VE482 Final Review

Memory management

Memory hierarchy

From expensive to cheap, fast to slow, small to large:

Registers -> Cache -> Main memory -> Magnetic disk -> Magnetic tape

Memory manager

- **Keep track** of which part of the memory is used
- **Allocate** memory to processes when required
- **Deallocate** memory at the end of a process

Remark. It is the job of the hardware to manage the lowest levels of cache memory

Model the hierarchy

Simplest model -- No memory abstraction

- Program sees the **actual physical memory**
- Programmers can access the whole memory
- Limitations when running more than one program:
  - No more than one program in the memory at a time
    - Or program A might change program B's memory
  - Have to copy the whole content of the memory into a file when switching program (**swapping**)
  - More than one program is possible if using special hardware
    - Use protection key to avoid processes interfering with each other and OS
    - Still need to solve the problem of **absolute memory** (Use absolute address will cause problems when running more than one program)
- Two main problems:
  - **Protection**: prevent program from accessing other’s memory
  - **Relocation**: rewrite address to allocate personal memory (problem of absolute memory)
  - Solution: invent a new memory abstraction -- **address space**

Memory abstraction -- address space

- **A set of addresses** that a process can use
- **Independent** from other processes’ memory
- To give each process its own private address space, a simple way is to use **base and limit registers**:
  - When the program runs, the address that it is loaded is saved into base register
  - And the length of the program is saved into limit register

- The address requested in the program should **add** base register before being used & **compare** to limit register to check whether it exceeds the space

Swapping

As more programs are run more and more memory is needed after booting, more memory than available might be needed.

-> Processes are **swapped in (out) from (to) the disk**, managed by OS

Some extra space may be allocated when swapping from disk

- Room for growth (data or stack)

Two ways to manage the usage of memory:

bitmaps

- Memory is divided into several allocation units
- an allocation unit <-> a bit in bitmap
- **0 -> free; 1 -> allocated**
- smaller allocation unit <-> larger bitmap
- bigger allocation unit <-> may be less accurate <-> waste in the last unit
- drawbacks: need to search the bitmap when swapping from disk

lined lists

- maintain a **linked list** of allocated and free memory segments
- a segment either contains a hole or a process
- The node of the linked list contains: **H (hole) / P (process)**, **start address**, **length**, and pointer to the next node.
- Sorted by address, so that it's easy to merge nodes when a process is swapped to disk.
- A double linked list may make it easier to merge.
- Algorithms to allocate memory when swapping from disk:
  - **first fit**: scan from begin and find the first hole that is big enough; spilt the hole to two parts
  - **next fit**: similar to first fit but start from where it stopped last time when finding the hole
  - **best fit**: find the smallest suitable hole
  - **worst fit**: find the biggest suitable hole

  If use separate lists for holes and processes, the algorithms can only check holes, without processes, which will be faster. If the linked list for holes is sorted by hole size, best fit will be as fast as first fit (next fit will be meaningless).

  - quick fit: maintains separate lists for some of the more common sizes requested

  However, separated lists sorted by hole size will be **slow when merging**.

  - Characteristics:
    - Speed: quick fit > first fit > best fit
    - Locally optimal: quick fit = best fit > first fit
    - Globally optimal: first fit > quick fit = best fit

Virtual memory

- Generalization of the base and limit registers
- Each process has its own address space
- The address space is split into chunks called **pages**
- Each page corresponds to a range of addresses
- Pages are mapped onto physical memory
- Pages can be on different medium, e.g. RAM and swap area

Paging

- Virtual address space divided into fixed-size units called **pages**, while corresponding units in physical address space are called **page frames**
- Pages and page frames are usually of **same size**
- **MMU** maps virtual addresses to physical addresses (store page table)
  - When visiting mapped virtual address, the physical address can be known from the page table
- MMU causes the CPU to trap on a **page fault**
  - When visiting page that is not mapped to page frame
  - OS copies content of a little used page onto the disk
  - The mapping is updated and new page frame is loaded

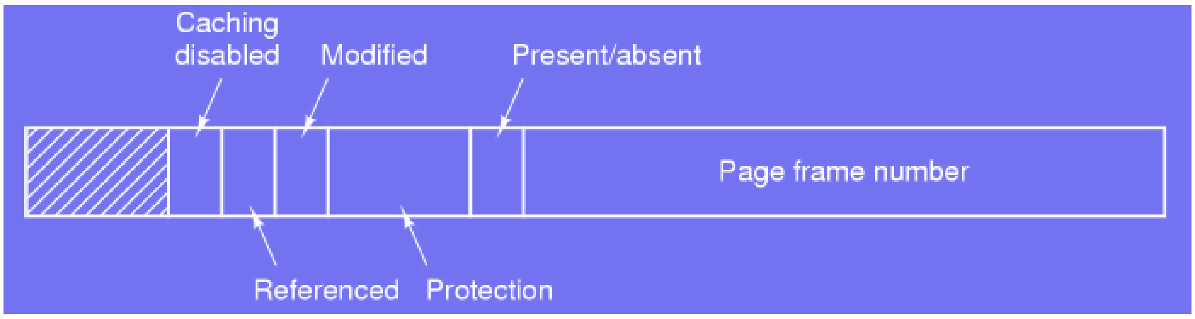
Page table

Virtual address: $\{\text{virtual page number}(\text{VPN}), \text{offset}\}$

- **VPN** can locate a page, and **offset** can locate a byte in a page
- VPN is used as **index of page table**, while corresponding page frame number is stored in the entry of that index.
- Replace VPN with page frame number to get physical address

Structure of a page entry

- **Present|absent**: 1|0; visiting an entry whose present bit is 0 causes a page fault
- **Protection**: 1 to 3 bits: reading/writing/executing
- **Modified**: 1|0 = dirty|clean; page was modified and needs to be updated on the disk
- **Referenced**: bit used to keep track of most used pages; useful in case of a page fault
- **Caching**: important for pages that map to registers; do not want to use old copy so set caching to 0



Translation Lookaside Buffer (TLB)

Be used to speed up paging

- Hardware solution implemented inside the MMU
- Keeps track of **few most used pages**
- $\{\text{valid}, \text{virtual page number}(\text{VPN}), \text{modified}, \text{protection}, \text{page frame number}\}$

When visiting a virtual address:

- Compare all the entries in TLB to find whether it's a **hit or miss** (through special hardware)
- **hit**: if not violet protection bit, page frame is acquired from TLB; if violet, a protection fault is generated
- **miss**: look up the page table, and replace an entry in TLB with the result.
  - When write back to page table, valid, modified, protection bits are copied, and set the reference bit
  - When load from page table, the information in the entry is copied from page table

Page replacement

On a page fault the following operations are performed:

- Choose a page to **remove** from the memory
- If the page **was modified** while in the memory it needs to be **rewritten** on the disk; otherwise nothing needs to be done
- **Overwrite** the page with the new memory content

To select the page to be evicted:

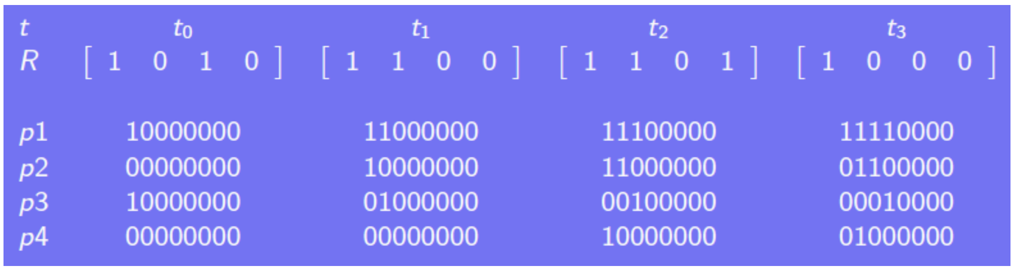
- **Optimal solution**
  - Label with the number of instructions that will be executed before the page is first referenced and order all the pages in memory
  - The page with **lower label is used first**
  - The page with **larger label is swapped out** of the memory
  - **Unrealizable**
- **LRU**
  - Recently heavily used pages are very likely to be used again soon
  - Hardware solution, for n page frames:
    - Initialize a binary $n \times n$ matrix to 0
    - When frame k is used set **row k to 1 and column k to 0**
    - **Replace** the page with the **smallest** value (read in rows)
    - e.g.: reference order: 1-2-3-4-3-2-1-4

$$\begin{bmatrix}0&1&1&1\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{bmatrix}\begin{bmatrix}0&0&1&1\\1&0&1&1\\0&0&0&0\\0&0&0&0\end{bmatrix}\begin{bmatrix}0&0&0&1\\1&0&0&1\\1&1&0&1\\0&0&0&0\end{bmatrix}\begin{bmatrix}0&0&0&0\\1&0&0&0\\1&1&0&0\\1&1&1&0\end{bmatrix}$$

$$\begin{bmatrix}0&0&0&0\\1&0&0&0\\1&1&0&1\\1&1&0&0\end{bmatrix}\begin{bmatrix}0&0&0&0\\1&0&1&1\\1&0&0&1\\1&0&0&0\end{bmatrix}\begin{bmatrix}0&1&1&1\\0&0&1&1\\0&0&0&1\\0&0&0&0\end{bmatrix}\begin{bmatrix}0&1&1&0\\0&0&1&0\\0&0&0&0\\1&1&1&0\end{bmatrix}$$

- **Aging**
  - Simulating LRU in software:
    - For each page initialize an n-bit software counter to 0
    - At each clock interrupt the OS scans all the pages in memory
    - Shift all the counters by 1 bit to the right
    - Add $2n - 1 \cdot R$ to the counter for each page

        - e.g.: 8-bit software counter



t	t_0	t_1	t_2	t_3
R	$\begin{bmatrix} 1 & 0 & 1 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 1 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 1 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}$
$p1$	10000000	11000000	11100000	11110000
$p2$	00000000	10000000	11000000	01100000
$p3$	10000000	01000000	00100000	00010000
$p4$	00000000	00000000	10000000	01000000

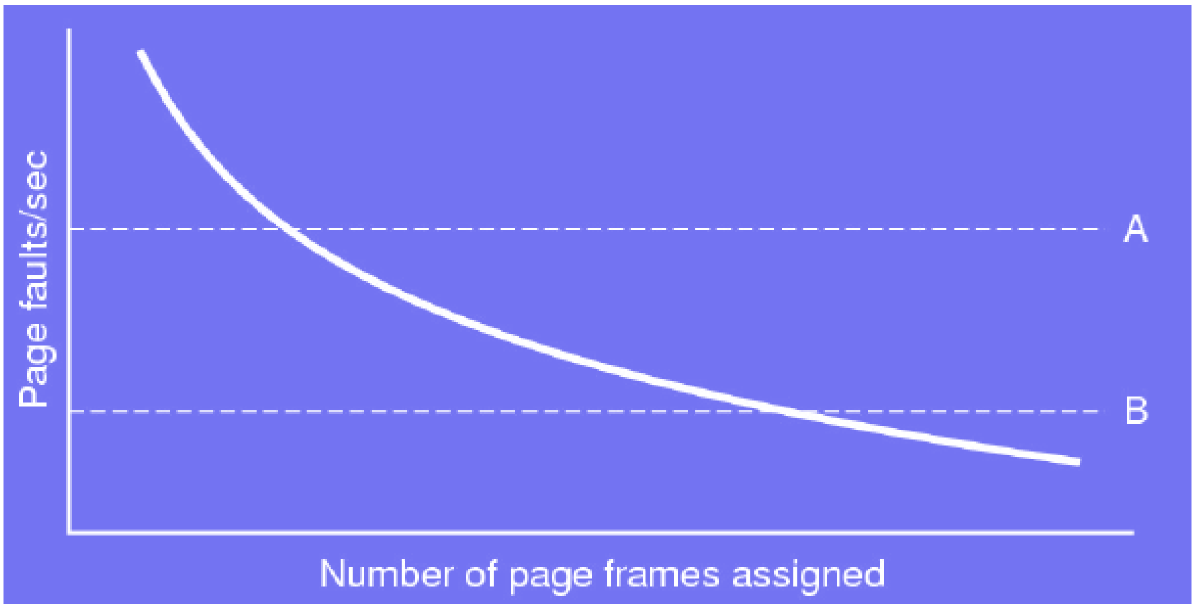
- **WSClock**
    - Basic notions:
        - **Demand paging**: pages are loaded on demand instead of in advance
        - **Locality reference**: during an execution phase a process only access a small fraction of all its pages
        - **Working set**: set of pages currently used by a process
        - **Thrashing**: process causes many page fault due to a lack of memory
        - **Pre-paging**: pages loaded in memory before letting process run
        - **Current virtual time**: amount of time during which a process has used the CPU
        - τ: age of the working set
    - Using a **circular list** of page frames for pages which have been inserted:
        - Each entry is composed of **time of last use**, **R** and **M** bits
        - On a page fault examine the pages the hand points to
        - If **R = 1**, bad candidate: **set R to 0** and **advance** hand
        - If **R = 0**, **age >** τ
            - If page is **clean**, then use page frame
            - **Otherwise** schedule write, move the hand repeat algorithm
        - If hand has completed one cycle
            - If **at least one write** was scheduled, keep the hand moving until a write is completed and a page frame becomes available
            - Otherwise (i.e. all the pages are in the working set) take any page ensure it is clean (if no, just take current page).

Local vs. global allocation

Onto which set should the page replacement algorithm be applied:

- **Local**, i.e. within the working set of process:
    - Allocate a portion of the whole memory to a process
    - Only use the allocated portion
    - **Number of page frames** for a process remains **constant**
- **Global**, i.e. within the whole memory:
    - Dynamically allocate page frames to a processes
    - Number of page frames for a process **varies over time**
    - Better in general, since fixed page frames number may cause thrashing or may waste memory when working set growing or shrinking.
    - Need to properly manage the number of page frames for each process.
    - One way to manage the allocation is **Page fault frequency**:
        - Adjusting the number of pages:
            - Start process with a number of pages **proportional to its size**

        - **Adjust** page allocation based on the page fault frequency
            - Count number of page fault per second
            - If **larger than A** then allocate more page frames
            - If **below B** then free some page frames



Page size

Finding optimal page size given a page frame size:

- In average half of the last page is used (**internal fragmentation**)
- **Large** pages will cause more **wasted space**
- The **smaller** the page size, the **larger the page table**
- **Smaller** page size requires **many TLB entries**
- Transferring small pages may cost **similar time** as bigger pages

Page size p, process size s bytes, average size for page entry e and overhead o:
$o = \frac{se}{p}(\text{page table}) + \frac{p}{2}(\text{internal fragmentation})$
Differentiate with respect to p and equate to 0:
$\frac{1}{2} = \frac{se}{p^2}$
Optimal page size: $p = \sqrt{2se}$
Common page frame sizes: 4KB or 8KB

Page sharing

Decrease memory usage by sharing pages:

- Pages containing the **program** can be shared
- Personal data should not be shared

Several basic problems arise:

- On a process switch, do not remove all pages if required by another process, or it would generate many page fault
- When a process terminates, do not free all the memory if it is required by another process, or it would generate a crash
- How to share data in read-write mode?
    - **Copy on write**: copy those pages that are modified, so for each process it has a READ/WRITE copy.

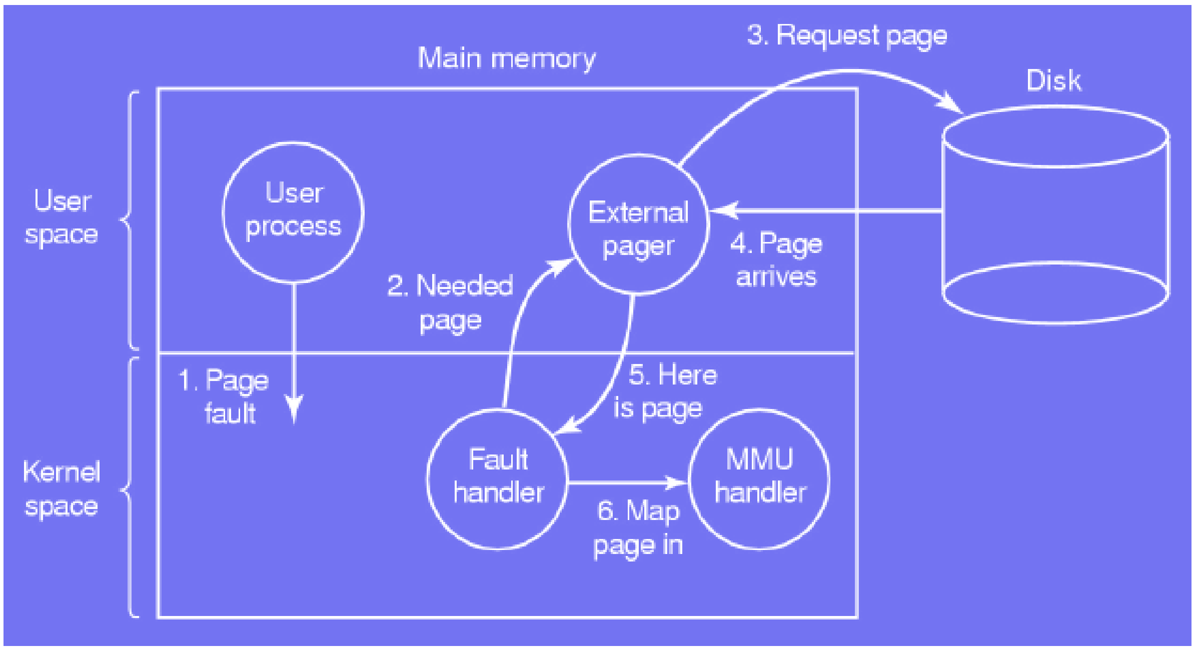
Importance of paging in the OS

OS involved in paging related work on four occasions:

- **Process creation**:
    - Determine process size
    - Create process' page table (allocate and initialize memory)
    - Initialize swap area
    - Store information related to the swap area and page table in the process table
- **Process execution**:
    - MMU resets for the new process
    - Flush the TLB
    - Make the new process' page table the current one
- **Page fault**:
    - Read hardware register to determine origin of page fault
    - Compute which page is needed
    - Locate the page on the disk
    - Find an available page frame and replace its content
    - Read the new page frame
    - Rewind to the faulting instruction and re-execute it
- **Process termination**:
    - Release page table entries, pages, and disk space
    - Beware of any page that could be shared among several processes

Page fault handling

1. **Trap** to the kernel is issued; **program counter** is saved in the stack; **state of current instruction** saved on some specific registers
2. Assembly code routine started: **save** general registers and other **volatile information**
3. OS search which page is requested
4. Once the page is found: check if the **address is valid** and if **process is allowed** to access the page. If not kill the process; otherwise find a free page frame
5. If selected frame is **dirty**: have a **context switch** (faulting process is suspended) until disk transfer has completed. The page frame is marked as reserved such as not to be used by another process
6. When page frame is **clean**: **schedule disk write** to swap in the page. In the meantime the faulting process is suspended and other processes can be scheduled
7. When receiving a **disk interrupt** to indicate copy is done: page table is updated and frame is marked as being in a normal state
8. **Rewind** program to the faulting instruction, program counter **reset** to this value
9. Faulting process scheduled
10. Assembly code routine starts: **reload** registers and other volatile information
11. Process execution can continue



page replacement algorithm in...

User space:

- Need to use some mechanism to access the R and M bits
- Clean solution
- Overhead resulting from crossing user-kernel boundary several times
- Modular code, more flexibility

Kernel space:

- Fault handler sends all information to external pager (which page was selected for removal)
- External pager writes the page to the disk
- No overhead, faster

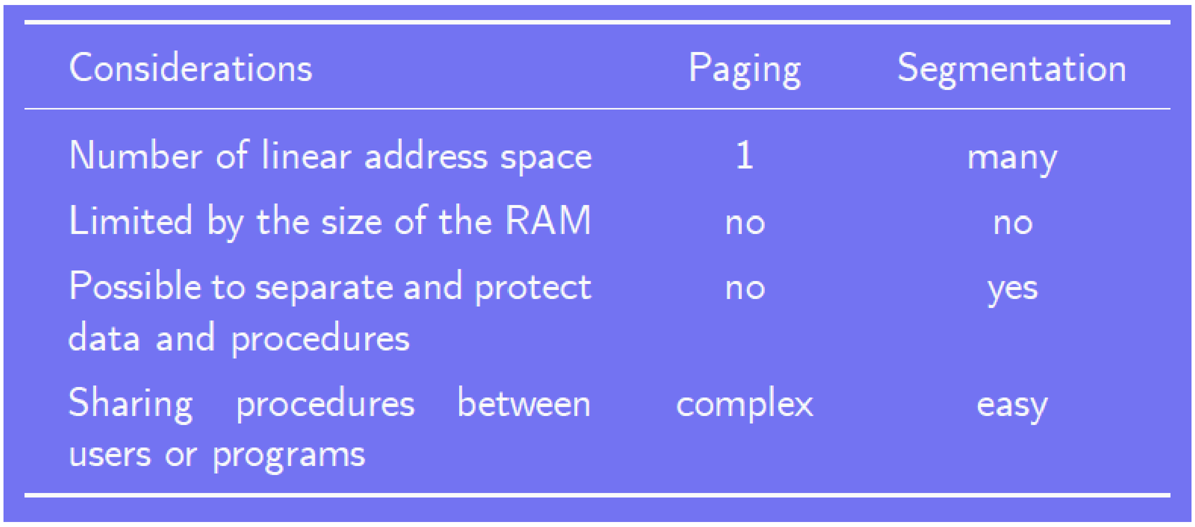
Segmentation

Basic paging is one dimension: address ranges from 0 to MAX.

Some programs may arrange data in different tables in a page. If a table grows bigger, it might overlap other tables, so **segmentation** is raised as a solution is needed to manage this.

Handling segmentation in the OS:

- Each segment contains **a linear sequence of addresses**
- **Length** of different segments may **differ**
- Each segment has a **number** and an **offset**
- **Segment table**: contains the starting physical address of each segment, the base, together with its size, the limit
- Segment table **base register**: points to the segment table
- Segment table **length register**: number of segments used in a program



Considerations	Paging	Segmentation
Number of linear address space	1	many
Limited by the size of the RAM	no	no
Possible to separate and protect data and procedures	no	yes
Sharing procedures between users or programs	complex	easy

Input-Output

Basic hardware

The OS controls all the IO:

- Send commands to the device
- Catch interrupts
- Handle errors

Two main device categories:

- **Block devices**:
  - Stores information in **blocks** of fixed size
  - Can directly **access any block** independently of other ones
- **Character devices**:
  - Delivers or accepts **a stream of characters**
  - **Not addressable**, no block structure
- Some devices can not be categorized into the two categories:
  - **Clock**: cause **interrupts** at some given interval

File system deals just with abstract block devices.

Device controller

Most devices have two parts:

- **Mechanical**: the device itself
- **Electronic**: the part allowing to communicate with the device
  - The electronic part is called the **device controller**:
    - Allows to handle mechanical part in an easier way
    - Performs error corrections for instance in the case of a disk
    - Prepares and assemble blocks of bits in a buffer
    - The blocks are then copied into the memory

Memory-mapped IO

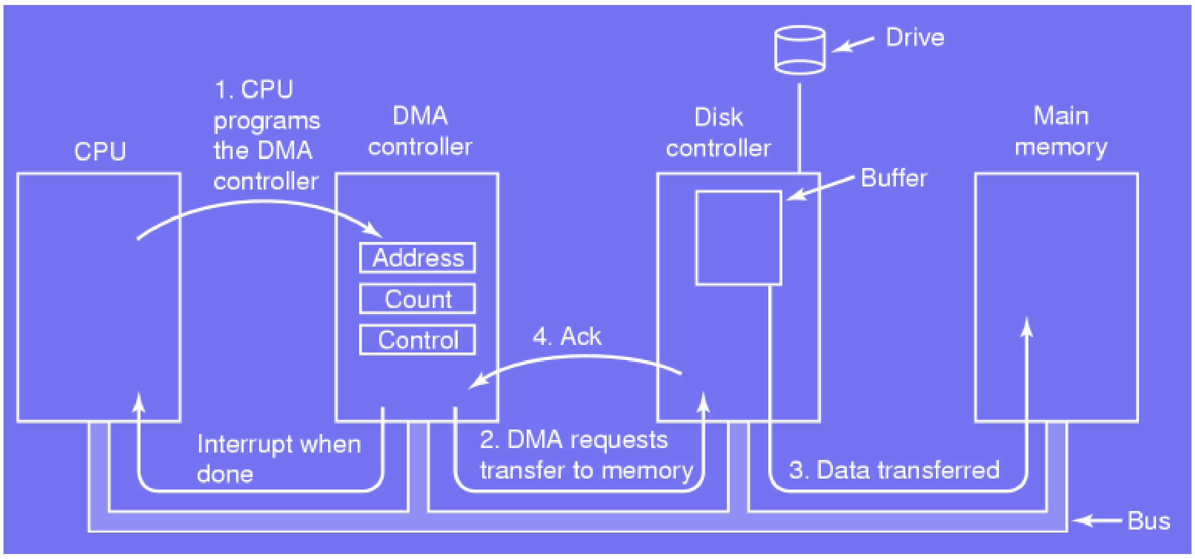
The CPU communicates with the device using **control registers**:

- OS **writes** on registers to: send|accept data, switch device on|off
- OS **reads** from registers to: know device's state

Modern approach (Memory-mapped IO):

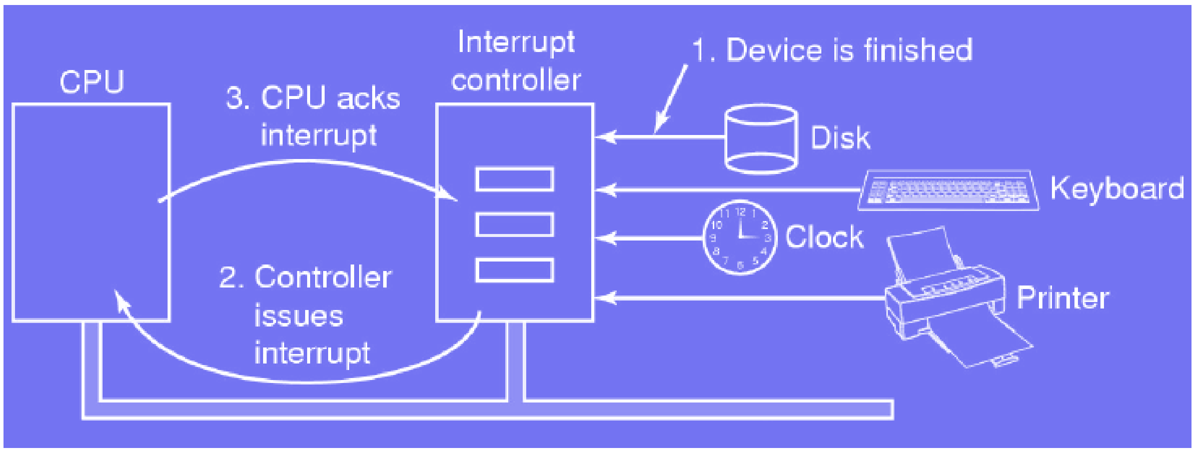
- **Map the buffer** to a memory address
- **Map each register** to a unique memory address or IO port
- Strengths:
    - **Access memory** not hardware: no need for assembly
    - **No special protection required**: control register address space not included in the virtual address space
    - **Flexible**: a specific user can be given access to a particular device
    - **Different drivers in different address spaces**: reduces kernel size and does not incur any interference between drivers
- Drawbacks:
    - Control register should be identified and **not cached**, bringing extra complexity for OS
    - Since there is **only one address space**, contents in memory and I/O devices should both **be examined to ensure which one is needed**. Extra work is needed if there are multiple buses

Direct Memory Access



1. CPU programs the DMA controller (so that it knows where and how much to transfer) and ask disk controller to load to buffer
2. DMA send a read request to disk controller
3. Data on the bus is transferred to main memory
4. Disk controller send an acknowledgement signal to DMA when the write is completed
5. DMA then increments the memory address to use and decreases byte count
6. If count is greater than 0, repeat 2-4.
7. DMA interrupt CPU

Hardware interrupts



Interrupt handling on a basic setup:

- **Push** program counter (PC) and PSW on the stack
- **Handle** the interrupt
- **Retrieve** program counter and PSW from the stack
- **Resume** a process

Precise interrupt's property:

1. Program counter is saved in a known place
2. All instructions before PC have been completed
3. No instruction after the one pointed by PC have be executed (or the executed instructions are rolled back)
4. Execution state of the instruction pointed by PC is known

For pipelined or superscalar CPU, precise interrupt may be not suitable, since property 2&3 cannot be ensured. This kind of interrupt is called **imprecise interrupt**:

1. Instructions near PC are in different stages of completion
2. General state can be recovered if given many details
3. Code to resume process is complex
4. The more details the more memory used

Imprecise interrupts are **slow**. Some computers solve this by making **I/O interrupts** precise while **fatal programming errors** imprecise (such process does not need to be restart)

Software IO

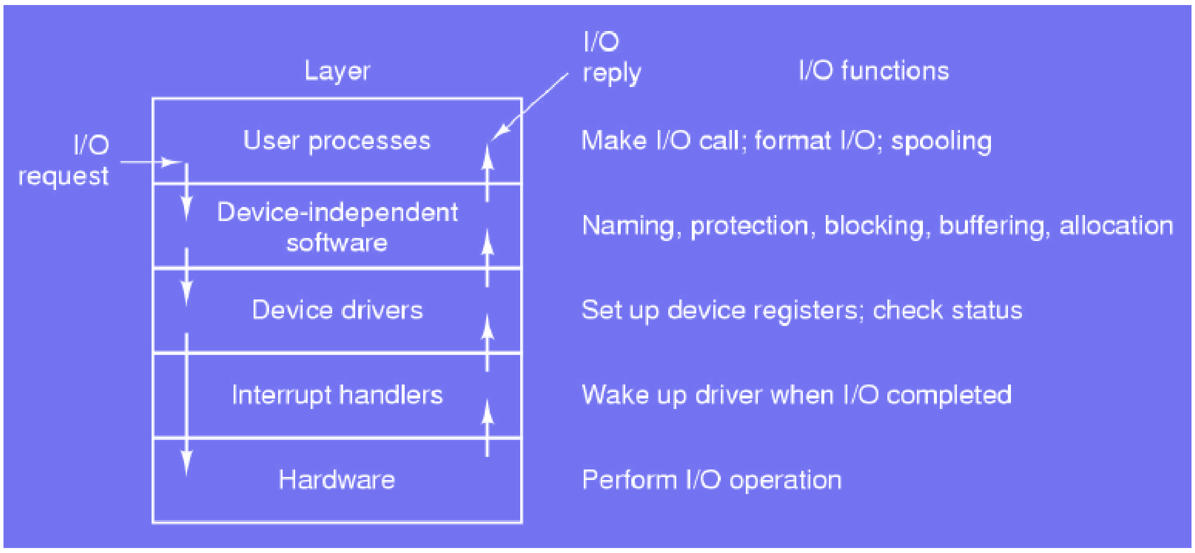
Main goals on the design of IO software:

- **Device independence**: whatever the support, files are handled the same way
- **Uniform naming**: devices organized by type with a name composed of string and number
- **Error handling**: fix error at lowest level possible
- **Synchronous vs. asynchronous**: OS decides if interrupt driven operations look blocking to user programs
- **Buffer**: need some temporary space to store data

communications strategies

- Programmed IO
    - Have the CPU to do all of the work:
        - **Copy** data **into kernel space**
        - Fill up device register and wait in tight loop until register is empty (**busy waiting**)
        - Fill up, wait, fill up, wait, etc.
    - Advantages and drawbacks:
        - Simple
        - Inefficient
- Interrupt IO
    - **Copy** data into kernel space, then fill up device register
    - The current process is **blocked**, the scheduler is called to **let another process run**
    - When the register is empty an **interrupt** is sent, the new current process is stopped
    - Filled up, block, resume, fill up, block, etc.
    - Advantages and drawbacks:
        - This enables CPU to do other things instead of just waiting.
        - Interrupt **waste CPU time**
- DMA
    - DMA act as Programmed IO, but this leaves CPU doing other works.

I/O Software Layers


I/O request
Layer
I/O reply
I/O functions
User processes
Make I/O call; format I/O; spooling
Device-independent software
Naming, protection, blocking, buffering, allocation
Device drivers
Set up device registers; check status
Interrupt handlers
Wake up driver when I/O completed
Hardware
Perform I/O operation

Interrupt handlers

Actions to performs on an interrupt:

1. **Save registers**
2. Setup a **context** for handling the interrupt
3. Setup a **stack**
4. Acknowledge interrupt controller + re-enable interrupts
5. **Load** registers
6. **Extract information** from interrupting device's controller
7. **Choose** a process to run next
8. **Setup MMU** and **TLB** for next process
9. Load new process registers

10. Run new process

Device drivers

Use **plugin design strategy**:

- Similar devices have a <u>common basic set of functionalities</u>
- OS defines which functionalities should be implemented
- Use <u>a table of function pointers</u> to interface device driver with the rest of the OS
- <u>Uniform naming</u> at user level

Basic functions of a driver:

- Initialization
- Accept generic requests, e.g. read, write
- Log events
- Retrieve device status
- Handle device specific errors
- Specific actions depending on the device
- Device driver should **react** nicely even under special circumstances

General remarks on drivers:

- **Location**: user or kernel space (mainly in kernel)
- Drivers can be **compiled in kernel**
- Drivers can be **dynamically loaded** at runtime
- Drivers can **call certain kernel procedures**, e.g. to manage MMU, set timers
- I/O errors framework is device independent
- Clean and generic API such that it is easy to write new drivers

File systems

Goals that need to be achieved:

- Store **large** amount of data
- **Long** term storage
- Information **shared** among multiple processes

High level view of a file-system:

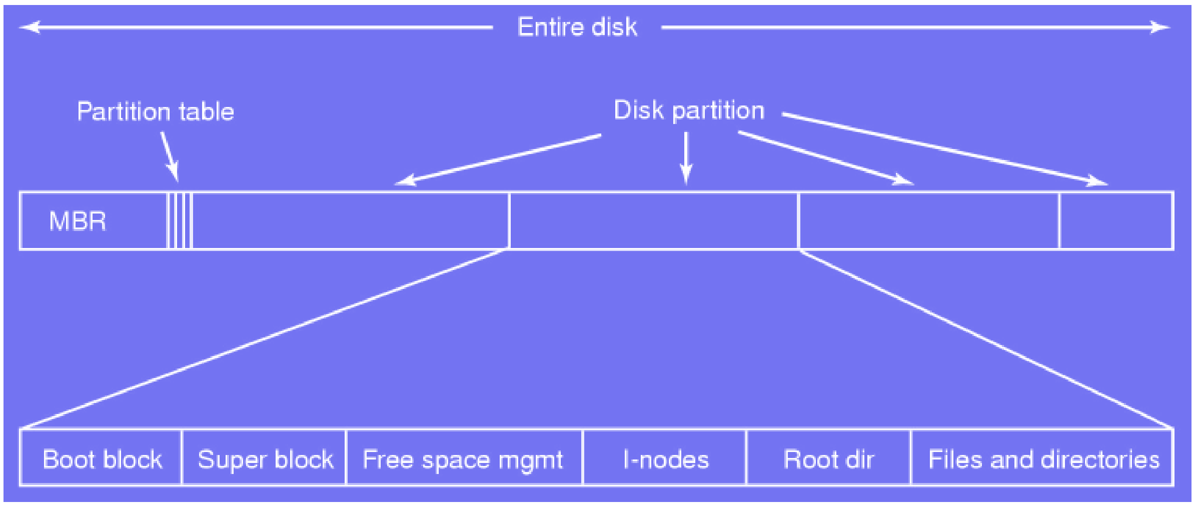
- Small part of the disk memory can be directly accessed using high level **abstraction** called a **file**
- File name can be case sensitive or insensitive
- File name is a string with (an optional) suffix
  - Some OS (e.g. Unix) does not care file extension, but others (e.g. Windows) assign meanings for them.
- Each file has some attributes containing special information

File organization:

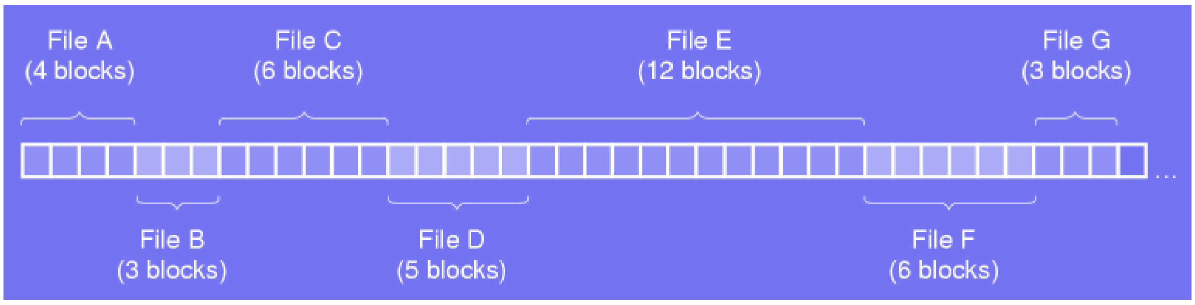
- Files are grouped inside a **directory**
- Directories are organized in a **tree**
- Each file has an **absolute path** from the <u>root</u> of the tree
- Each file has an **relative path** from the <u>current location</u> in the tree

Implementation

Basic disk layout



Contiguous allocation



Store each file as a **contiguous run of disk blocks**.

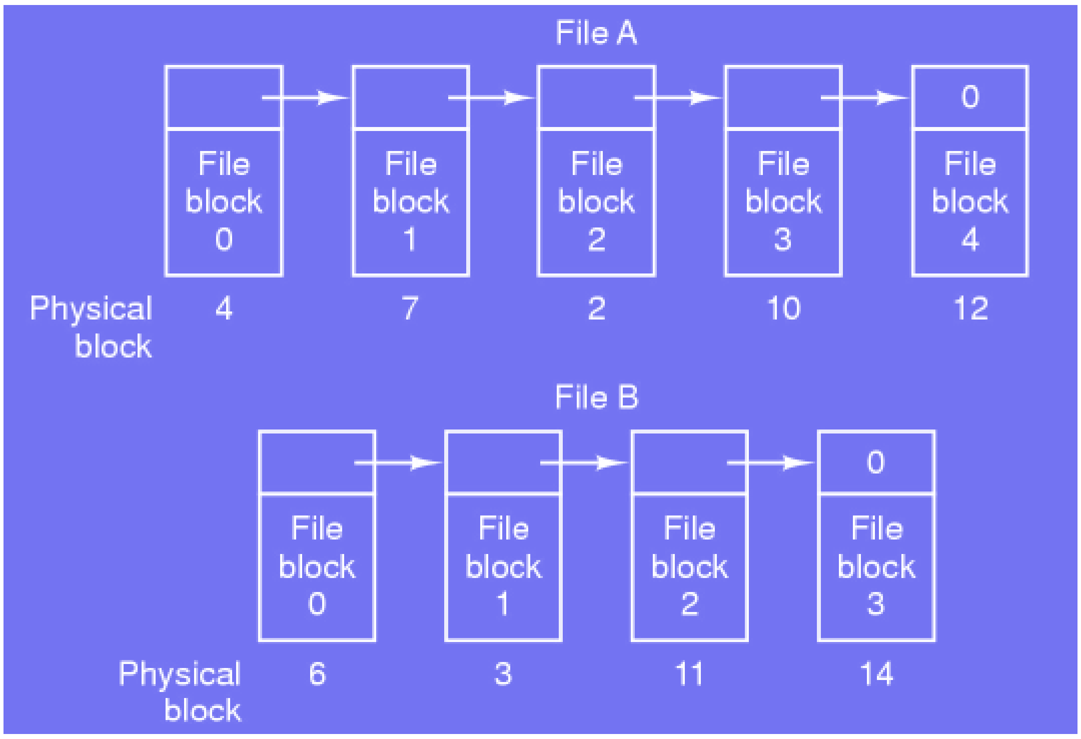
Advantages:

- **Simple** to implement
  - Keeping track of where a file's blocks are is reduced to two numbers: **disk address of the first block** and **the number of blocks in the file**.
- **Fast**: read a file using a **single disk operation**

Drawbacks:

- Fragmented
  - When deleting a file, a hole appears. This will make the disk fragmented.
  - Compact the disk is expensive
  - Reuse holes need to know the file size in advance, which is not realizable.

Linked list



Keep each file in a linked list of disk blocks.

Advantage: **no fragmentation**

Drawbacks:

- **slow random access**
- The amount of data storage in a block is no longer 2^n, since it needs to store the pointer

File Allocation Table

Used to solve the two drawbacks of linked list.

Basic idea:

- Have a pointer for each disk block
- Store all the pointers in a table
- Save the table in the **RAM**
- For a file using disk blocks 4, 7, 2, 10, 12:
  - Start from index 4, and 7 should be stored in the corresponding entry in the table
  - Go to index 7, and 2 should be stored there,...

Advantage:

- **faster random access**, since the table is in RAM, without making disk references.
- The whole block can be used to store data.

Drawbacks:

- A **large table** should be always in the memory.

Index node (inode)

A data structure associated with file, storing:

- The **file attributes**
- Disk addresses of the file's blocks

- The last entry is reserved, pointing to **address of block of pointers for additional disk addresses** appeared when a large file needs more blocks that can fit in an inode.

Advantage:

- requires **little memory**
    - Only the corresponding file is opened, then the inode needs to be in the memory
- **fast**

Implementation for directories

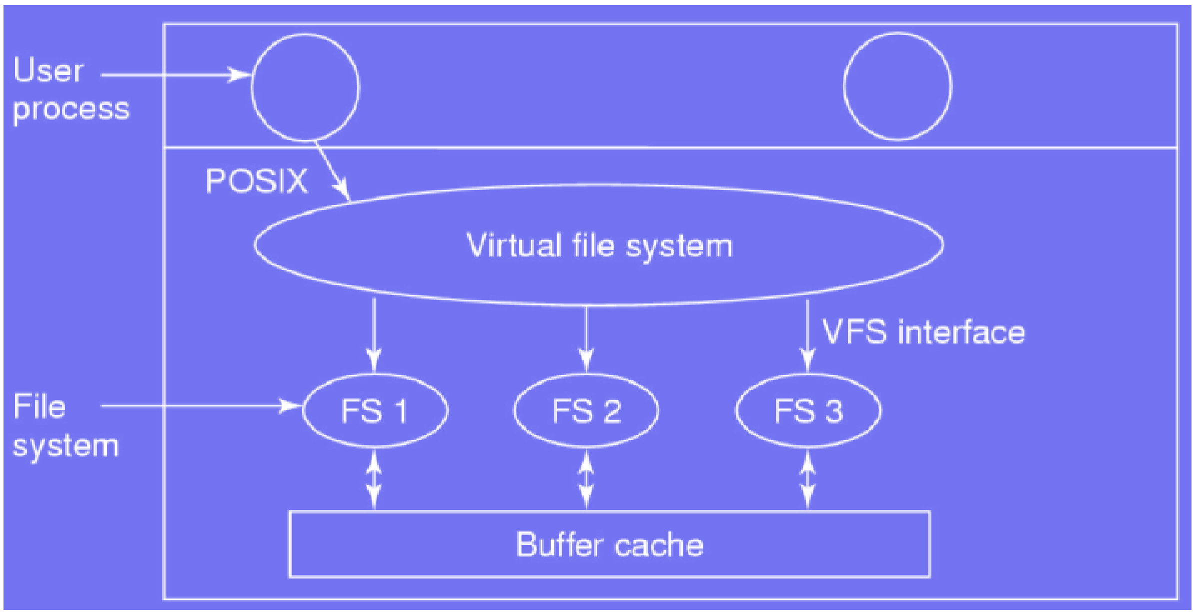
唉看不动了（

Journaling FS

Keep a journal of the operations:

- **Log** the operation to be performed, **run** it, and **erase** the log
- If a **crash** interrupts an operation, **re-run** it on next boot

Virtual FS



Management and optimizations

Block size

Using **small** blocks:

- Large files use **many blocks**
- Blocks are **not contiguous**
- A small block size leads to a **waste of time**

Using **large** blocks:

- Small files do **not fill up** the blocks
- Many blocks **partially empty**
- A large block size leads to a **waste of space**

Keeping track of the free blocks

- Using a **linked list**: free blocks addresses are stored in a block
    - e.g. using 4KB blocks with 64 bits block address, how many free blocks addresses can be stored in a block?
- Using a **bitmap**: one bit corresponds to one free block
- Using **consecutive free blocks**: a starting block and the number of free block following it

File system consistency

Checking an inode based FS:

- Using the inodes: **list all the blocks used** by all the files and **compare** the complementary to the **list of free blocks**
- For every inode in every directory **increment a counter** by 1 and compare those numbers with the **counts stored in the inodes**

Common problems and solutions:

- **Block related inconsistency**:
    - List of free blocks is missing some blocks: add blocks to list
    - Free blocks appear more than once in list: remove duplicates
    - A block is present in several files: copy block and add it to the files
- **File related inconsistency**:
    - Count in inode is higher: set link count to accurate value
    - Count in inode is lower: set link count to accurate value

Caching

Refined idea:

- Useless to cache inode blocks
- Dangerous to cache blocks essential to file system consistency
- Cache **partially full blocks** that are being written

Extra remarks

Quotas: assign disk quotas to users

Fragmentation: how useful is it to defragment a file system?

Block read ahead: when reading block k assume $k + 1$ will soon be needed and ensure its presence in the cache

Logical volumes: file system over several disks

Backups: how to efficiently backup a whole file system?

RAID: Redundant Arrays of Inexpensive Disks

Security

Basics on security

Security setup and dangers

Simple reasoning:

- Security is needed to **protect** from some danger
- If the danger is **unknown** it is **impossible to avoid** it

To define the dangers, the setup must be known:

- General setup: operating system
- Processes: privileges
- Memory: sensitive information processed
- I/O devices: intruders
- File system: sensitive data

Threats

In an OS, **threats** can be divided into **four categories**:

- **Data stolen**: confidentiality
- **Data changed**: integrity
- **Intrusion**: exclusion of outsiders
- **Denial of service**: system availability

Basics on cryptography

Security and cryptography

Cryptography, the science of secret:

- Confidentiality
- Data integrity
- Authentication
- Non-repudiation

Two basic **encryption strategies**:

- **Symmetric**: same key used to encrypt an decrypt
- **Asymmetric**: many can encrypt but only one can decrypt

In an OS setup:

- **Symmetric** protocols best fit **confidentiality**
- **Asymmetric** protocols best fit **authentication**

Data integrity and authentication

Ensure that data has not been altered using hash functions:

- Easy to compute
- Infeasible to generate a message with a given hash
- Infeasible to modify a message without modifying the hash
- Infeasible to find two different messages with same hash

Prove that a user is really who he pretends to be:

- Secret
- Challenge-response

- Token
- Biometrics

Basic mechanisms

Authentication

Most obvious strategy is to **setup a login and password**:

- Password should not be displayed when entered
- Should something be displayed when typing the password?
- When to reject a login: before of after the password input?
- What if the hard disk is mounted from another OS?
- Solutions based on asymmetric cryptography are safer

Access Control Lists

Access control lists are used to give users different privileges:

- **Administrator**: root|admin
- **Privileged users**: belong to special groups
- **Regular users** cannot access IO devices

Keeping a system secure – Basic

- Keep the system **minimal**
- **No new** software versions
- Regularly **update the system**
- Install software only **from trusted parties**
- **Strong passwords** or no password

Keeping a system secure – Advanced

- Apply the basic strategy
- **Filter** any outgoing network traffic
- **Block** any incoming new connection
- Keep a **checksum** of all the files
- Only use **encrypted network traffic**
- Use **containers** or **virtual machines** to run sensitive services
- Associate with each program a profile that **restricts its capabilities**

Keeping a system secure – Paranoiac

- Apply the advanced strategy
- **Encrypt** all the disk, including the swap
- Isolate the computer, **no network** connection
- Keep an **encrypted checksum** of all the files
- Physically block all the ports, **no external device** can be connected

Multiple processors systems

Multiprocessor setup

Shared memory model

Multiprocessors:

- CPUs communicate through the **shared memory**
- Every CPU has **equal access** to entire physical memory
- Access time: 2-10 ns

Three main approaches:

- **Each CPU has its own OS**: no sharing, all independent OS
- **Master-slave multiprocessors**: one CPU handles all the requests
- **Symmetric Multi-Processor**: one OS shared by all CPU

SMP approach

Problems likely to occur:

- More than one CPUs run the same process
- A same free memory page is claimed at the same time

Basic idea of the solution:

- Many parts of the OS are independent
- Split the OS into **multiple critical regions**
- Add a **mutex** when entering those regions
- Add mutex to all shared tables

The solution works but **add complexity** to the OS:

- How to divide up the OS
- Easy to run into deadlock with the shared tables
- Hard to keep consistency between programmers

Synchronization

Synchronization strategies:

- Single CPU:
  - Kernel space: **disable interrupts**
  - User space: take advantage of **atomic operations**, e.g. mutex
- Multiple CPUs:
  - Disabling interrupts on the current CPU is not enough
  - The **memory bus needs to be disabled**

A first idea:

1. **Lock** the memory bus by asserting a special line on the bus
2. The bus is only available to **the processor which locked it**
3. **Perform** the necessary memory accesses
4. **Unlock** the bus

New multiprocessor issue: slows down the whole system

Solution: do not lock the whole bus, but **just the cache**

Cache with multiprocessors

Multiprocessors cache implementation:

- The requesting CPU **reads the lock** and **copies it in its cache**
- As long as the lock is **unchanged** the **cached value can be used**
- When the lock is **released** all the **remote copies are invalidated**
- All other CPUs must **fetch the updated value**

Problem:

- The TSL instruction is used to acquire the lock
- The TSL instruction requires write access
- Any modified cached block must be invalidated
- A whole cache block needs to be recopied each time
- <u>Much traffic is generated just to check the lock</u>

Improved approach:

- Check if the lock is free using a **read**
- If the lock appears to be **free** apply the **TSL** instruction

If two CPUs see the lock being free and apply the TSL instruction does it lead to a race condition?

A safe solution:

- The value returned by the read is only a hint
- **Only one CPU can get the lock**
- The TSL instruction prevents any race condition

More solutions

- **Ethernet binary exponential back-off algorithm**:
    - Do not poll at regular intervals
    - Add a **loop** where <u>waiting time is doubled at each iteration</u>
    - Setup a **maximum waiting time**
- Set a **mutex** for each CPU requesting the lock:
    - When a CPU requests a lock it **attaches itself at the end of the list of CPUs requesting the lock**
    - When the initial lock is released it **frees the lock of the first CPU on the list**
    - The **first CPU** enters the critical region, does its work and releases the lock
    - The following CPU on the list can start its work

Spinning or switching

New perspective:

- On a **uniprocessor**: the time spent on waiting is wasted
- On a **multiprocessor**: one CPU is waiting while another works
- **Switching** is <u>expensive</u> but **looping** is a <u>waste</u>

There is no optimal solution:

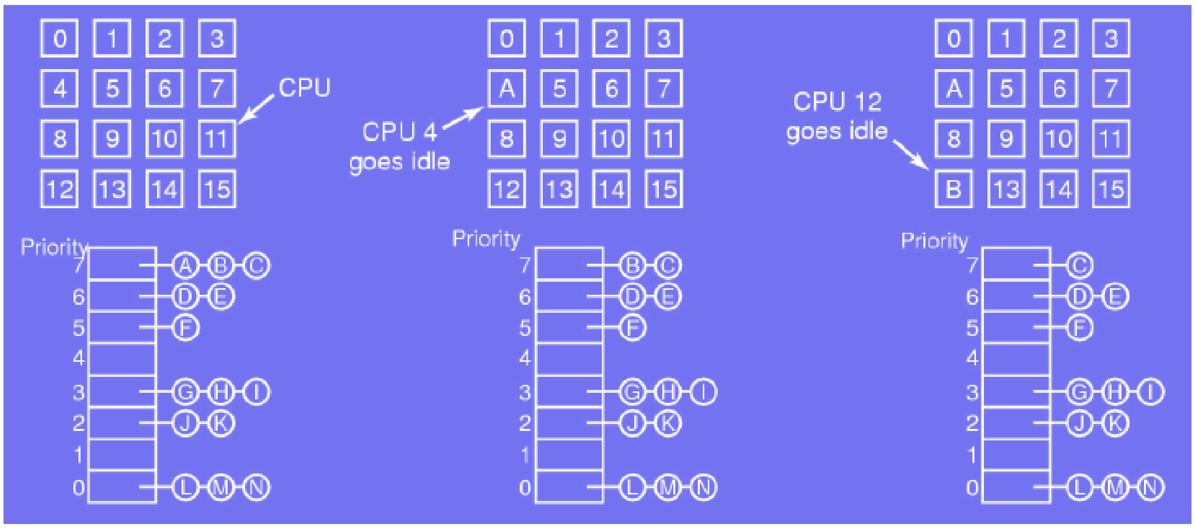
- Only know which solution was best after
- Impossible to have an always accurate optimal decision

Mix of waiting and switching with a (variable) threshold

Multiprocessor scheduling

- Which thread to run?
- Which CPU to run it on?
- Threads of a process run on the **same CPU** → **no need to reload** the whole process
- Threads of a process run **in parallel** → threads can cooperate more **efficiently**

Time-sharing



Advantages:

- Single data structure for ready processes
- Simple and efficient implementation

Improvement:

- **Smart scheduling**:
  - A thread holding a spin lock sets a **flag**
  - Scheduler lets such thread **run after the end of the quantum**
  - **Clear** the flag **when lock is released**
- **Affinity scheduling**:
  - Thread is assigned a CPU when it is created
  - Try as much as possible to **run a thread on the same CPU**
  - **Cache affinity is maximized**

Space-sharing

When a process is created the scheduler checks if there are:

- **More free CPU than threads**: run a thread per CPU until completion
- **More threads than free CPU**: wait for more free CPU

Gang scheduling

Gang scheduling schedules processes by group:

- **Group** related threads into a gang
- All gang members **run simultaneously** on different CPUs
- All members **start and end at the same time**
- No intermediary scheduling decision is taken

Distributed systems

Characteristics of distributed systems:

- Composed of **autonomous entities**, each with its own memory
- Communication is done **over a network** using **message passing**
- The system must **tolerate node failures**
- All the nodes **perform the same task**

Cluster: set of connected computers working together

Setup and challenges

General idea of how a cluster works:

- Computing nodes connected over a LAN
- A **clustering middleware** sits on the top of the node
- Users view a large computer

Example. A single master node handling the scheduling on slave nodes

Main challenges:

- **Scheduling**: where should a job be scheduled?
- **Load balancing**: should a job be rescheduled on another node?

Interprocess communication

Over a distributed system semaphores and mutexes cannot be use

Message passing strategy:

- `send(destination,&message)`
- `receive(source,&message)`: blocks or exit if nothing is received

Potential issues:

- Messages can **get lost**, e.g. sending or acknowledging reception
- **Confusion** on the process names
- **Security**, e.g. authentication, traffic encryption
- **Performance**

More on distributed computing

Advances in network technologies lead to the development of:

- **Volunteer computing**: volunteer offer part of their computational power to some project
- **Grids**: collection of computer resources from multiple locations to reach a common goal
- **Jungle computing**: network not necessarily composed of “regular computers”