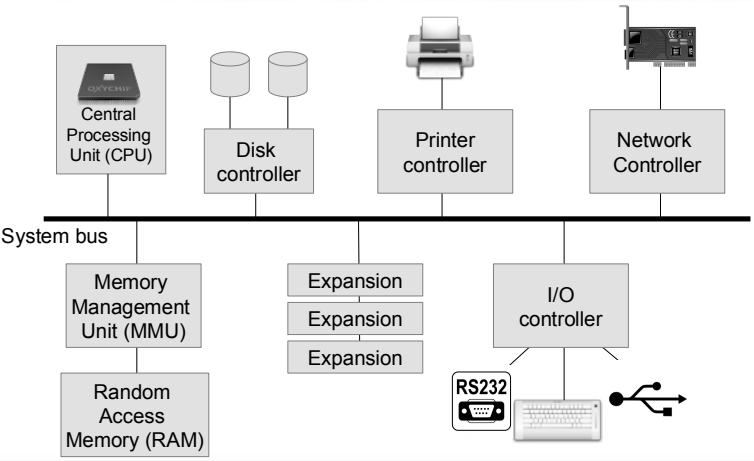
Marek Wilkus Ph. D. **http://home.agh.edu.pl/~mwilkus**
Faculty of Metallurgy and Industrial Computer Science
AGH UST Kraków

AGH

Back to the computer hardware

AGH

Computer system hardware basics



AGH

What is a system bus?

- A set of lines responsible of transferring information from/to the CPU.
  - Address bus - A set of address lines to select the proper place (address) in memory map.
  - Data bus - A set of lines corresponding to bits of word (byte, or multiple of byte), with which the information is sent and obtained.
    - These are **bidirectional** lines.
  - Control lines - lines for notifying CPU about events and setting the cimputer system to the correct state depending on the task performed by the CPU.

AGH

A bus of a PC's processor
(quite old one, but newer are just more complex)

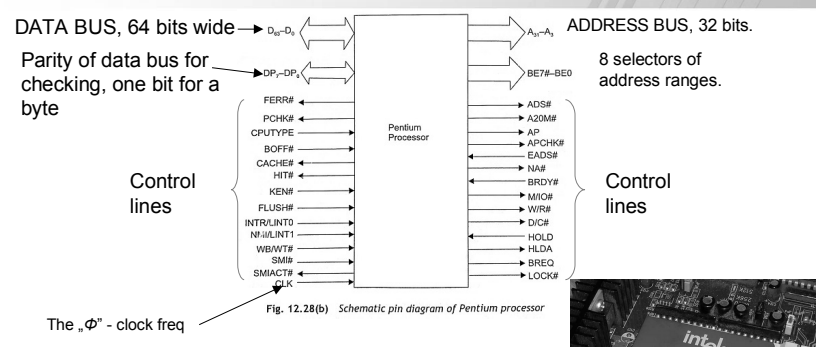
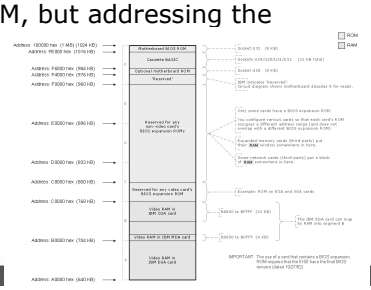


Fig. 12.28(b) *Schematic pin diagram of Pentium processor*

AGH

CPU communicates with outside world

- Every computer has a **memory map** - an alignment of memory locations. There are specific memory locations for:
  - RAM
  - Read-only memory containing the firmware (or BIOS).
  - Different devices' memory, I/O and configuration registers.
- To communicate with a device, CPU writes or reads the information like from RAM or ROM, but addressing the device's address.
- The CPU shown previously allows to „bank" the memory having RAM in one range and devices in another, saving RAM which would be „shadowed" by devices' registers →wasted.



Control bus - important lines

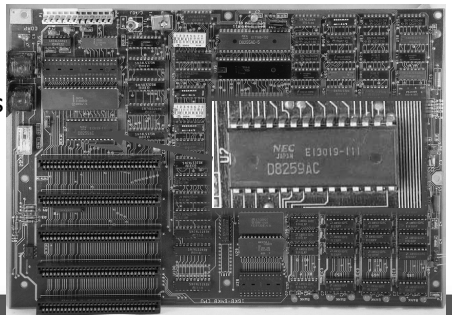
- The clock - main frequency the system runs.
- RESET - puts the CPU back in the operation from the beginning.
- READ/WRITE - as in the name.
- HOLD - The CPU holds off from mastering the bus and devices on the bus may use it to communicate with each other.
- DMA - Holds the MMU from talking with RAM so DMA controller may freely use the RAM.
- INT… - Causes the **interrupt**.

The interrupt

- Presence of the interrupt signal causes the CPU to stop executing currently executed code and jump to the specific interrupt handling code - the **ISR** Interrupt Service Routine).
- After returning from the ISR, CPU has no information that something like this happened.
- Interrupts are used by devices to signal different situations in the system to the CPU.

Interrupts

- **External** interrupts are caused by devices, like:
  - Timer counter overflown,
  - I/O disk unit finished getting/saving the data,
  - Network controller is ready,
  - Sound chip finishes playing the current sample and CPU has to get it the next one.
- Older systems have an **interurpt controller** chip that, when a device signals interrupt, prepares the memory unit to shove the ISR address under instruction pointer and triggers the single CPU's interrupt.



Internal interrupts

- ...are called **exceptions** and are issued by the situations **inside the CPU**. For example:
  - ALU caused a divide by 0.
  - A register has overflown.
  - The opcode CPU tries to execute is not in the instruction set.
  - The memory area program tried to access is not in the program's memory range.
    - ...so it's ime to `Segmentation fault (core dumped)`

Types of internal interrupts

- **Faults** are the interrupts with which we can generally continue running some code as if nothing happened (we can retry current instruction).
  - Division by zero (we will get just no result)
  - Invalid opcode (Intel) - operation will not be executed.
  - We try to do FPU instruction, but there is no FPU (no result)
  - Invalid memory request (segfaults),
  - Memory refers to a page which is not there - time to find it in the „swap"/paging area and restore it somewhere.

Types of exceptions (2)

- **Traps** are triggered by specific instructions (Intel) or with exceptional situations.
  - In some Motorola architectures traps were fired every time an unknown instruction occurred. This has been cleverly hacked in early Macs.
  - Injecting traps to the code allows to stop the code any time and analyze its execution.
  - Breakpoints.
- Continuing the execution from the trap causes always the **next** instruction to execute

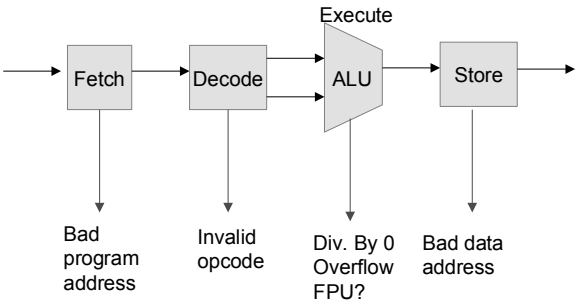
Types of exceptions (3)

- **Aborts** - Unrecoverable errors. We would continue, but we can't be sure even about instruction pointer's location. Further execution may lead to undetermined state, I/O faults etc.
  - Exception during an exception handler call (double fault)
  - CPU registers check failed.
  - Exception during an exception handler call of double fault exception (triple fault).
- We usually display a message and initiate a soft reset in such cases.

The „A-trap" hack

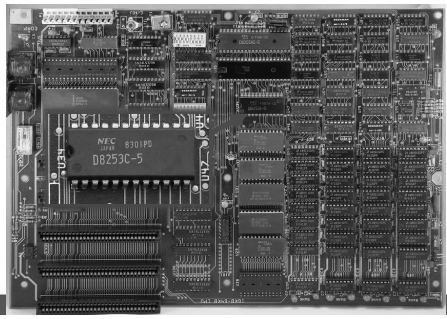
- Early Macintosh had an interesting hack called a „A-trap".
- ROM contained not only the most simple I/O routines (BIOS), but more complex things like mouse support routines, and even drawing windows, controls or buttons on the screen. This was called a „**Toolbox ROM**".
- The illegal instructions starting with 0xA… caused the trap…
- ...simultaneously, the MMU detected the instruction and depending on its next bits pointed the address of the next instruction to the proper location in the Toolbox ROM.
- CPU continued operation, but instead of the instruction it executed code from the Toolbox, then it returned to executing the program.
- This significantly sped up the GUI.

Exceptions generation



Timer interrupts

- Modern operating system **must** have a regular timer interrupt.
  - Switching actively running programs,
  - Controlling the time of processes,
  - I/O devices scheduling…
- In older systems, there was a **PIT** - Programmable Interrupt Timer - a chip which fired interrupts every overflow of the built in counter.
  - The counter was increased with clock pulses.
  - Every interrupt, the counter was re-set to some value.
  - Not a very fast, or a very precise thing, but still useful.



Timer interrupts

- If PIT is not present or unreliable, it is possible to try with **TSC** - Time Stamp Counter - a 64-bit counter counting CPU cycles from the reset. Can be periodically read and fires an interrupt at some value.
  - Not very reliable in multi-core CPUs (if a core is suspended for power saving, its TSC may not count).
  - ...and hibernation/suspending freezes the TSC.
  - Easier to program than HPET - Linux uses it by default.
- A **HPET** - High Performance Event Timer - Produces periodic interrupts with better resolution and reliability. Built also around a 64-bit up counter.
  - More reliable in multimedia applications, harder to program.
  - In some newer CPUs it takes long time to program it back.
  - Not used in the newest Linux versions.

Timer - if all else fails

- Use an RTC which can fire interrupts at a programmed time 2-32768 times per second, not a very precise tool.
- Do we have anything else in the system?
  - Measurement device, known measurement time, interrupt triggered at the end of measure. (some power management devices are used this way)
  - Timers built in other devices (usually sampling converters)
  - ACPI timer - timer in modern power management chip.

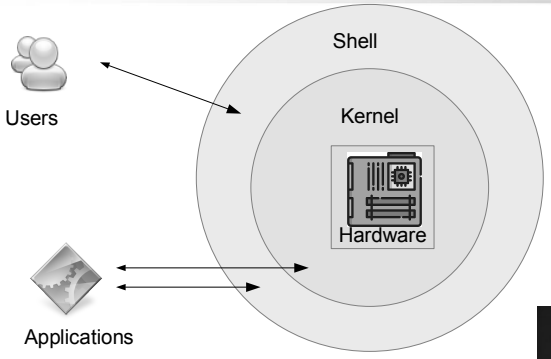
But why do we need it?

- Proper time measurement is a key factor for modern operating systems to work.
- Without a precise timer, multitasking operating systems usually have significant problems.
- So how the operating systems work?

Operating system...

- ...is a program that manages computer resources and works as a bridge between the user (user's programs) and a computer hardware.
- Management of computer's resources includes memory, non-volatile storage, access to devices, but also CPU time.

The 3-layer model



The 3D depiction of this model in „The UNIX operating system - AT&T Archives", 1982

Components of the OS

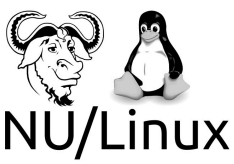
- **Kernel** - communicates directly with hardware, or using it **device drivers**, sometimes stacked in **abstraction layers**. Controls memory, scheduling and resources.
- **Shell** - is an interpreter of system's functions offering them to the user and user's programs.
- **Applications** - executing user's tasks in the system.

Modern OS principles

- The main objective of the OS is to plan the usage of system's resources to **maximize effectiveness.**
- Additionally, the Os should make the computer system **comfortable and efficient to use** and program.
  - To do it, it offers various shells and may offer a compatibility layers to run the same applications in different computer systems.
- Applications and shell do not communicate directly with hardware. They use kernel's **system calls** (or orther abstraction) to do it.

A quick remind about UNIXes

- UNIX is a multi-user, multitasking OS, initially developed as kernel+shell+programs, developed since 1960s at various universities. Available commercially in various **distributions** like Solaris, AIX, HP-UX, Tru64 etc.
- **GNU** is a free-as-in-freedom implementation of UNIX (as **G**NU's **N**ot **U**NIX :)) developed part-by-part when UNIX became commercial.



- **Linux** is a **kernel** of OS, on which e.g. GNU can work.
- **BSD** - is another distribution of UNIX-like OS, based on own kernel.

Distribution

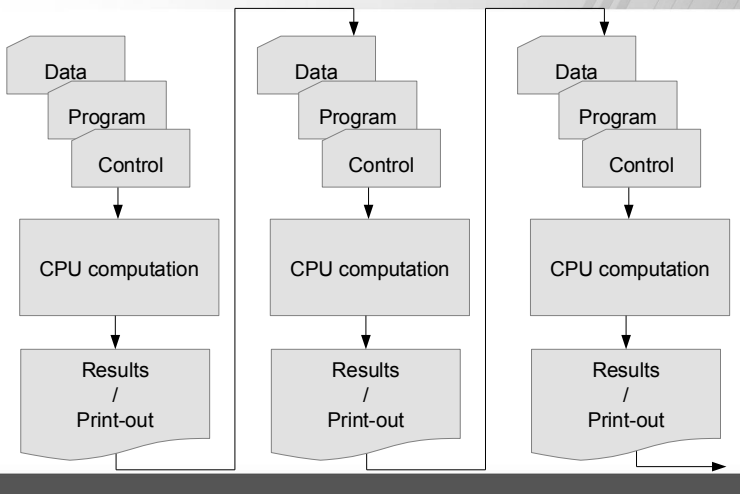
- A kernel, shell and applications configured for specific task, packaged in a single software set, is called a **distribution**.
- There are Linux distributions like Debian, SuSE, Fedora, Arch, Gentoo or Ubuntu (and >100 more).
- BSD distributions are FreeBSD, NetBSD, PC-BSD etc.
- Windows is also packaged in various distributions, starting from the same system with different programs (Win7 Home, Professional, Ultimate, etc.) to versions with differently configured kernels (FLP, Server, Terminal Server).

Is Linux GNU-only?

- There is nothing limiting us from e.g. using Linux kernel with BSD shell/applications. Debian had such project (Debian GNU/kFreeBSD).
- In 1995, IBM put OS/2 shell on Mach kernel, runnin an IBM PC OS on a RISC RS/6000 computer. Was working, was not stable.
- It is possible to run Gnu on Solaris kernel (Nexenta project).
- GNU on Windows? → Cygwin or MinGW!
- GNU kernel, HURD, is in development since 1980s

History of operating systems

Batch OSes



Batch OS

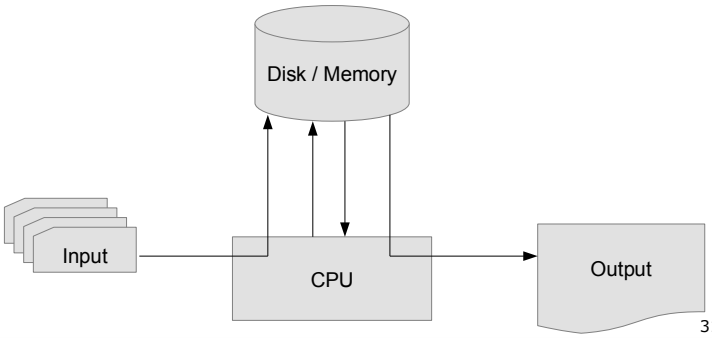
- Tasks executed in order.
- Every task is made of an input data/program read, calculation, and results print-out/saving.
- Next task is started after the previous one ends.
- The order of tasks is dependent on operator.
- A similar requirements tasks are grouped in batches (by operator).
- Rather operation methodology than operating system.

Batch OSes

- Advantages:
  - Simplicity - just executing task after task.
- Disadvantages:
  - No interactive tasks.
  - Only one task is executed in a specific time.
  - During slow I/O, CPU is not doing any productive work. During CPU calculation, I/O devices do not do anything productive too.

SPOOLing

- Simultaneous Peripheral Operation On-Line



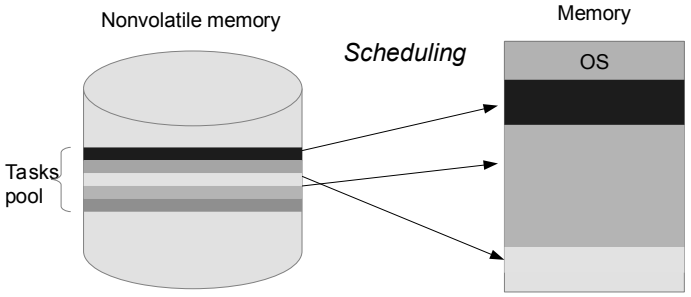
Spooling

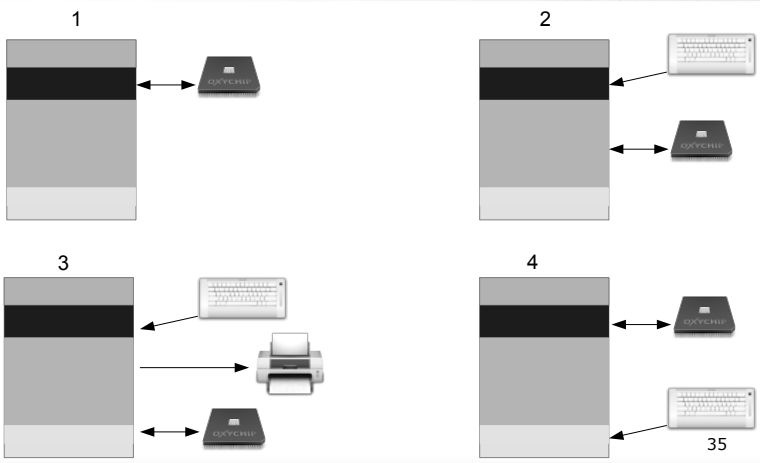
- When CPU is calculating one task, the I/O unit loads another task to the other part of the memory.
- More efficient task throughput by the cost of larger memory.
- With a memory large enough, it is possible to write results of a finished task, run active task and fetch the next task at the same time.

Spooling

- Advantages:
  - More efficient usage of the computer.
  - Smaller number of CPU „stops” for I/O.
  - CPU and I/O work at the same time.
- Disadvantages:
  - More memory required.
  - There are better and worse orders of CPU-intensive and I/O-intensive tasks. Still operator decides.

Multi-program batch system





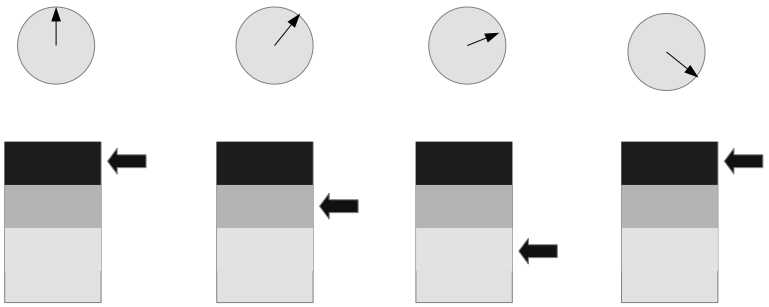
Multi-program batch system

- Multiple tasks are loaded in the memory at the same time.
- Tasks to be done are kept in nonvolatile memory.
- When a currently executed task does an I/O, a CPU can be used for another task.
- When the task finished its I/O, the system returns to the task it paused.

Multi-program batch system

- Advantages:
  - Even more efficient usage of CPU and I/O.
  - Faster execution of a set of programs.
- Disadvantages:
  - Tasks scheduling and CPU assignment algorithms are complex.
  - A computationally heavy task can seriously block an entire system.
  - Requires more efficient hardware.

Time sharing system



Time sharing systems

- CPU is executing multiple tasks, constantly switching between them.
- With a high switching frequency, users do not notice that the programs are multiplexed - it looks like the computer is doing multiple tasks simultaneously and users may interact with multiple programs at the same time.
- With multi-user systems, it looks like every user has own computer.

A popular OS history

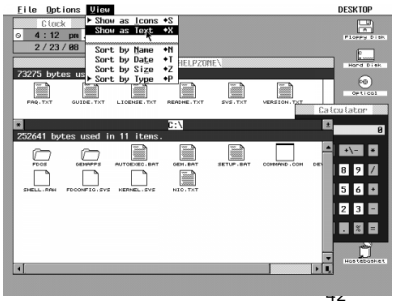
- CP/M (G. Kildall, 197x-8x) - Text-based, single-task, single-user…
  - ...we can pretend that there are many users, but one logged in at a specific time. But there was no protection and the „home directories" were disk partitions…
  - MS-DOS evolver from it.
- MS-DOS
  - 08.1980: Tim Patterson, Seattle Computer Products - a prorotype,
  - 12.1980: QDOS, Microsoft buys non-exclusive license.
  - 1981: Adapted to a new IBM PC computer as MS-DOS 1.0
  - 07.1981: MS buys an exclusive license with all rights for $50 000. Now it's a MS-DOS.

DOS (2)

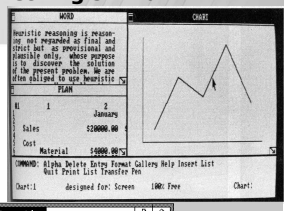
- When MS had non-exclusive license, IBM developed their own DOS…
  - PC-DOS 1.0 i 1.05 - 160kB disks support. COMMAND.COM shell, with a full file operations support (in CP/M there was a „PIP" program needed for file operations).
- 1983: PC-DOS 2.0, directories and simple I/O streams supported, files can be opened and supported by handles, device drivers can be resident. Support for 360kB disks.
- 1984: DOS 3.0 i 3.3 - FAT16, PC-AT, hard disk support (20MB, more with custom drivers), additional tools. 1.2MB and 1.44MB disk support.
- DOS 6.22 - 1994/5 - the last DOS sold as separate product.

DOS - 1990s - end of life

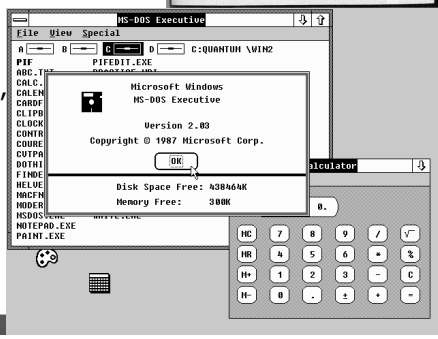
- DOS 7.0 - Part of Windows 9x.
- Caldera DOS, DR-DOS, MUDOS, DOS Plus - alternatives.
- FreeDOS - an open source DOS.
- A pseudo-multiprogram DOS graphical shell called GEM - a non-active program is suspended.
  Open source since mid-2000s.



MS Windows
"Visi-On looks nice, let's make something similar"

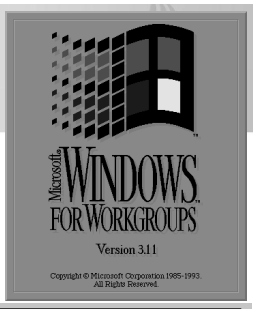


- 1984 - Windows preview - mostly semigraphics.
- 06.1985 - Windows 1.0, 1.01,

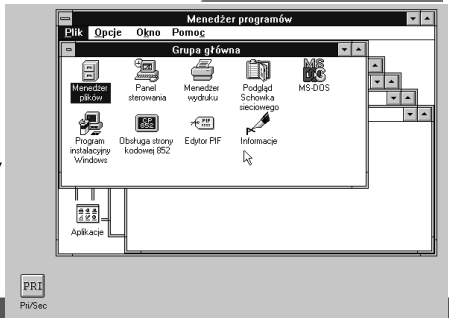


- 1987 - Windows 2.0 and Windows/286 - windows can overlap, multitasking, Windows PE (exe) standard, unified device drivers standard.

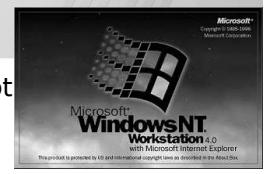
MS Windows



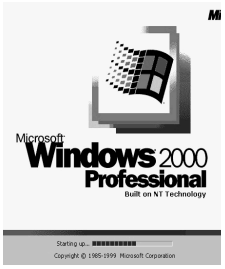
- 1990 - Windows 3.0 - requires >=768kB RAMu. Better GUI and multi-program support. Inter-program interoperativity (OLE, copy-paste in an entire system).
- 1993 - Windows for Workgroups 3.11 - OS-level network support. Support for multimedia, dynamically loaded drivers, localization (3.2 supported composing characters for reading and writing).



Windows - next versions



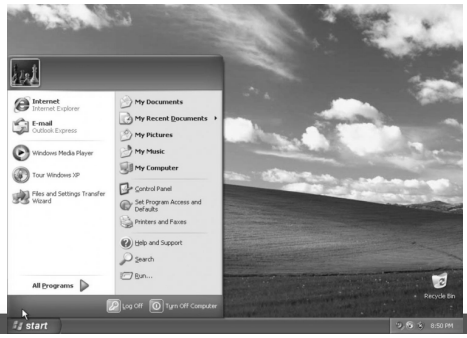
- 1994 - Windows NT - Windows being not an MS-DOS shell, more improvements.
- 1995 - Windows 95 - Even better shell, DOS is still in the background.



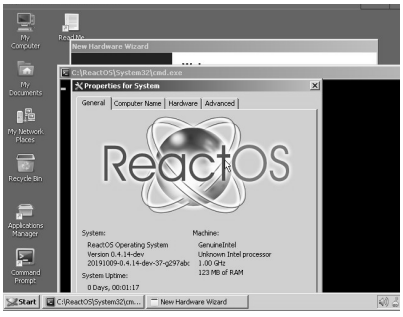
- 1996 - Windows 95 OSR2 - „Give me my Netscape back!".
- 1997 - Windows NT 4.0 - A professional OS for network and workstations.
- 1998 - Windows 98.
- 02.2000 - Windows 2000 - NT 5.0.
- 09.2000 - Windows ME (Millennium) - the last Windows with DOS in it.

Windows (XP era)

- 10.2001 - Windows XP (NT 5.1) - requires 300MHz, 128MB RAM, 1.5GB HDD, one of the longest supported Windows version.
- 2003 - Windows Server 2003.
- 2005 - Windows Vista,
- 2007 - Windows 7,
- 2012 - Windows 8,
- 2014 - Windows Server 2012R2,
- 2015 - Windows 10.

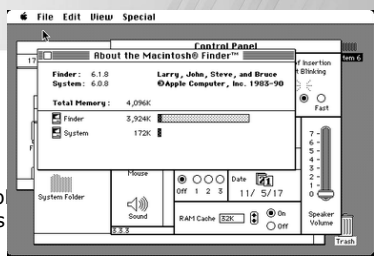


ReactOS

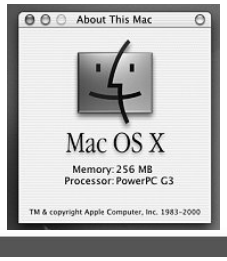


- In Linux, it is possible to run Windows programs using **WINE** abstraction layer which translates Windows system/API calls to Linux/Unix calls.
  - So it looks like an API re-written for a new platform.
- If we could add a process scheduler, virtual memory and drivers support, won't it make an open source Windows?
- The ReactOS project (1996 - FreeWin95) - currently in an early alpha version.

Mac OS

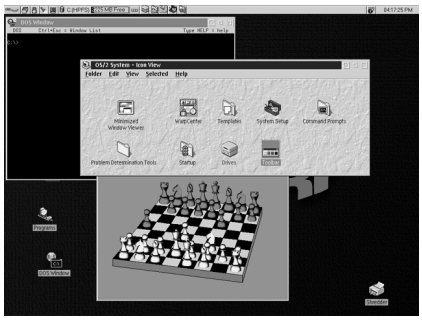


- An operating system for Macintosh 128K computer, called just „System Software", later Mac OS.
- Components:
  - Kernel - monolithic, later with loadab configuration and modules, with calls **toolbox ROM.**
  - Finder - a file manager and graphics shell,
  - Disk Utility - A disk subsystem driver pretending to be just another disk diagnostic tool (delete this and you'll loose disk access :)).
  - Applications: Multiplexed (non-active do not get CPU), some resident running in background in timer interrupts.
- Developed until late 1990s to version 9.2.2
- Later: Mac OS X, based on Berkeley Unix.



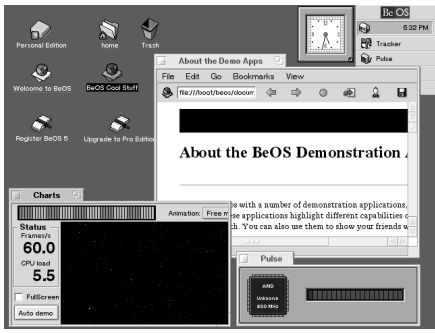
Other OSes: OS/2 (1990s)

- Initially IBM's PC-DOS + „Presentation Manager" GUI shell.
- Later, IBM ??got?? Windows 3.x code and embedded a Windows 3.x API to it.
- Later versions implemented Java at the kernel level making it extremely efficient in running Java software.
- Developed until mid-2000s as eComStation.
- Used in e.g. ATMs until early 2010s.



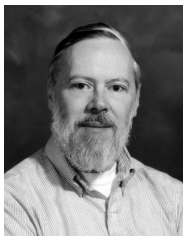
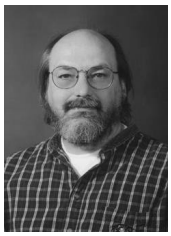
Other Oses: BeOS

- Created by Be Inc. in 1995 for BeBox PowerPC-based mutlimedia PCs.
- Lots of experimental features for multimedia-enabled operating systems.
- CPU scheduler using heuristics.
- Memory assginment dependent on devices used by program.
- Last version: R5 - 2001.
  Later R5 Dan0 (from source leak of version 6) and ZetaOS.



- Open source implementation: Haiku OS (currently beta)

UNIX

Denis Ritchie Ken Thompson Richard Stallman Linus Thorvalds

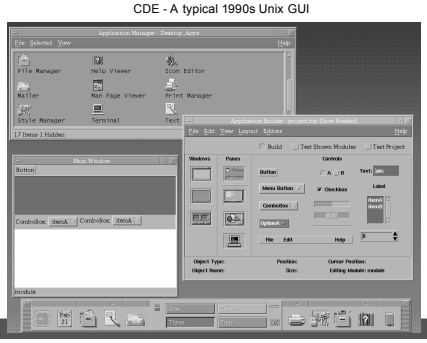
UNIX

- 1966 - Bell Labs, K. Thompson - MULTICS multiplexed operating system.
- 1969 - The first Unix - AT&T/Bell Labs, D. Ritchie, K. Thompson
- 1971 - Unix gets ported to PDP-11 - a computer affordable by an average university in the USA,
- 1973 - Unix 4th edition - entirely in C.
- 1975 - Unix 6th Edition - freely distributed in universitied and developed there. The beginning of verious distributions, including BSD (Berkeley Unix)

UNIX (next)

- 1977 - The first BSD tapes get distributed,
- 1980 - SCO Unix, Microsoft XENIX, Coherent UNIX Preview.
- 1981 - DEMOS - BSD re-written to assembly gets performance boost, but supports only a few computer architectures.
- 1982 - Silicon Graphics IRIX,
- 1983 - **GNU (R. Stallman)**
- 1984 - Hewlett-Prackard HP-UX,
- 1990 - IBM AIX,
- 1991 - Sun Microsystems:
  Solaris 2
- **1991 - Linux 0.01**
  (L. Thorvalds)

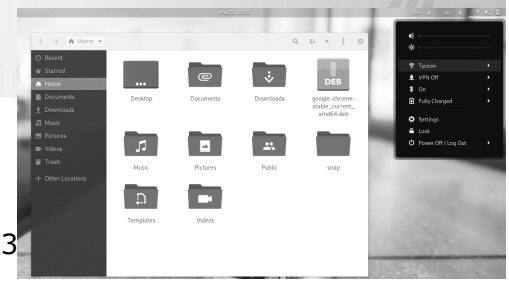
CDE - A typical 1990s Unix GUI



UNIX (the Linux era)

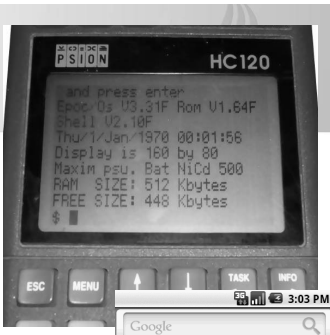
- 1993 - FreeBSD 1.0
- 1993 - **Debian GNU/Linux 0.x,**
- 1994 - Red Hat Linux, Caldera Linux, beginning of various Linux distributions,
- 13.03.1994 - **Linux 1.0**
- 1996 - Linux 2.0
- 1999 - Linux 2.2
- 2001 - Linux 2.4,
- 2002 - KDE 3 GUI for Linux and BSD,
- 2003 - FreeBSD 5.0,
- 2008 - OpenSolaris,
- 2010 - End of OpenSolaris development.
- 2011 - Oracle Solaris 11

Unix/Linux

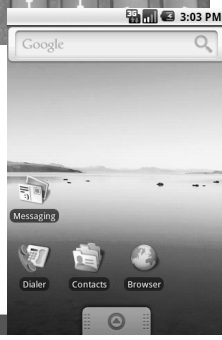


- 2015 - Oracle Solaris 11.3
- 2015 - FreeBSD 10.3
- 05.2016 - Linux 4.6
- 08.2016 - **Android 7.0**
- 08.2017 - Android 8.0
- 2015-17 - Init gets replaced by Systemd. System starts in parallel, logs are binary files and the amount of code to be rewritten is significant.
- 2020 - Linux 5.x,
- 2022 - Linux 6.x

Mobile OS



- 198x - PAL (DOS 3.x + extracodes)
- 199x - EPOC → Symbian
- Android:
  - 1.x (2008) - An OS for digital cameras.
  - 2.x (2009) - A shell in a web browser process.
  - 3.x (2011) - Multi-core and filesystems support, more efficient UI.
  - 4.x (2011)
  - 5.x (2014) - Backporting features from Linux kernel.
  - 2017-23 - PostmarketOS - Desktop Linux runnable on Android devices.



Android 1.6

Mobile OS

- BSD + API kits mechanism (BeOS) + system services fragmentation → iOS (bigger energy saving and performance).
- 2022 - uses more memory-safe code written in Rust. Better security by the cost of performance and memory consumption.
  - Linux is slowly rewritten a similar way.
- 2022 - Still some single-tasking OSes (qronOS, OmnOS, SyMobi) - in a closed-source „feature phone” and control systems.

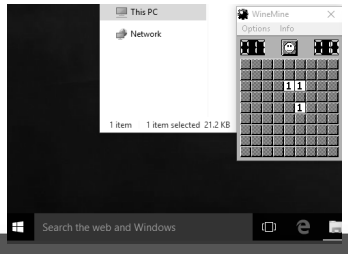
Processes and threads

The program is running

- Programs cannot interface directly with computer hardware.
- To read a file from disk, write to console or reserve memory, they use **system functions**.
- These functions are standardized in **API** - Application Programming Interface.
- In Unix, API is specified by POSIX standard.
- But how to call a function? How to send arguments and get result?
  - This is standardized in **ABI** - Application Binary Interface.

API, ABI

- In Linux, API offered by GLIBC and POSIX ABI is quite stable. Some programs will not work perfectly, and they should be re-compiled.
  - This is not a problem in open source world.
- In Windows, API and ABI **must** be stable. Most programs are distributed in binary form and re-compiling is out of question.
  - So there are thousands of API functions for backwards compatibility.



The Process

- ...is a program running in the system.
- Must be ran in a sequential way - in a moment, one instruction is executed.
  - This instruction is pointed with instruction pointer.
- Consists of one or more **threads** - CPU scheduling units.
  - OS kernel schedules assignment of CPU with specific threads.
  - It is generally not possible to have process without threads.

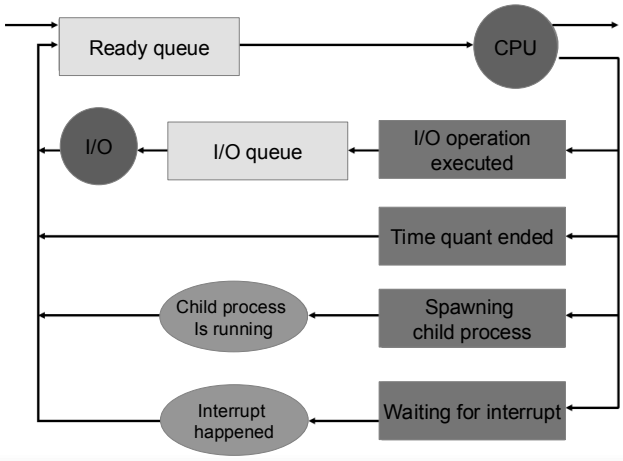
Process control block

- To manage process, OS has a data structure called **control block**. It consists of:
  - Unique Process Identifier - PID,
  - Pointer to text section of the process,
  - Instruction Pointer state,
  - Dump of processor registers,
  - Data/BSS section pointers,
  - Process state (New/Running/Waiting for CPU/Waiting for IO/Finished),
  - Open files, permissions, queues, scheduling info.

Queues used in scheduling

- Job queue - processes started and without any CPU assignments.
- Ready processes queue - Processes which are waiting only for CPU, because they finished I/O, or got preempted and other processes are running.
- Device queues - Proesses waiting for a specific I/O device or resource.

Process states and their queues



Schedulers

- Program which selects processes to ready queue is called a **long-term scheduler** or **job scheduler**.
  - Because processes enter the system relatively rarely, it may not be fast.
- Program which choses the job from the ready queue and sets it running on the CPU is called a **short-term scheduler**.
  - As programs are switched all time, it must be **very** fast.
- Program which suspends processes when they will wait for a long time (e.g. for I/O) and swaps them to disk is called a **medium-term scheduler**.

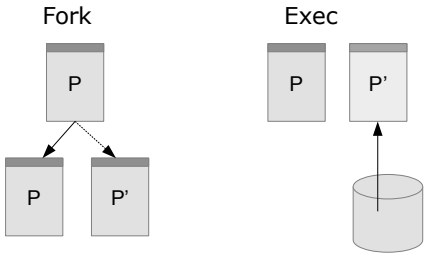
Context switching

- To suspend one process and resume another, the scheduler performs a **context switching**.

1. Update process control block.
2. Update scheduling information.
3. Dump needed memory areas.
4. Save the updated control block to kernel memory.
5. Retrieve proper control block of resumed processes.
6. Verify its integrity (anti-ROP mechanism).
7. Set CPU registers and instruction pointer.
8. Return control to the process.

Creating process

- UNIX: Fork and exec functions:



- Windows: CreateProcess/CreateThread functions family.

Ending process

- Process calls a system function to be ended.
- ...or is **killed** by the system:
  - Overwrite instruction pointer in the control block with the pointer to the ending system function call.
  - Switch the context.
  - Preformed if process hangs, causes some interrupt (segfault), parent process ends, or uses too much memory (OOM killer).

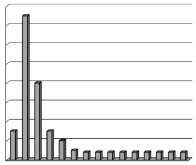
When to switch context?

1. Process executes an I/O operation.
2. An active process became ready (e.g. by an interrupt).
3. A waiting process became ready (e.g. I/O operation completed).
4. Process ends.

- If only the process can ask the system to switch context (1 and 4), it's a **cooperative multitasking**.
- If system can halt running process (2 and 3), it's a **preemptive** multitasking.
- Modern operating system kernels have preemptive scheduling.

Cooperative scheduling

- Much simpler, does not need a clock interrupt.
- Dangerous - will the process give the CPU back to the system?
  - By default, CPU phases frequency vs length is like →
  - But process may hang in a loop.



- Older Windows system used it.
  - ...so GUI programs had to execute „process events" function every now and then.
- Later Windows used it only for drivers.
  - NT almost does not use it.

Preemptive scheduling

- Better for interactive applications.
- More complex, needs better scheduling algorithms.
- But **risky** - what if the halted process is executing a system call? Another process may break its kernel state!
  - We may wait with context switching until the system call ends,
  - We can mask interrupts when system call is executed (what with I/O then?)
  - We can look for consistency of kernel structures and not switch when they're not OK.
- Windows NT, parts of 9x and UNIX use it.

Scheduling algorithms: FCFS (First Come, First Served)

3 processes entering the system:

1. P1	CPU time: 24 ms
2. P2	CPU time: 3 ms
3. P3	CPU time: 3 ms

Gantt diagram for FCFS in this example:

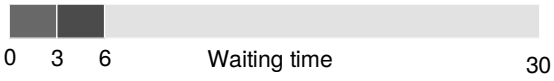
Average waiting time:

(0+24+27)/3 = 17 ms

FCFS scheduling

But what if the order would be different:
1. P2
2. P3
3. P1

Gantt diagram is now:



Average waiting time:

(0+3 +6)/3 = 3 ms

FCFS

- Works with cooperative multitasking.
- Is simple
  - ...and that's the end of advantages.
- Non-optimal.
- Highly dependent on the order of entering processes.
  - In modern systems it's almost non-deterministic!
- The „Convoy effect" - lots of shorter processes wait for one long process.
- Totally unusable in time-sharing systems.

SJF
Shortest Job First

1. P1	CPU time:	6 ms
2. P2	CPU time:	8 ms
3. P3	CPU time:	7 ms
4. P4	CPU time:	3 ms

Gantt diagram for SJF:

Average waiting time:
(3+16+9+0)/4 = 7 ms

For FCFS it would be:
(0+6+14+21)/4 =10,25 ms

SJF

- We can prove that this is an optimal method.
- Putting the short process before long one decreases waiting time for it!
- Frequently **considered** (but rarely used) in long-term scheduling.
- Works with cooperative and pre-emptive methods.
- Problem: This is good only theoretically!
  We cannot predict the length of next CPU phase's time.

SJF - estimate the next CPU phase's time

$$f_{n+1} = \alpha\, t_n + (1-\alpha)\, f_n$$

where:
α – weight (0÷1)
t_n – n-th CPU phase length
f_n - previous function value.
if α =0 - only older values are considered,
if α =1 - we consider only recent values.
Usually α =0,5

Enhancements to SJF

- Shortest Remaining Time First (SRTF)
  - Expansion of SJF for pre-emptive.
- If a process with shorter CPU phase enters the system, we can pre-empt currently running process.
- Context switching requires saving of scheduling/timing information.
- Enhancement: Give priorities to processes.
  - Problem: Process of the small priority may never get a CPU (starvation).
  - Example: In 1973 an IBM 7094 machine was phased out. Operators found a low-priority process which got no CPU since 1967.
  - Solution: Bump the priority for older processes.

Round-Robin (RR)
A scheduling for time-sharing systems.

- Every process gets a quantum of time - a small „time slice”.
- After the slice gets used, the next process gets a new quantum.
- Processes are selected with FCFS.

- So **n** ready processes with quantum **q**, each process waits no move than (n-1)*q of time.

RR, quantum=25ms

1. P1 CPU time: 24 ms
2. P2 CPU time: 3 ms
3. P3 CPU time: 3 ms

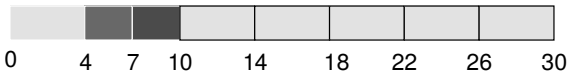
Gantt diagram:

Average waitinig time is:
(0+24+27)/3 = 17 ms
Context switches: 2

RR, quantum=4ms

1. P1 CPU time: 24 ms
2. P2 CPU time: 3 ms
3. P3 CPU time: 3 ms

Gantt diagram:



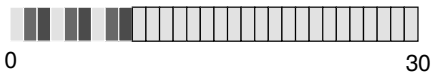
Average waiting time:
(17)/3 = 5,66 ms

Context switches: 7

RR, quantum=1ms

1. P1 CPU time: 24 ms
2. P2 CPU time: 3 ms
3. P3 CPU time: 3 ms

Gantt diagram:



Context switches: 29

RR scheduling

- Efficiency is highly dependend on time slice **q**.
  - For large **q**, algorithm becomes FCFS.
  - For a very small **q**, the system uses too much time for context switching.

- Generally 80% CPU phases should be shorter than **q**.

Many queues

- Different types of processes can be scheduled using queues with different scheduling algorithms.
  - Foreground processes - or interactive - RR
  - Background processes - FCFS
- Foreground processes have a priority against background processes.

- To avoid background process starvation, OS can allocate e.g. 20% of time for background queue.

Moving processes between queues

- We have a few (e.g. 3) queues.
  - Q0 has q=8ms
  - Q1 has q=16ms
  - Q2 is a background queue and is FCFS.
- New process are scheduled in Q0.
- When it uses all quantum of Q0, it is moved to Q1.
- If it uses all quantum of Q1, it's moved to Q2.
- Q2 is executed if Q0 and Q1 are used up.

- Currently it is one of most advanced short-time scheduling algorithms.

Thank You for attention

AGH

AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Introduction to Computer Science
Part 6

Version: 2023

Marek Wilkus Ph. D. **http://home.agh.edu.pl/~mwilkus**
Faculty of Metallurgy and Industrial Computer Science
AGH UST Kraków

Memory management

Program is starting

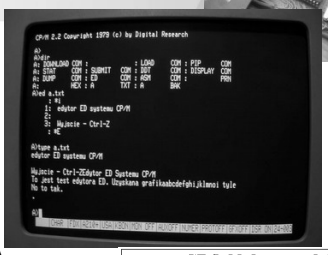
- Program's space is reserved in the memory.
  - Program binaries and constants
  - Program's variable space: Stack and heap.

- However the memory is organized, in program's code, it's starting from 0 and is continuous.
- All incontinuities are introduced later, during memory operation.

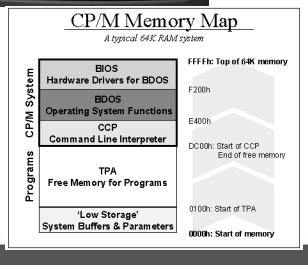
Memory manegement

- Program loaded into RAM → Process.
- Process is executed from the RAM, using variables in the RAM.
- Processes are **relocatable** - they can reside in any part of RAM.
- Program **has no knowledge** where it is located - addresses go from 0 - they have to be calculated during operation by OS or MMU.

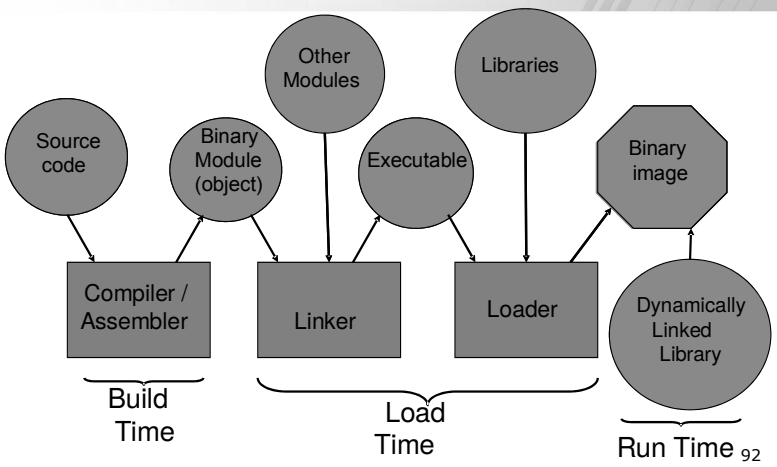
History



- 198x – CP/M – Memory for OS, BIOS and programs (TPA).
  - TPA – Transient Program Area - for programs.
  - Running a new program causes overwriting of TPA.
  - If we can bankswitch TPA, we can pretend 2 programs are multiplexed.
  - CP/M is a single-tasking OS - no timesharing!



Source code → Process



Program's calls → Memory addresses

- Translation can be done at any stage:
  - During **building** - if the memory addresses are hard-coded, it's an **absolute code**, for e.g. small processors, DOS .com executables, some low-level device drivers.
  - During **loading** - OS may locate the program's image in a memory area, calculate and patch its addresses then.
  - During **run time** - If the code is dynamically located and relocated during run, there must be some mechanism to dynamically alter memory addresses.

Loading the program

- If the program is not loaded to the memory until it's needed to execute it, the program is **loaded dynamically**. Most operating systems avoid loading programs „in advance", although some high-level methods for it exist.
- If some rarely used features of the program is compiled into external binaries, they don't occupy the memory but are loaded when needed. It is also possible to relocate some features to external **dynamic link libraries** or **shared objects** (DLL, SO) and use them when needed by one or many programs.
  Additional advantage: Fixing a bug does not need to recompile everything.

Overlays

- Historically used when programs required more memory than it was in system.
- Example: Two-pass compiler:
  We have 150kB of free RAM, program's components are:
  - Pass 1 code: 70kB
  - Pass 2 code: 80kB
  - Additional symbols: 20kB
  - Common functions: 30kB
- Total: 200kB

Overlays

- We can split the operation to two passes:
  1. Symbols, common routines, Pass 1 code, overlay support (10kB): 130kB total.
  2. Symbols, common routines, Pass 2 code, overlay support: 140kB Total.
- Both are under 150kB.
  - Overlays are held as absolute binaries and are loaded when needed by the overlay support module.
  - Overlay-based program does not need any complex memory management in the OS - it does these things by itself.

Memory space - logical and physical

- **Physical address** - An address used in the memory, of a specific word in the RAM.
- **Logical address** - An address generated by the CPU running a program.

- To translate logiacl to physical addresses, modern hardware uses programmable MMU - Memory Management Unit.
- Usually, the translation works by adding an **offset** to the logical address. The offset is stored per-process in the **offset register**.
- Userspace program works only on logical addresses.

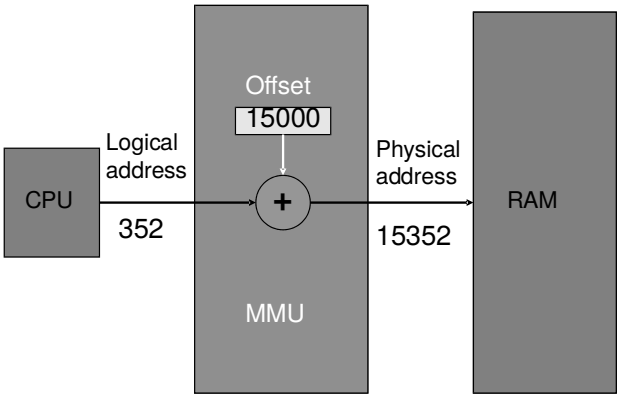
Intel's virtual address

- ...is between logical and physical addresses.

- It is an address in the memory, assuming an entire memory is one, continuous space.

- It hides paging, memory banking, parallel access ti multiple words or some aspects of dividing memory to segments.

Logical→Physical address



Swapping

- A process can be temporarily **swapped** to the disk to conserve memory.
- Then, another process may be swapped back to RAM.
- Usually with swapping-only systems, processes return to the same memory areas, so no re-computations are necessary.

- Swapping != paging! Currently paging is used.

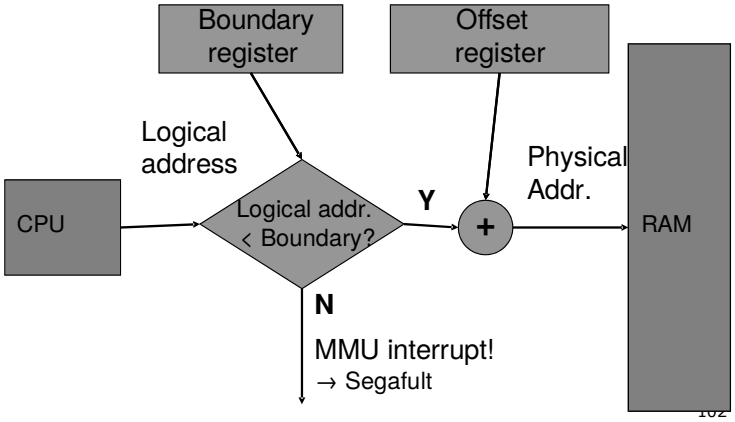
How to give memory to process?

- RAM is occupied by:
  - OS - Usually redises near interrupt vector and I/O devices adresses.
  - User space processes, usually located „above".

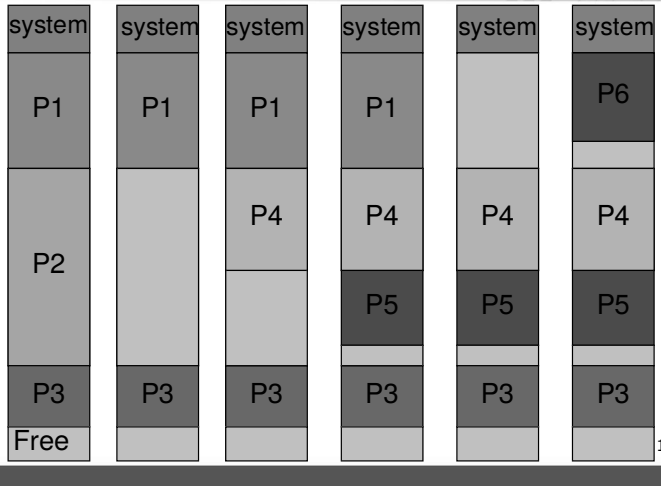
- To make user space processes not collide with each other, MMU is programmed with **offset register** and **boundary register:**
  - Offset - a minimum physical address a process can address,
  - Boundary - maximum logical address.

- In some MMUs, to make things faster, every process has a **context register** to quickly switch the MMU to current process context.

Controlling what process can do



Multiple processes, continuous reserving



How to chose a „hole”?

„Hole” - a continuous area of free memory.

- **First match** - the first free space equal or larger than process size. Searching can be done starting from the beginning or match from previous searching.
- **Best match** - chosen to leave as small leftover „hole” as possible.
- **Worst match** - because maybe the largest „hole” will be OK for some other program?.

- Usually, first (speed) or best (efficiency) match approach are better.

Fragmentation

- **External fragmentation** - There is enough free memory, but it is not continuous.
- **Internal fragmentation** - To minimize unusable small „holes”, if a few byte leftover is left after reserving, it is reserved too. Maybe it will fit some dynamic variables? But certainly it will decrease number of „holes” to search.

- To miminize external fragmentation, processes can be re-located to occupy continuous area.
  Or, memory can be divided to constant-size blocks - then we have a tradeoff between external and internal fragmentation.

Paging

- Physical memory is divided to **frames** of a 2^n size (e.g. 4kB, generally 512B - 16 MB).
- Logical memory is divided to **pages** of the same size.

- Free memory is in the free frames list.

- N-page process is loaded into N frames. They may not be continuous.

- The **Page Table** translates logical to physical addresses.

- External→internal fragmentation tradeoff is present.

Paging in the CPU

Every logical address is composed of two parts: Number of page and offset in this page.

If page size is a power of 2 and:
- Address space is 2^m,
- Page size is 2^n,

then, m-n more significant bits of address point to the page number ($=2^{(m-n)}$),

- n less significant bits point to the offset.

Page table

- Is kept in RAM too.
- Its size is dependent on size of the memory or address space.
- So one memory access is in fact **two memory accesses. This is slow.**

- To make things faster, we can **store** the most frequently used table records in some fast memory - a **cache**.
- This cache is usually installed near CPU, has 8..32768 records.

- If page's address is not found in cache, we have to perform 2 RAM accesses.
  With a well designed buffering algorithm, we may get 80-98% of page hits.

Cache memory

- Fastest: CPU Registers:
  - Too small for cache.
- Dedicated L1 cache:
  - Faster than RAM, connected directly to MMU.
  - Expensive.
- Dedicated L2/L3/… cache:
  - It is possible to make a paging hierarchy (page, hierarchized page directories - Intel does this)
  - Deeper levels are looked more rarely - can be slower and less expensive.
  - Some may be even expandable.
- RAM: Slowest in this comparison.

Effective access time

- An average time of accessing a memory address.
- For example:
  Cache lookup: 20ns
  RAM access: 100 ns
- Page hits - % of pages found in cache.
  For page hits: 20+100= 120 ns
  For page faults: 20+100+100=220 ns
- For 80% hits:
  Eff. time=0,8*120+0,2*220=140 ns
- For 98% hits:
  E.C.D.=0.98*120+0.02*220=122 ns

Memory protection

- With special protection bits, any frame can have permissions to read, write or execute code from.
- Page access bit - if it is set, the page can be accessed from the current context (process, mode).

  If not → illegal access → interrupt → segfault.

Multi-level paging

- Address space in modern systems is very large.
- If page table has millions of entries, the page table may be larger than a process!
  - ...and lookup becomes too long.
- Large tables can be divided to smaller ones.
- For example, 32-bit address can be split to:
  20-bit page address and
  12-bit offset

  or:

  10-bit page directory entry address,
  10-bit page address
  12-bit offset.

Going multi-level

- Every level has its own table, so getting the physical address may require e.g. 4 memory accesses.
  E.g. access time may be 520 ns.
- But every level may have own cache which makes things faster.
- So for 98% page hit:
  Access time is 0,98*120+0,02*520=128 ns.

Inverted page table

- One entry - one frame in RAM's physical addresses.
- Every entry has PID of process using the frame.
- Can be sped up using hashing tables.
- PROBLEM: Many processes cannot share the same memory fragment.
- Currently quite experimental thing.

Shared pages

20 users use text editor in the same machine. Do we really need 20 frames of the same editor's binary?

- If the code does not modify itself, we can point multiple pages to the same frame.
- Every process than has own data memory, and shared code area.
- Currently many programs and libraries use it.
- That's, among other factors, why it's hard to determine memory usage in some Linux systems.

Zero page

- If a program reserves memory, it doesn't matter it will use it right now.
- So if we reserve a heap, at first, it does not point to the free memory, but to the **zero page**.
- If the memory is read, it returns 0s.
- If the memory is **written**:
  - System gets an interrupt,
  - A free frame is found,
  - The pointer is re-connected from zero page to the free frame,
  - The page table is updated,
  - Program is resumed.
- This way, it is theoretically for program to reserve e.g. 200GB of memory in system with 128MB of RAM.
  - The limit is page table size.

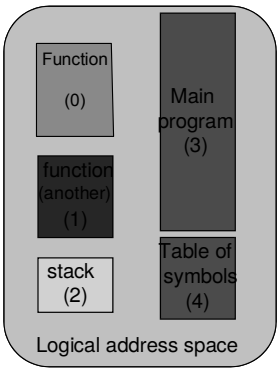
Segmentation

- How to keep which pages are for the shared binaries, which are for external symbols, which are for data?
- Memory can be reserved in **segments**, which is a mechanism running above paging.
- Logical addresses are a set of segments, so every address is just a segment number and offet in this segment.
- This is then calculated to Page-Offset or similar method and addressed.
- Modern MMUs support segmentation.
- Segments can be organized to:
  - Program's code,
  - Functions segments,
  - Program's or function's data,
  - Specific routines,
  - Etc.

Segmentacja

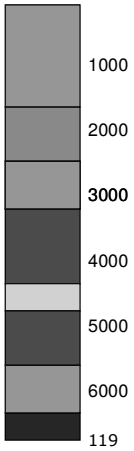
- Logical address: Segment ID + offset
- Segment table then has:
  - base (physical address of the start of segment)
  - length (of the segment)
- The segment table is kept in RAM.
- As segment count and sizes are different, memory accesses can be verified using length of the segment.
  - Aren't we trying to go past the segment?

Example



	Base	Length
0	1400	1000
1	6300	400
2	4300	400
3	3200	1100
4	4700	1000

Segment table



PAE - Physical Address Extension

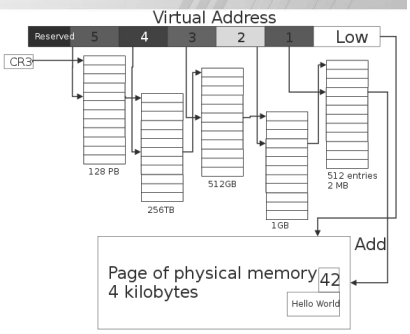
- 32-bit Intel's Architecture's addres space is:

  2^32 = 4 294 967 296 bajtów = 4GB

  How to support more?
- We will double the length of page table's entry.
- So there is still 32 address lines, but address can be twice long.
- So still **every context (process) can address 4GB at max**, but these may not be the same 4GB than the other process has!
- MMU supports it since Pentium 4.
- Supported by 32-bit Linux/Unix, Windows later than XP (XP too, but it was intentionally disabled)

Intel 64-bit

- Even more complex and multi-level.
- Two addressing modes:
  - 4-level paging on 64-bit addresses. Maximum address space is 256TB.
  - 5-level paging - virtual memory up to 128PB, but with less caching (it takes longer to find an address)



Virtual memory

- Does the process use the reserved memory efficiently?
- It was found that most processes use only a small part of the reserved memory for most of time.
- So what if we store some pages on disk
  - Paging file
  - Paging partition (Unix' „swap").

Swapping vs paging

- Swapping is for an entire process.
- Paging is for specific pages.
- In virtual memory approach, most programs do not reside entirely in RAM.
- The paging routine predicts which pages will be in use and transfers them from paging device to the RAM.

Page fault - what to do?

- Page has not been found in RAM - interrupt fires and it's OS job to bring it from the disk.
  - Process is halted and preempted,
  - Checking is the page „legal" in the context of the process,
  - Disk I/O (very slow)
    - ...a few processes may get CPU this time.
  - Correcting of page table,
  - Waiting for free CPU to restore the process.
- This is slow.
  - According to some measurements it can be 250000 times slower than ordinary RAM access.

How to speed it up?

- Predicting of the needed page must be done correctly to minimize page faults.
- Some pages may be marked that it's better to replace them than other.
  - If the page has not been written to, it may be replaced by other one faster than the page which needs to be saved on disk.
    - A „dirty bit" informs OS that the page has been changed in RAM.
  - If the page is unused for a long time, it may stay this way for a bit longer.

Which page to replace?

- FIFO - Replace the „oldest" page in the memory.
- Requires to store how long the page is in RAM.
- Problem: It may be used by some OS programs.
- Problem: sometimes increasing page cound for program increases page faults cound (Laszlo Belady's anomaly).

Which page to replace?

- Optimal: Rteplace the page which will not be used for the longest time.
- This is the optimal way.
- Problem: we cannot predict how long the page will stay unused.
  - So it is used for theoretical comparisons.
- Least Recently Used:
  - Store access counter for every page,
  - Replace the page which has been unused for the longest time.

LRU improvements

- We may store the information in two bits:
  - Was the page used recently?
  - Dirty bit.
- Then:
  - 0,0 - not recently used, not modified - best page to replace.
  - 0,1 - not recently used, but modified - worse thing, it's needed to write it to disk.
  - 1,0 - recently used - may be needed in a moment.
  - 1,1 - recently used and modified - wirst page to replace.
- We can use CPU's instructions to find the lowest value.

Process goes out of control

- Process' **heap** goes from one side of the logical memory area.
- Process' **stack** goes from the other side.
- We can increase the stack by:
  - Calling functions by functions,
  - Using single'use „scratch" variables.
- We can increase the heap by:
  - Reserving large memory areas,
  - Using many variables,
  - Creating duplicated of existing data structures.
- What will happen if they meet?

Overwriting

- We can **overwrite the stack** by the heap.
- This will cause return from the recent function point to illegal address
  → segmentation fault (core dumped).
- What if we use this **intentionally?**
- **How to break the stack?**
  - Make program reserve too much memory - e.g. by loading a specifically crafted file.
  - Or by executing user's commands in a specific order.
  - Or by misusing some unguarded functions.
  - ...

„Where do you want to jump today?"

- By cleverly overwriting the stack, the return from the function will jump **to the place we want**.
- We can execute **any** part of the program's code this way!
- Now we **disassemble** the code and look for some interesting parts - writing files, or executing programs from the disk.
- If it is programmable by the data in the heap, it is possible to craft this data too, and such „**gadget**" is extremely useful.

Making program do unusual things

- This way we can e.g. force a program running on a level of other user to „return" to the code which will spawn a shell with this user's privileges, but in our control.
- This technique is called a **return-oriented** programming and is currently a serious security-related problem.
- The malicious data can be supplied as:
  - A file - image, data, document - making program reserve a lot of memory and write file contents to it,
  - A user-supplied data - e.g. by abusing a form input,
  - A malicious library patched into program,
  - ...

Other solutions

- ASLR - Address Space Layout Randomization - what if we cannot be sure where the „gadget” is?
  The address space can be randomized.
  If not hardware-accelerated, slows down the program.

- Retpoline - every return, jump thru kernel's function.
  Kernel verifies is everything OK, and allows to return or not.
  This is AWFULLY slow.
  Sometimes it's >20% performance loss.

Windows NT kernel fix

(ctxswap.asm, Windows source tree: base/ntos/ke/$ARCHITECTURE$)

```
:PublicProc _KiDispatchInterrupt ,0
:PublicFpo 0, 0

        mov     ebx, PCR[PcSelfPcr]     ; get address of PCR
kdi00:  lea     eax, [ebx].PcPrcbData.PbDpcListHead ; get DPC listhead address

;
; Disable interrupts and check if there is any work in the DPC list
; of the current processor.
;

kdi10:  cli                             ; disable interrupts
        cmp     eax, [eax].LsFlink      ; check if DPC List is empty
        je      short kdi40             ; if eq, list is empty
        push    ebp                     ; save register

;
; Exceptions occuring in DPCs are unrelated to any exception handlers
; in the interrupted thread.  Terminate the exception list.
;

        push    [ebx].PcExceptionList
        mov     [ebx].PcExceptionList, EXCEPTION_CHAIN_END

;
; Switch to the DPC stack for this processor.
;

        mov     edx, esp
        mov     esp, [ebx].PcPrcbData.PbDpcStack
        push    edx

.fpo (0, 0, 0, 1, 1, 0)

        mov     ebp, eax                ; set address of DPC listhead
        CAPSTART <_KiDispatchInterrupt,KiRetireDpcList>
        call    KiRetireDpcList         ; process the current DPC list
        CAPEND <_KiDispatchInterrupt>

;
; Switch back to the current thread stack, restore the exception list
; and saved EBP.
;

        pop     esp
        pop     [ebx].PcExceptionList
        pop     ebp
.fpo (0, 0, 0, 0, 0, 0)

;
; Check to determine if quantum end is requested.
;
; N.B. If a new thread is selected as a result of processing the quantum
;      end request, then the new thread is returned with the dispatcher
;      database locked. Otherwise, NULL is returned with the dispatcher
;      database unlocked.
;
```

Prepare for context switching after issuing return (lock interrupts, check do we chain returns using a return list)

Windows NT kernel fix

```
.fpo (0, 0, 0, 3, 1, 0)

kdi60:  sub     esp, 3*4
        mov     [esp+8], esi            ; save registers
        mov     [esp+4], edi            ;
        mov     [esp+0], ebp            ;
        mov     esi, eax                ; set next thread address
        mov     edi, [ebx].PcPrcbData.PbCurrentThread ; get current thread address
        mov     dword ptr [ebx].PcPrcbData.PbNextThread, 0 ; clear next thread address
        mov     [ebx].PcPrcbData.PbCurrentThread, esi ; set current thread address
        mov     ecx, edi                ; set address of current thread
        mov     byte ptr [edi].ThIdleSwapBlock, 1
        fstCall KiReadyThread           ; ready thread for execution
        CAPSTART <_KiDispatchInterrupt,SwapContext>
        mov     cl, 1                   ; set APC interrupt bypass disable
        call    SwapContext             ; swap context
        CAPEND <_KiDispatchInterrupt>
        mov     ebp, [esp+0]            ; restore registers
        mov     edi, [esp+4]            ;
        mov     esi, [esp+8]            ;
        add     esp, 3*4
kdi70:  stdRET  _KiDispatchInterrupt    ; return
```

Save registers for context switching and save the return address!
If it won't be on the list → Something modified return address!

. . . . .

PROBLEM: Still „speculative” attacks are possible.

Thank You for attention

AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Introduction to Computer Science Lecture Part 7

Version: 2023

Marek Wilkus Ph.D.
AGH Kraków
http://home.agh.edu.pl/~mwilkus

File system

What is a file?

- **File** - a logical unit of data storage stored in a non-volatile memory.
- **File** - an identifable collection (set) of information, stored in a non-volatile memory.
- **File** - is a series of bytes, bits, lines or records in a device.

- File attributes:
  - Identifier or name (made according to FS conventions),
  - Type (if required),
  - Location (...on a specific device),
  - Size (in bytes, characters, words or blocks),
  - Permissions and attributes,
  - Time, date records, owner records,
  - FS-specific metadada.

Two types of file usage

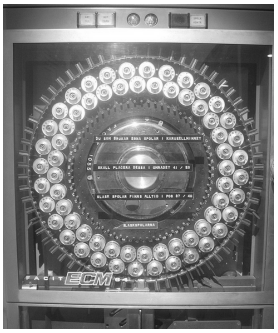
- Modern **Computer** operating systems use filesystem as a primary data stroage. The system usually is also stored in the filesystem.
  - It means that filesystem driver has to be loaded very early.
  - Problem with filesystem → OS fails to run.
  - Usually support for many file systems is present, and some OF may support adding external ones (Linux, F.U.S.E.).

Two types of file usage

- Simple portable devices or proprietary appliances (data collectors, measurement systems, media players etc.) may use filesystem for auxiliary purposes - for getting additional data or writing data to.
  - Filesystem implementations are usually much more simple.
  - Sometimes, only the simplest FAT is present.
  - OS does not reside in the file system - it is usually stored in a read-only memory.
  - Theoretically, the device can run without a disk.

History

- Tape-based computers: File is a device connected to I/O channel.
- Later: There can be more files on a device, but it is still about changing tapes - manual or automatic (Facit Carousel project),
- Later: There may be more files in a single tape. It was needed to increase tape reliability (vacuum, etc.)
- Later: Indexed tapes.



- Disk-based systems: A simple file list (identifier-offset-length).
- Later: Complex file systems.

History of implementations

- 1970s-80s - microcomputers:
  - Device (e.g. tape, printer, floppy drive - is a file)
  - Later: On a random-access device there can be more files. Previous approach is still in place for devices.
  - No place for disk OS - it is loaded from tape or bootstrapped from disk drive's ROM.
  - CP/M-based machines: Disk drive is a computer. Computer is a terminal. (that's why disk drives were so expensive - there was a second complete computer inside!)

History - Hierarchical FS

- One of the first hierarchical FS - Unix FS (UFS).
- IBM PC (DOS 1.x) - Simple file list on a floppy disk.
- IBM PC (DOS 2.x) - Directories. Hard disks up to 20-30MB (or more - with drivers), no standard for larger HDDs.
- DOS 3.x-6.x - FAT - Multiple partitions, nested directories.
- Windows 95 - VFAT i 95 OSR2 OSR2 - FAT32 - bigger disks support, extension of 8+3 naming.
- Windows NT - NTFS - New Technology File System - Streams, extended attributes, sparse files, object filesystem (never fully implemented).

File-related operations done by OS

- **Creating file**:
  - Finding free place in the file list and reserving space
  - Inserting entry.
- **Writing to file** - Identifier of a file and data is given. It is important to keep the information where to write (offset).
- **Reading file** - Identifier of a file, length and memory buffer is given. The same offset can be used.
- **Seeking position in a file** - moving the offset.
- **Removing a file** - Removing of directory entry and, if needed, discarding the space used by file.
- **Truncating file** - Discarding file's data without removing directory entry.

Other operations

- **Appending** - Adding new records to existing file.
- **Renaming** - usually done with the same command as moving, manipulates metadata.
- **Opening a file** - Creating a **handle** to perform other operations on file by other OS functions.
- **Closing a file** - Finishing working with a file by discarding its handle.

- **In mutli-tasking and multi-user OS, files should be open the way to minimize program's interferences.**

File Types

- OS can detect file types by:
  - Extensions - in MS-DOS/Windows, 3 last characters of the name, after the dot,
  - Magic bytes - normal in Unix and modern Macs, by contents of a file (see Unix command: file, and /etc/magic information file),
  - Creator/application attributes (Classic Mac OS) - A specific ID of program to open soecific file is written in its metadata.
  - Mixed - e.g. Magic bytes + extension (GUI Linux) or Megic bytes + Creator ID (early Mac OS X)

Accessing a file

- **Sequential** - Processed record-by-record in the order the data is written to, used in early implementations and sometimes forced in multi-CPU software.
- **Random access** - Any order is possible, file can be read backwards or seeked record by record. Used in modern filesystems and database-dedicated FS.
- **Indexed access** - There is an index file specifying „what goes where" - used in older DB systems.

Directories

- **Single-level** - Only name uniqueness has to be maintained.
- **Two levels** - Every „user" has own directory with files, no further hierarchy.
- **Multi-level tree-like** - currently used in most systems.
- **Non-cyclic graphs** - There are many ways to reach a file. Currently implemented, but rarely used.

Protection

- File systems allow to specify what an user can do:
  - **Reading** file's contents,
  - **Writing** or overwriting a file,
  - **executing** a file, or passing it to the Os' loader,
  - **appending** to file,
  - **removing** file,
  - **Modifying metadata** - including name and attributes.
- Typical UNIX user classes:
  - **Owner** - by default, user which created a file,
  - **Group** of users defined by OS rules,
  - **Other** users.

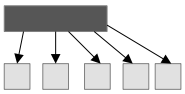
How to reserve space for file?

- **Continuous** - no fragmentation, file after file.
  - Advantages: Simple, minimal drive seeking, possible to implement on a microcontroller.
  - Disadvantages: Almost no way to append to file, hard to write to. This is mostly useful in Read-only file systems (ROM memory of embedded devices, BIOS modules, loaders, encryption systems).

- **List-based** - Every file block has a pointer to the next block, like in a singly-linked list.
  - Advantages: No external fragmentation, files can be appended to, seeking forward is fast.
  - Disadvantages: Random access is difficult, if a link breaks, the „hanging" chain cannot be used anymore and is usually discarded, pointers take much space.

How to reserve space for file?

- **Index-based** - like list-based, put pointers are in a single structure.
  - Advantages: like in a list-based, the index can be buffered so seeking is faster, no external fragmentation.
  - Disadvantages: Pointers usually take more space than in a list-based method, **a single point of failure.**



- **Mixed implementation ideas:**
  - If the index overflows, a pointer to the next part is given like in a list-based. This slows the seeking down.
  - Multi-level indexing - Makes more sense, especially if first entries are indexes and other are hierarchical continuation of the index. This is used in Unix.
  - Making things faster - organize blocks in a balanced BST by the offset - sequential seeking goes faster then.

Performance!

- Having a list-based reservation and random access, the performance will drop - accessing block n requires n accesses to the disk.
- Increasing performance:
  - When a file is created, its access mode can be declared - but when it ends with direct access, the size must be known (useful for VM disk images).

    For sequential access, a list-based or tree-based reservation is then used.
  - It can be dependent on file size - small files may be reserved as a continuous space, larger in e.g. tree-based.
  - Some FS allow to use clusters (groups of sectors) of different sizes - smaller files may get 4kB, larger - 64kB, then filled to the end with 64kB. Efficient, but difficult to implement.

Management of a free space

- **Bitmap** - In a continuous bit vector occupied blocks are 0, free are 1. There is a CPU command to find the position of the first bit lit, so it is easy to find a free block.
- **Linked list** - a „virtual" free space „file".
- **Grouping** - A set of blocks is hierarchically defined as a free space.
- **Counting** - An address of the first block and number of free blocks following it is stored.

Increasing performance

- **Disk caching** - Entire fragments of files, records, or even disk tracks are stored in unused RAM for reading or writing.
- **Early freeing** - In memory-constrained system, the buffered block can be freed from the memory right after the file operation. If the buffer is exactly the size of processed data structure, does not decrease performance.
- **Reading ahead** - Reading and buffering next blocks of the file before they will be needed.
- **RAM-disk** - A fast disk for operations, this is a volatile medium.
- **Delayed writing** – Flushing buffers as late as possible - preferably at the program exit - to minimize unneeded writes.

Increasing performance

- **Copy On Write** - If making copy of the file, create the metadata block, but reserve space and copy data only when it is modified.
  - Things go much faster,
  - ...however, it's more difficult to implement in multitasking and multi-user systems. Usually, the „dirty bit" for a block is used.
  - Problems: Nov. 2023, OpenZFS, „dirty" bit is not forwarded to **next** blocks when they dirty.
    - Result: Files corruption!

Operator actions

- **Checking the FS** - chkdsk (Windows), fsck (Unix/Linux) - done periodically or after removing the disk without unmounting (the „clear bit").
- **Backups and restoring** - Copying the disk to the backup media.
  - **Incremental** - by using a previous backup as reference.
  - **Disk imaging** - by copying an entire disk to the file.
- Disk **Compression** - mounting the compressed file as a „virtual" disk, the file is on the „real" disk.
  - More space, slower access.
  - Similar strategy is used with encryption.

Example - DOS FAT

- **Directory** - records consist of file name (8 chr), extension (3 chr), length, attributes (h s r a), creation date, FAT offset/entry.
- **FAT** (file allocation table) - elements of this table represent consecutive blocks (sectors, or clusters). Written at the beginning of partition in 2 copies for safety.

Example: File F1 is in blocks: 7,8,11,3, F2 is in 4, ang F3 is in: 1,2,5,6,9

```
No:  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 15 17 18 19 20
    02 05 ff ff 06 09 08 11 ff 00 03 00 00 00 00 00 00 00 00 00
```

FAT 12 - each entry is 12 bit long, so 0-4096, with 8kB clusters (considered large) gave maximum drive size of 32 MB

Later versions - FAT 16 (up to 2 GB) and FAT 32.

The biggest problem: Time to seek thru the FAT list.

Example – NTFS (Windows)

- **Volume** - a virtual representation of a partition - maty be a disk, partition, mounted resource or a group of disks.
- **Cluster** - A base allocation unit - usually 4kB, smaller than in FAT. Modern disks have sectors as large as clusters.
- **File** is written as an object consisting of attributes. File metadata, location in the volume and data itself are attributes.
- **Master File Table** (MFT) - Stores file records.
- **File identifier** - an unique identifier of each file, consists of 48-bit MFT position and 12-bit „serial" number.

NTFS – What is stored in?

- **MFT backup** - first 16 or 32 entries are backed up for fixing the rest of MFT,
- **The Log file** - ontains operations performed on the disk,
- **Volume file** - NTFS version, volume size and characteristics, the „clear bit",
- **Attribute table** - A table of available metadata attributes and operations on them,
- **Main directory**
- **Cluster bitmap** - for storing info about free clusters,
- **Loader** - Windows loading code,
- **Bad blocks log**,
- **If a problem occurs** during disk operation, it is simple to recover FS to working condition by rolling back the **transaction**.

NTFS – Inside the „System Volume Information"

- $Mft - Table of file/directory records.
- $MftMirr - a backup of the first 4/16/32 MFT records - they store what is in the „System Volume Information" directory.
- $LogFile - Transaction log.
- $Volume - Volume Information File.
- $AttrDef - Available attributes.
- $$ - Root directory definition.
- $Bitmap - Free Space bitmap
- $Boot - Boot program (only in Windows boot partitions)
- $BadClus - (?deprecated?) Bad clusters information.
- $Secure - Permissions definitions.
- $UpCase - Used for translating file names, maintaining sorting order.
- $Extend - All extended attributes for volume like per-user permissions, quotas, network sharing etc.

NTFS – file operations

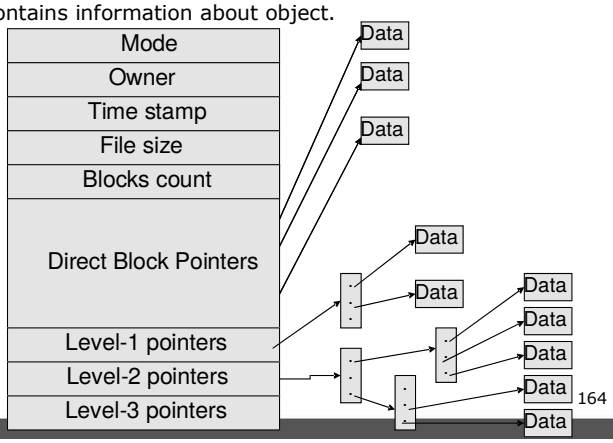
- Every operation is a transaction.
- At first, the operation is written to the transaction log. Then, the operation is performed and confirmed in the log.
- If there was a failure, the log is re-processed and operations are recovered or discarded.
- This guarantees that operations will not damage the filesystem. **Data may be damaged**, but filesystem should allow to recover undamaged data, not totally „fall apart".

NTFS

- If the cluster does not store the information → mark as bad, append to the bad clusters list, use another cluster.
- There are more layers of protection against bad blocks and this is one of the last ones.
- Repeating problems → FS driver writes it to the system log, it's a good time to look for a new hard disk.

Classical Unix FS

- Directory: File names (up to 256 chars), inode pointer. Different from a file by one attribute.
- I-node - contains information about object.



Unix filesystem

- **Superblock** - A record of static drive's parameters, filesystem size, block lengths, access modes, quotas.
- **Cylinder group** - now deprecated - a small „sub-filesystem" for faster single-file operations:
  - Have own superblock,
  - Have a „cylinder block" (incl. free blocks list, file links, quotas, attributes),
  - I-nodes,
  - Data blocks (to the end of the group).

Typical problems

- Classical UNIX/Solaris: Permanent FSCK and data loss at boot-up:
  - Operator partitioned the disk not according to documentation and overwrote the superblock with partition start.
- Linux: FSCK continuously indicates non-existing i-node removal:
  - Part of the filesystem was synchronized from the network and got a metadata from the future.
- FAT implementation: Directory record split in half:
  - This is not present in most FAT documentation: Directory record is a file too and must be read traversing thru FAT structure.

Ext4 FS

- Default FS in modern Linux distributions.
- Successor of Ext3, Ext2, Ext, Minix FS…
- Capabilities:
  - Devices up to 1EB (1 Exabyte=1024PB, 1PB=1024TB),
  - Unlimited directory depth,
  - Transaction log with checksums,
  - API for on-the-fly disk encryption,
  - Delayed writing of i-nodes from cache.
  - Year 2038 problem is delayed by adding 2 bits (now it is Year 2446 problem).
  - FS-level quotas are more flexible than „hard" partitioning.

Ext4 - Space management

- Blocks are purely „virtual" - space is allocated in **extents**.
- There are 4 entries for 4 extents in metadata of a file. Maximum size is then 128MB * 4 = 512MB
- Bigger file - multi-level storage in a self-balancing binary search tree aligned the way that it's preferable to seek by offset.

Ext4 - External fragmentation minimization

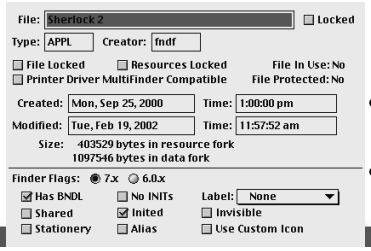
- Allocate-on-flush - Blocks are allocated when the file is finally written to disk.
- Preallocating - It is possible to pre-allocate blocks for a specific file.
- New files are written the way that number of useless holes is minimized.
- A small (1-5%) part of the disk is locked for better holes management.
- For SSD - Trim support.

Ext4 - Directories

- A specific H-tree structure.
- To obtain all file entries from specific directory, we llok for range of files, not a specific value. The H-tree is optimized for such searching.
- If we get to the directory entry, most of time an entire content is stored in its leaves.
  - The exceptions are mostly links.

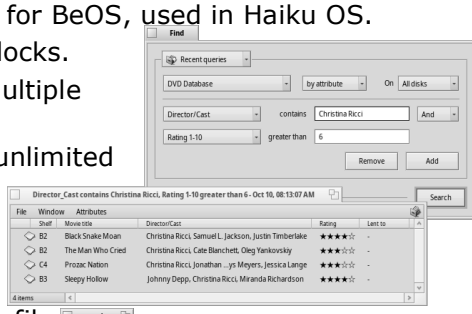
More Metadata! Apple HFS, BeFS

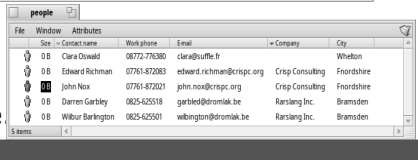
- Apple HFS and later: Every FS object has a "resource fork" describing metadata like:
  - Is the object a link, directory, file, or „non-object”,
  - Icon and color,
  - Program opening a file ("Creator"),
  - Icon location,
  - Is an object named,
  - Text comment,
  - Position/size of application window, (OK, this was a bad idea)
  - Many more, most are not used anymore.
- In Mac OS X, it was slowly phased out, replaced by attributes stored in hidden dot-files.



BeFS

- Developed around 1996 for BeOS, used in Haiku OS.
- Extents based on disk blocks.
- Extents are packed in multiple levels like in UFS.
- Every file may have an unlimited number and types of metadata…
  - ...And it’s user-configured.
- Metadata is identified by file type.
- So the file system can be used as a personal database.





MS WinFS, NewFS, OFS, ???

- Never released in Windows system, but developed since NT3.51.
- Relative DBMS as file system,
- file: Data, Metadata, Resources, ways of accessing.
  - Way of accessing: Program + conversion module.
- Incompatible file types problem solved? - NO! - small number of these modules.
- ...Data and data-relation oriented.
- An application can generate FS on the fly - like Unix’ ProcFS, /dev, mail, cloud disks etc.

Windows 95: MS Exchange Mail Client pretends to be a folder

- Problems:
  - Formats are evolving and are incompatible.
  - No interoperability between platforms.
  - No backwards compatibility.

Project Longhorn (Vista): Folder as calendar →

Shortcuts and links

- Most modern filesystems use hierarchical structures for storing data (folders and files). However, there may be some exceptions.
- In Windows, it is possible to mount a directory to another directory - by using file system links. However, more popular is just using an „LNK” shortcut file, they are interpreted only by GUI shell (Explorer).
- Forcing to create a shortcut to a shortcut ends, depending on versions, with Explorer error, nothing at all or crashing all Explorer instances for all logged in users.
  - This technique was frequently used to infect USB drives when „autorun” became less popular.

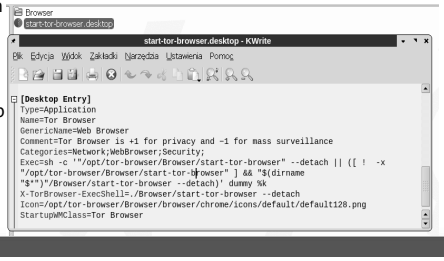
Shortcuts in Linux/Unix

```
mcbx@w4800:~$ ls -al /usr/bin/xw*
-rwxr-xr-x 1 root root 31216 mar 18  2018 /usr/bin/xwd
-rwxr-xr-x 1 root root 10664 sty 30  2016 /usr/bin/xwdtopnm
lrwxrwxrwx 1 root root     2 maj  5  2018 /usr/bin/xwe -> we
-rwxr-xr-x 1 root root 43808 mar 18  2018 /usr/bin/xwininfo
lrwxrwxrwx 1 root root     2 maj  5  2018 /usr/bin/xwpe -> we
-rwxr-xr-x 1 root root 27080 mar 18  2018 /usr/bin/xwud
mcbx@w4800:~$
```

- Soft link - Used in the console and GUI, both get information about it. It is a separate filesystem object of a „link" type. By deleting the source we get a broken link and loose the data.
  Creating: **ln -s source destination**

- Hard link - An entire filesystem entry for a file which already has an entry. File „disappears" - its blocks are discarded - when the last link gets deleted. Programs have no information that there are another links to the file.
  Creating: **ln source destination**

- *.desktop file - A full description of an appearance, interactions and opening of a program. Used only in GUI shell, and is created like a text file (which it actually is). Some text-mode shells may have support for starting desktop files, but usually only in GUI mode (when they are ran in a window, not in a console-only system).



Thank You for attention