UNIVERSITÀ DI TRENTO | Dipartimento di Ingegneria e Scienza dell'Informazione

Dipartimento di Ingegneria e Scienza dell'Informazione

Corso di Laurea in
Ingegneria Informatica, delle Comunicazioni ed Elettronica

Operating System

Docente

Tommaso Zoppi

Studente

Cristiano Berardo 234428

Anno accademico 2023/2024

Contents

Chapter 1

What is Operating System?

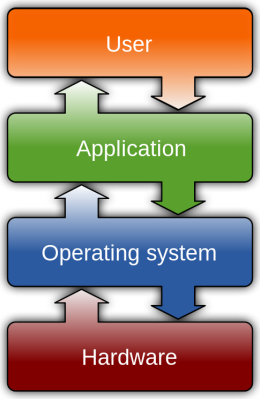
Operating System is a combination of two parts:

- Hardware;
- Software;

Operating System is a program that acts as intermediary between a user of a computer and the computer hardware.
The main goal of OS are:

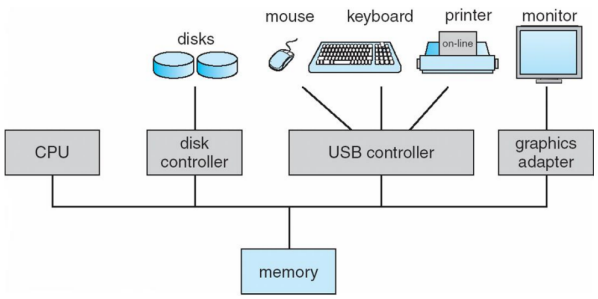
- Execute user program and make solving user problems easier;
- Use the computer hardware in an efficient manner;
- Make the computer system easier to use;

To sum up, an OS provides an environment in which other software can do useful work. A computer System consists on 4 components: HW, SO, Applications and Users. The Operating System coordinates Hardware and Applications. An OS is also a resource allocator (it manages all resources), and a control program (it controls the execution of programs to prevent errors and improper use of the computer).



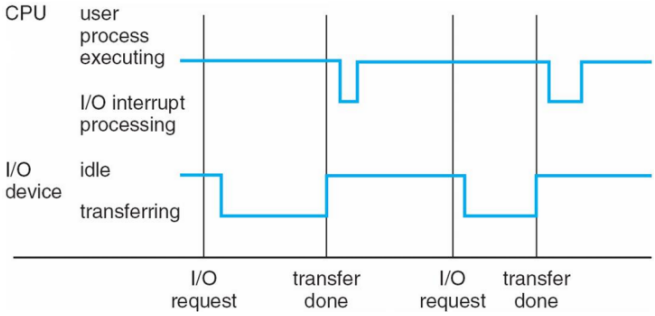
1.1 Computer System

A Computer system when booted up launches a procedure called bootstrap which launches the OS kernel and starts execution.



1.1.1 How do all these things work together?

Device controller informs the CPU that it has finished its operation by causing an interrupt, an event in the OS.



Example: when you try to transfer data, the CPU is notified when the device (may be a pen drive) has finished to transfer all the data through a message of interrupt.

Every interrupt message has a memory address associated. The table that determines all of these addresses is generated at startup. When an interrupt occurs, the CPU saves the current Program Counter (PC) value and jumps to the first instruction of the routine to handle the interrupt, executes the routine and then jumps back to where the PC was. We can say that an OS is interrupt driven.

What if the CPU is doing something very important when an interrupt message arrives? There are two types of interrupt:

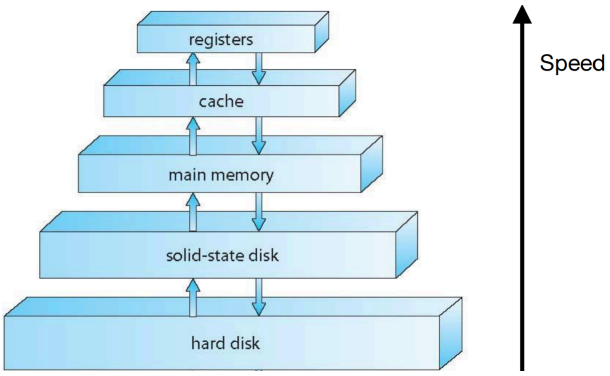
- Maskable interrupts: can be handled whenever critical instructions are being executed;
- Non-maskable interrupts: have high priority and have to be handled right as they arrive;

The structure of interrupt implies that CPU has to manage I/O processes after the transfer of each portion of data has been executed, we don't want that. The DMA (Direct Memory Access) manages the data transfer and notifies the CPU only after everything is finished

1.2 Memory

- **Cache - SRAM**: it is really fast but expensive (1 ns latency). It's the first memory che CPU checks when searching for a piece of information. It is *volatile*;
- **Main memory - DRAM**: large storage media that the CPU can access directly (DRAM), typically volatile (20 ns latency). It is *volatile*;
- **Secondary storage - HDD/Flash NAND**: extensions of the main memory (non volatile) such as HDD and SSD (250 000 ns latency). It is *not volatile*;

Caching: copying information into faster storage system. Main memory can be viewed as a cache for secondary storage. For example, if the CPU finds the information in the main memory, it copies that information into the cache so it can be accessed faster.



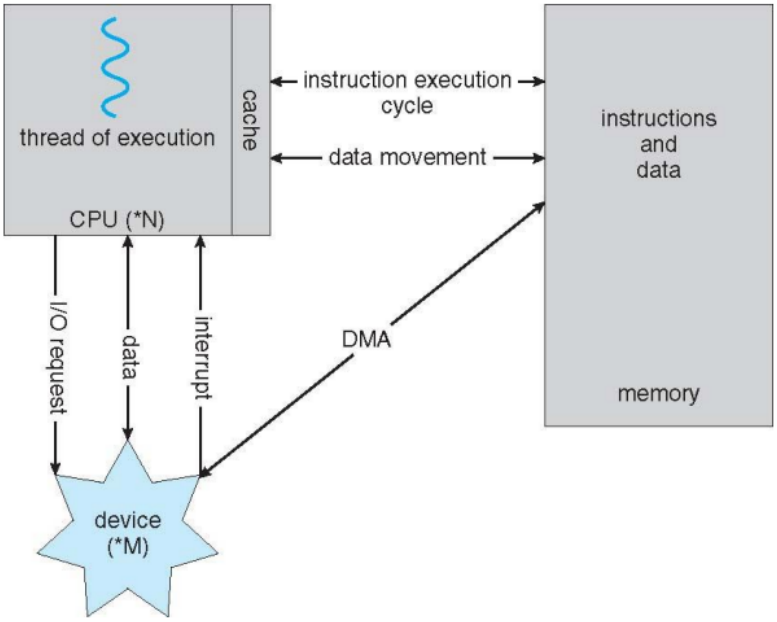
Without main memory and cache every time we have to access data we have to search it into secondary memory, which is much slower.

1.2.1 Performance of various levels of storage

Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

1.3 Modern system architectures: how a computer works

First idea of a computer by Von Neumann is: Device comunicate with CPU and CPU comunicate with memory. This first idea is un-efficient because every input bother the CPU, so now device use the DMA (Direct Memory Access) to bypass CPU to send or to receve data directly to or from the main memory.



1.3.1 Processor

Lot of system use a single general-purpose processor. There are two different type of processors:

- Multi-processor, each processor, with one core, has its own cache and register
- Multi-core, each core in the chip share cache and register with other core

Also there are two different way to work for multiple tasks:

- Asymmetric processing (specific task)
- Symmetric processing (random tasks)

Asymmetric Multi-processor

Each processor, with one core each, is assigned a specific task. After performing the task, the processor waits another specific task and does not help other processors.

Symmetric Multi-processor

Each processor, with one core each, is assigned a random task. After performing the task, the processor helps the other processors and improves the overall performance.

Asymmetric Multi-Core

It is single chip, with several core inside. Each core works on a specific task and after performing the assigned specific tasks waits, without helping the other cores, for other specific tasks.

Symmetric Multi-Core

It is single chip, with several core inside. Each core works on a random tasks and if one or more cores perform the task, the core helps the others, thus improving the overall performance.

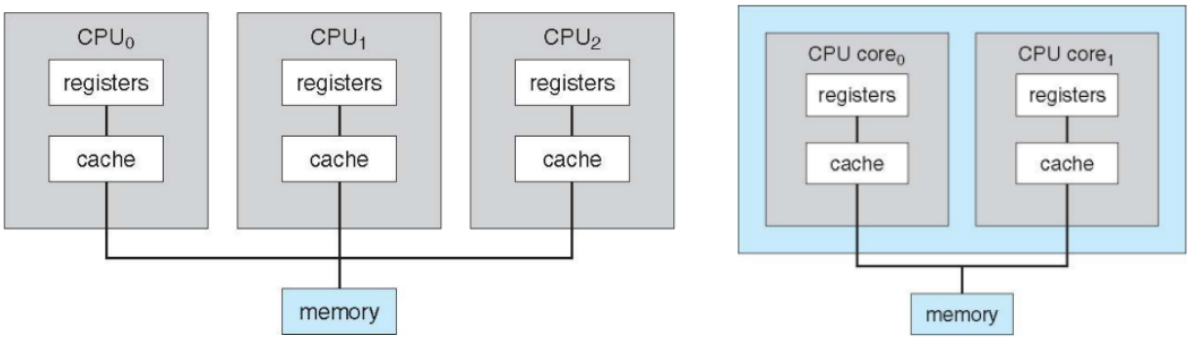


Figure 1.1: Multi-processor on the left; Multi-Core on the right

Is not it the same, why modern system are multi-core?

- On-Chip buses are faster;
- Single chip required less power;
- If one core die you throw the CPU in the bin

1.3.2 High performance computer HCP

If you do not care about power consumption and want a powerful machine because you are Google, Amazon etc., you should consider to build a Cluster. A cluster is group of many computers, even with different OS, working together to complete tasks, such as cloud computing. The application must be written to use parallelization.

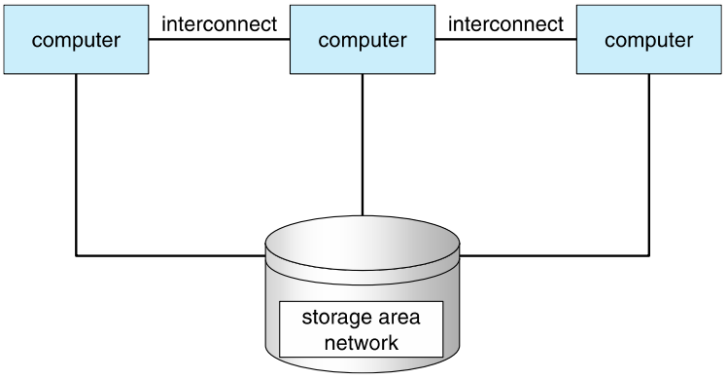


Figure 1.2: SAN

Also Cluster server can chose with type of implementation to use:

- Asymmetric: each machine has a hardware copy of the first machine, and if the first computer brakes the other machine, in hot-standby, starts working.
- Symmetric: has multiple nodes running applications, monitoring each other, if one brakes another machine takes its place.

1.4 Startup

When you turn on the PC lots of things happen:

- The bootstrap routine is invoked
- It stays in a pre-defined portion of the memory (firmware)
- The bootstrap routine initializes registers, memory and device controllers
- Calls the kernel and puts it in the main memory
- Starts daemons i.e., programs not in the kernel but that have to be run at startup like start and initialize driver for the monitor
- Once everything is ready, the OS is ready to provide an environment in which processes can be run
- A process is a program in execution

1.4.1 Improve performance

Executes multiple programs concurrently on a single CPU with one core need to improve efficiency. There are two methods to improve performance: **Multiprogramming** and **Timesharing**

Multiprogramming (Batch system)

Think to have one processor with one core, system can not waste time, like wait for I/O. Multiprogramming organise tasks to do (code and data) so always CPU has something to do. A subset of total jobs in system is kept in memory and when CPU has to wait (I/O), OS switches to another job.

Doesn't actually create the illusion of simultaneous execution for users.

Timesharing (multitasking)

Is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating interactive computing. Users feel like they are interacting with their programs simultaneously, even though the CPU is only working on one program at a time.

We can also decide how to execute programs: by always finishing executing program-N or by starting to work first with the program that requires the shortest execution time, or, like in the previous paragraph, by switching between different tasks.

1.4.2 Switching Mechanism

Multiprogramming (Batch system)

- Relies on I/O events (like waiting for data from disk) to switch between processes.
- When the currently running program needs to wait for I/O, the CPU switches to another program that's ready to run.

Timesharing (multitasking)

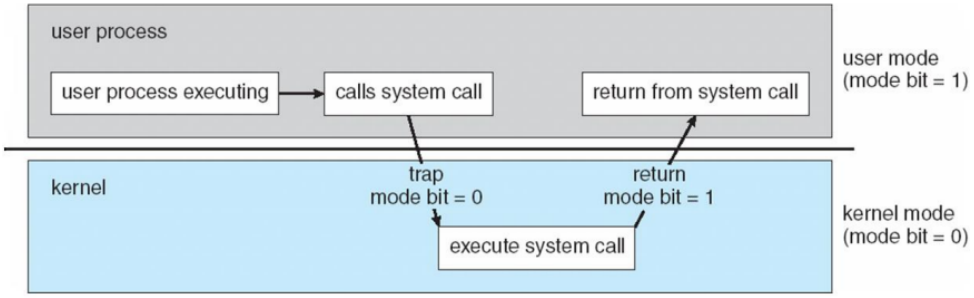
- Uses a time-slicing mechanism to switch between processes.
- The operating system allocates a short time slice (quantum) to each process, and after the quantum expires, it switches to the next ready process, regardless of its need for I/O.

In summary, while both multiprogramming and timesharing improve resource utilization, timesharing adds the concept of time slicing to create the illusion of simultaneous execution for multiple users, making it more user-friendly and interactive.

1.4.3 Security and OS protection

Two different levels to interact with system function:

- User mode (only some privileges decided by company or others admins)
- kernel mode (all privileges)



- **Protection** – any mechanism for controlling access of processes or users to resources defined by the OS
- **Security** – defense of the system against internal and external attacks

System first distinguish users to determinate who can do what using: user IDs and group ID
An attach can threat to:

- Confidentiality: absence of unauthorized disclosure of information
- Availability: service availability (CPU detect and stop malicious process)
- Integrity: absence of improper system alterations

1.4.4 Program counter

- Single-threaded process has one program counter specifying location of next instruction to execute
- Multi-threaded process has one program counter per thread

1.4.5 Storage Management

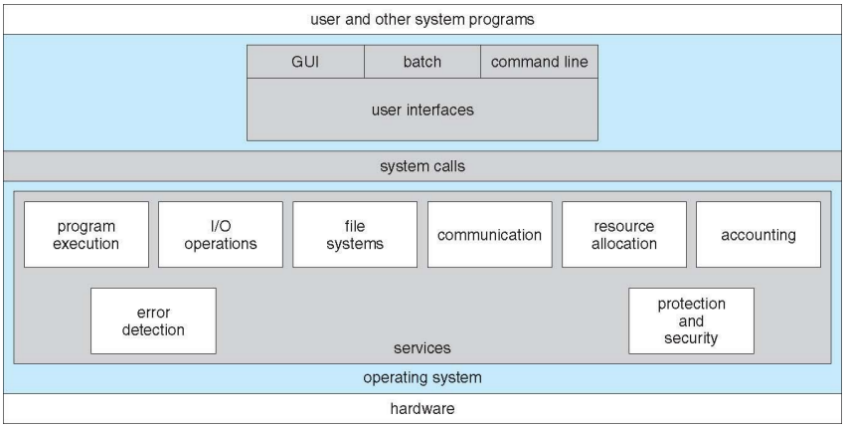
OS provide to manage traffic data from RAM to SSD and vice versa, also OS give the impression that it is organise in directory.

- **Multitasking** environments must be careful to use most recent value, no matter where it is stored in the storage hierarchy
- **Multiprocessor** environment must provide cache coherency in hardware such that all CPUs have the most recent value in their cache

Chapter 2

Operating-System Structures

OS has the task of interconnecting HW and SW. It provides function that are helpful to the user: system call, services, user interface, I/O, file system, network interface, error detection, resource allocation, security and so on.



2.1 Operating System Services

2.1.1 (G)UI

CLI or command line interface allows detect command entry. The basic CLI in windows is Command Prompt, also known as cmd.exe or cmd. CLI existing in each OS (Windows, Max, Linux and Solaris etc.).

Nowadays the UI is more user-friendly tanks the graphics, programme icons, button, directory, files etc. Even if OS now include GUI, they also have the old CLI.

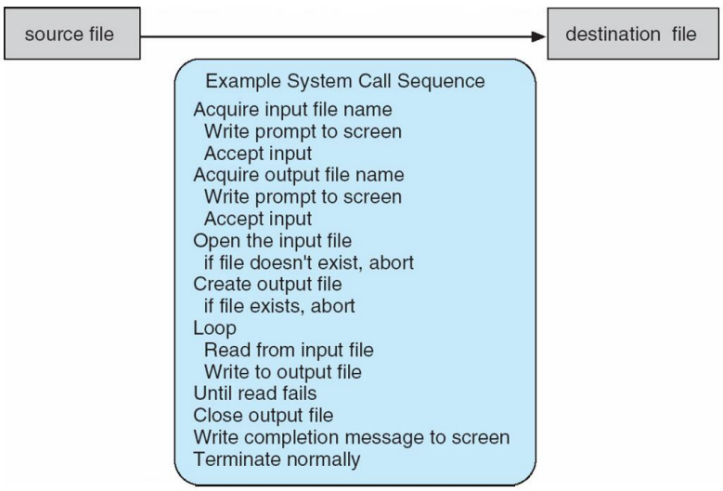
2.2 System Calls

System Calls are typically written in high-level language (C, C++). Most libraries include specific functions that allows programmers to call the operating system call.

Most common API are:

- Win32 API for Windows
- POSIX API for POSIX-based systems (UNIX, Linux and Max OS X)
- Java API for the Java virtual machine (JVM)

Example of System call to copy the contents of one file to another file:



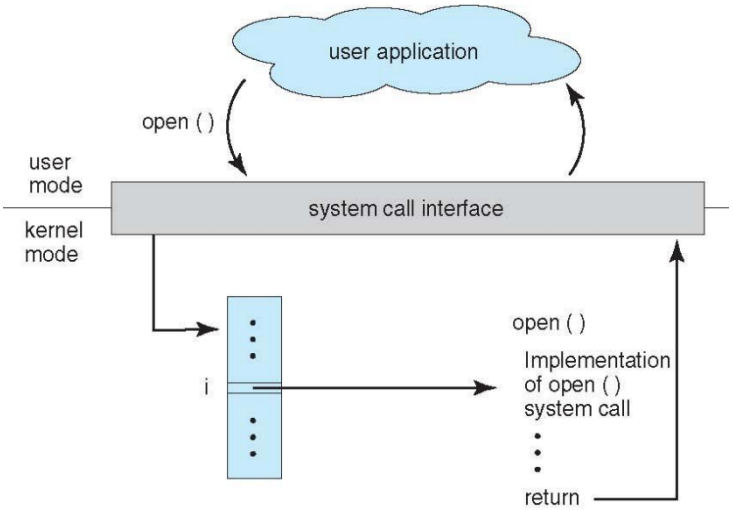
Example of API:

```
#include<unistd.h>
ssize_t read(int fd, void *buf, size_t count)
```

But why do not programmers call the operating system call themselves instead of calling the library function?

The answar is **portability**. Programmer do not care about the implementation and the name of OS-call. So we shift the problems to the libraries. Different libraries (for different OS) implement the right system call.

Another answar is **Error**. If i call an OS-call with different type, e.g. an integer instead of character parameter, I bother the OS and the CPU because OS throw an error. Although if I call a function inside the program language library, it checks if all is right and forwards the request to the OS.



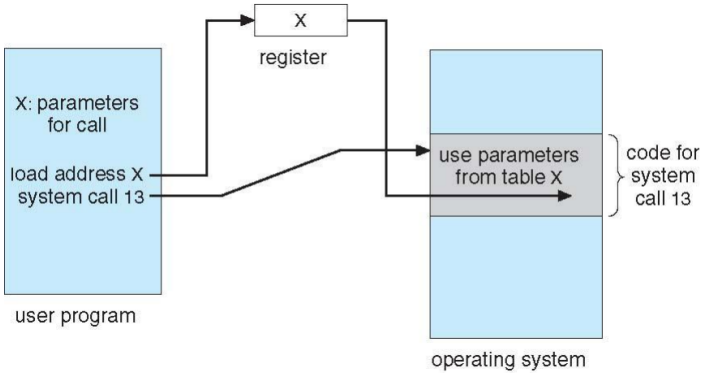
2.2.1 System call parameter passing

There are three way to pass the parameters to OS-call:

– The **firth**, the easiest way, is to copy all parameters into the registers but not always this method works. Normally the registers are less then the parameters.

– The **second** way is to copy all parameters into a table or block and storing it in the ram and loading on register address of the cell where parameters are located.
This approach is used by Linux and Solaris OS.

– The **last** approach is: Parameters placed, or pushed, onto the stack by the program and popped off the stack by the operating system.



2.2.2 Types of system call

- Process control
  - create process, terminate process
  - end, abort
  - load, execute
  - wait for time
  - wait event, signal event
- File management
  - open, close
  - create file, delete file
  - read, write, append
- Device management
  - read, write, reposition
  - request device, release device
  - logically attach or detach devices
- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
- Communications
  - create, delete communication connection
  - Shared-memory
  - transfer status information
- Protection
  - Control access to resources
  - Get and set permissions
  - Allow and deny user access

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

Figure 2.1: Windows and Unix System Call

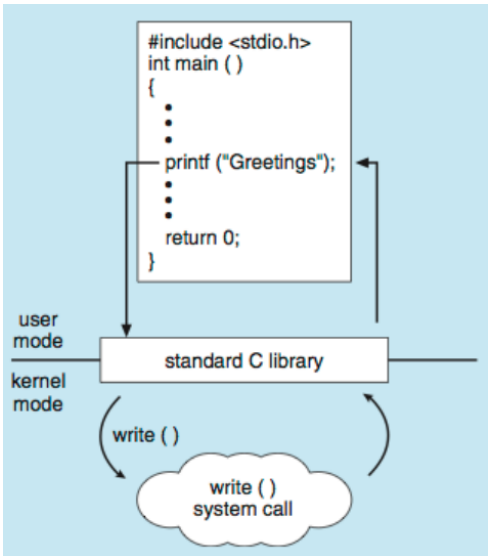


Figure 2.2: Standard C library

2.2.3 MS-DOS

The first OS (1982-2000) can run only single program, to open new program user must reload shell.

2.2.4 FreeBSD

First multitasking OS. Shell can open more then one program at the same time.

2.3 Operating System Design and Implementation

There aren't the best way to create an OS, it depends to different things:

- HW
- SW
- Updating
- Usability...
- **User Goals:** easy to learn, reliable, safe and fast.
- **System goals:** operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient

Some of this things conflict with the each other. GUI is more user-friendly but is more complex to implement GUI instead of CLI.

Some important principle to separate:

- Policy: What will be done? policies decide what will be done
- Mechanism: How to do it? Mechanisms determine how to do something

Example:

Problem: exercising many processes

Mechanism: Scheduling

Policy: processes that use disk first, processes from a given user first, daemons first

2.4 Different kernel implementation

There are different approach to implement the kernel in modern operating system; let's have a look of most of them.

2.4.1 Monolithic Kernel - UNIX

Original UNIX Kernel had a monolithic kernel structure splitted in two parts:

- System programs
- The Kernel: Consists of everything below the OS-call interface and above the physical HW. Also it provides file system, other OS-function.

– The pro of this approach: fast and energy efficient.
– The cons: is not modulable, thus even small update requires recompilation from scratch and re-installation; no option to customize

This approach is the most used for OS kernel: linux use it and also android.

2.4.2 Layered Approach

OS is divided into a number of layers each with specific task built on top of lower layers.

The **bottom** layer is the HW. The highest is the user interface.

This is more modular than monolithic approach. If you want to add or update some function in N-layer you just do it and recompile only N-layer and not all kernel.

Another aspect of this method, that remember TCP/IP stack, is that each level can only communicate with the lower level.

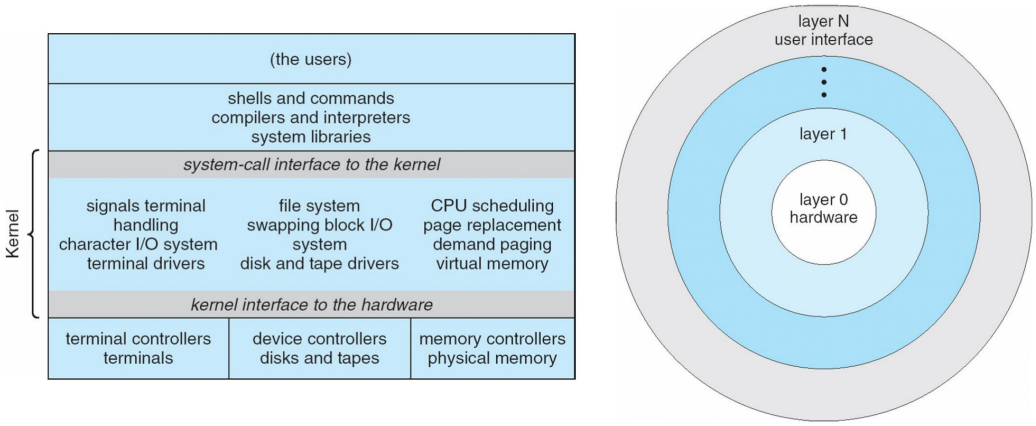


Figure 2.3: Monolithic kernel left, Layered kernel right

2.4.3 MicroKernel System Structure

MicroKernel is the tecnique that allow the most modular kernel. **Mach** is a particular example of microkernel, Max OS X kernel (Darwind) is based on Mach.

Obviously ther are pro and cons:

Pro: Easy to extend (best **modulability**); **portability** from OS to another with new architectures; **reliable** less cose running in kernel mode; **secure** malicious users process cant damage others.

Cons: Worst performance, lots of overhead is not direct communication because app must send a message to basic function of kernel.

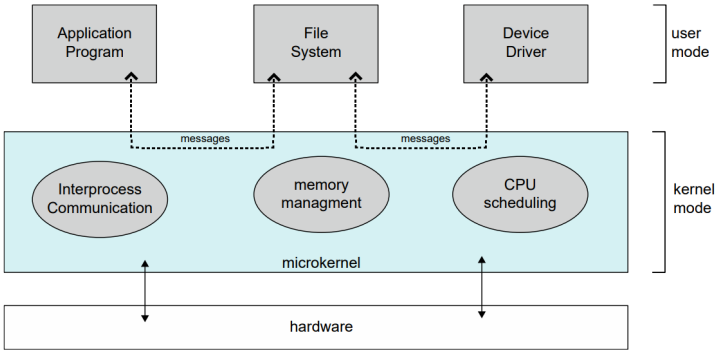


Figure 2.4: micro kernel

2.4.4 Solaris Modular Approach

Solaris is OS used in business and/or commercial area that heavily use Oracle applications (Oracle DB, Java).

2.4.5 Hybrid System

Modern OS is a blended of monolithic, layer and microkernel features. For example Windows is mostly monolithic but also it use microkernel structure. Also Mac OS X and Linux have a mix of different kernel structure.

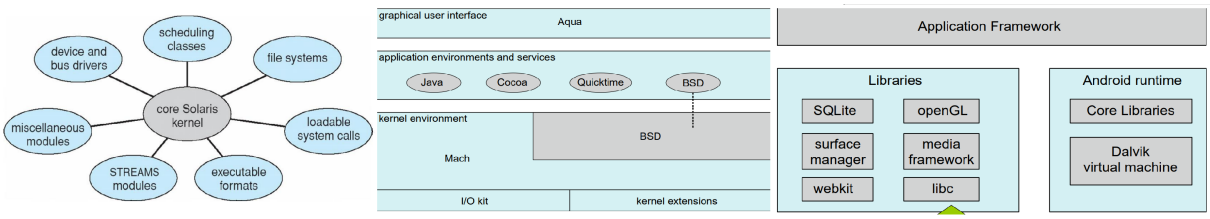


Figure 2.5: Solaris left; Mac OS X center; Android right

Chapter 3

Process

3.1 Process Concept

Process is a program in execution. Process execution must progress in sequential fashion, there are not parallel execution of instructions of a single process.

Program is **passive** entity stored on disk (executable file), process is **active** so program becomes process when an executable file is loaded into memory.
One program can be has several process, think about multiple users executing.

An OS executes a variety of programs:

- Batch system – jobs
- Time-shared systems – user programs or tasks

Process has multiple parts:

- The program code, called text section
- Current activity including PC, processor register
- Stack containing temporary data (function params, return addresses, local variables)
- Data section containing global variables
- heap containing memory dynamically allocated during run time

Memory layout of a C program

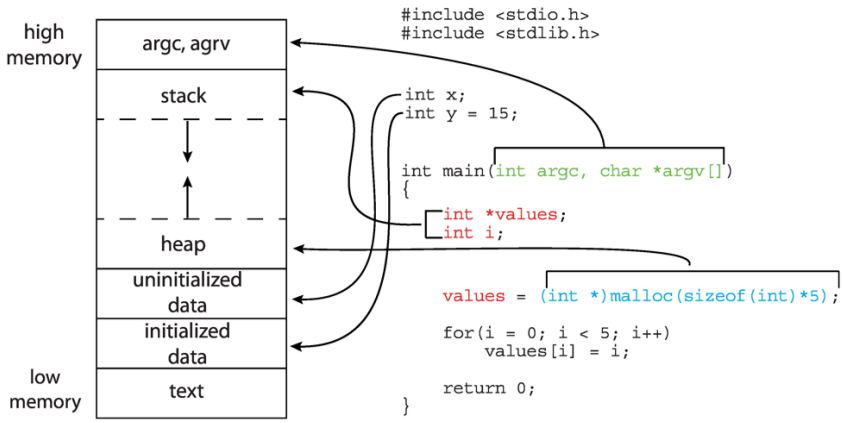


Figure 3.1: Relation to C program and memory usage

We see that the heap grows upwards and the stack vice versa, this structure is used for efficiency.

3.1.1 Process State

Process during this life change it states:

- **New:** The process is being created
- **Ready:** The process is waiting to be assigned to a processor
- **Running:** Instruction are being executed
- **Waiting:** Process waiting for some event to occur (mouse click, I/O, interrupt)
- **Terminated:** The process has finished execution

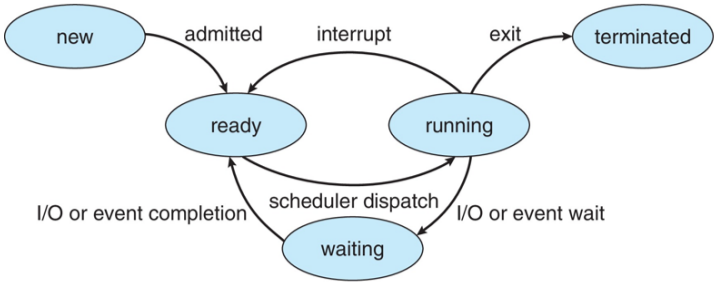


Figure 3.2: Diagram of Process State

3.2 Process Control Block (PCB)

All process must have information about the state, this block of information is called PCB and contains:

- Process state - running, waiting...
- Program counter - location to the next code to execute
- CPU register
- CPU scheduling info - priorities, scheduling queue
- Memory-management info - memory allocated to the process
- Accounting info - CPU usage, time since start, time limits
- I/O status info - list of open files

All of this information are used because when i resume the process i want the same data loaded into register and cache.

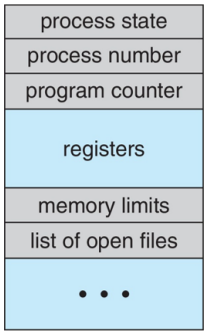


Figure 3.3: Process Control Block (PCB)

In Ubuntu we see PCBs in the folder: `cat /proc/self/status` in this way we see the PCB of cat call, if we replace self with 1 (PID), we see the PCB of systemd. Or using top command to see all Process ID.

3.2.1 Context Switching

When process require to wait, CPU stops to work on it an pass to another task. When CPU switch to another process, the system must save the state of the old process and load the salved state for the new process via **context switch**. The context is represented in the PCB.

Context switching time is pure overhead; the system does no useful work while switching, it is just a waste of time. More complex are OS and PCB and more longer context switch.

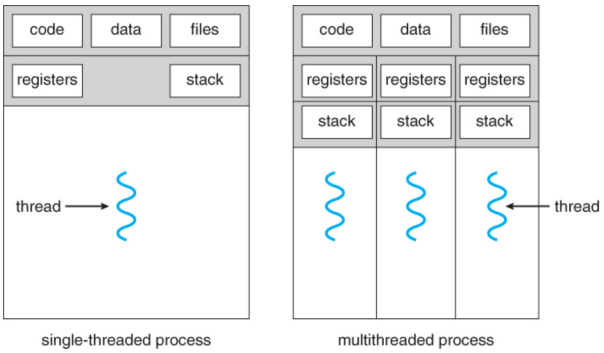
3.3 Threads

First of all threads are different from process, because if P1 run only one task it is a single thread process, otherwise is P1 run more tasks it becomes multi-thread process.

So far, a process has a single "script" of function, but now I want to code a program to execute in parallel to implement parallel search ecc.

How can I do it?

Each thread has a PC assigned, thus a process has multiple program counter (Multiple locations can execute at once).



3.3.1 Multitasking in Mobile System

Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended.

IOS:

- Single foreground process- controlled via user interface
- Multiple background processes– in memory, running, but not on the display, and with limits
- Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback

Android runs foreground and background, with fewre limits:

- Background process uses a service to perform tasks
- Service can keep running even if background process is suspended
- Service has no user interface, small memory use

3.4 Process Scheduling

Process scheduler selects among available process for next execution on CPU core, the main goal of this process is to maximize the use of CPU, quickly switching onto CPU core. Maintains updated two queue:

- Ready queue: for all process that are ready to being execute
- Wait queues: for process that wait for some events (I/O)

Process normally migrate from one queue to the other.

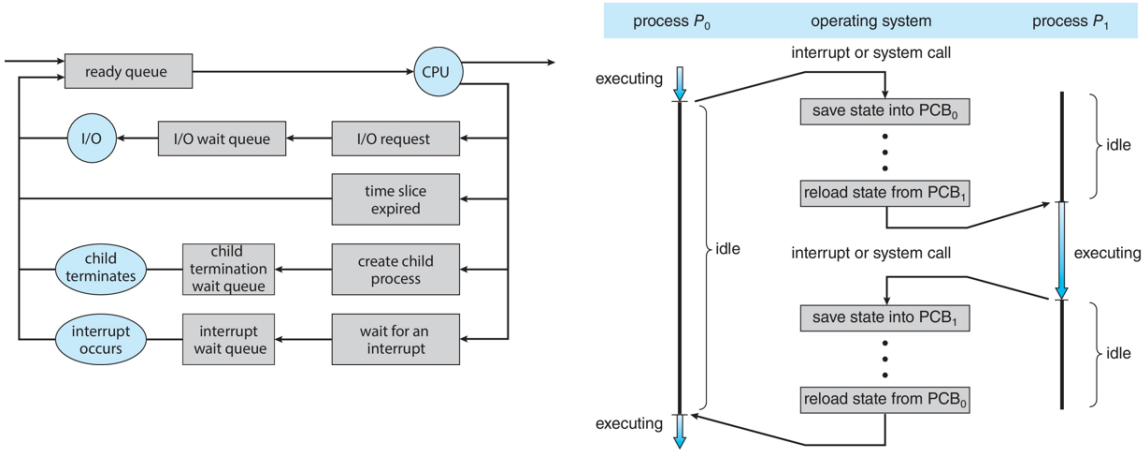


Figure 3.4: Process scheduling left; CPU switching right

3.5 Process Creation:

Parent process create children process, witch can create other processes, forming a tree of processes.

Each process is **identified** by a process identifier: **pid** and they also has **resources** that could be shared by sharing option:

- Parent and children share all resources
- Children share subset of parent's resources, like C++ Inheritance
- Parent and child share no resources

Processes can decide how to **execute**: parent and children execute concurrently or parent waits until children terminate.

Address space:

- Child duplicate of parent
- Child has a program loaded into it

UNIX example of OS-call:

- **fork()** : to create new process, return a pid_t of children created
- **exec()** : used after fork() to replace the process' memory space with new program
- **wait()** : used by parent to waiting for the child to terminate.

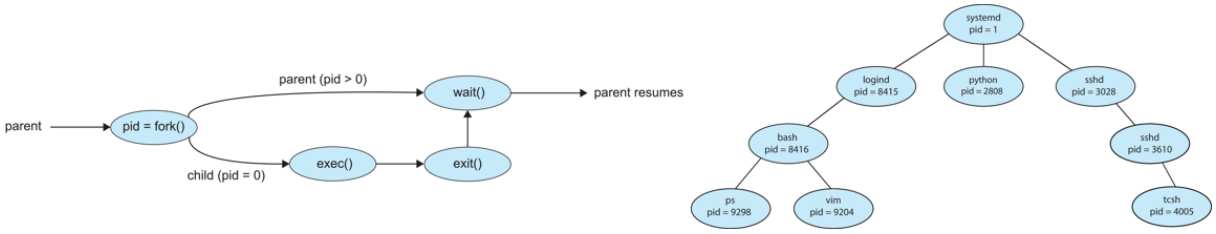


Figure 3.5: OS-call left; process tree in Linux, *pstree command*

C program forking separate process

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main() {

  pid_t pid;

  /* fork a child process */
  pid = fork();

  if (pid < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed");
    return 1;
  } else if (pid == 0) { /* child process */
    execlp("/bin/ls", "ls", NULL);

  } else { /* parent process */

    /* parent will wait for the child to complete */
    wait(NULL);
    printf("Child Complete");
  }

  return 0;
}
```

3.6 Process termination

Each process can decide to terminate the task at any time. Process usually executes last statement and than asks to the OS to delete it using **exit()** call.

All children's status returns to parent (via **wait()** call) and process' resources are deallocated by OS.

In some cases parent can use the **abort()** call to terminate the execution of children. Some reasons of doing it:

- Children has exceeded allocated resources
- The task assigned to children is no longer required
- OS call exit() call to the parent process and children must terminate to do the assigned task because OS does not allow to continue if its parent terminates.

Some OS wait until all the process terminates' children terminates, it called cascading termination. The parent process may wait for termination of a child process by using **wait() syscall**. The call returns status, information and pid of the terminated process.

```
pid  = wait(&status);
```

If no parent waiting (did not invoke wait()) process is a **zombie**.

If parent terminated without invoking wait(), process is an **orphan**.

3.7 Android process termination importance hierarchy

All mobile smartphones have limited resources such as memory compare to PCs. Thus, the android OS has to manage process termination to improve performance. Below there is a list of process, from most to least important:

- Foreground process;
- Visible process;
- Service process;
- Background process;
- Empty process;

Android terminates less important processes weed out unused memory.

3.8 Inter-process Communication

Process within a system may be **independent** or **cooperating**. Cooperating process can affect or be affected by other processes, including sharing data.

Cooperating processes need **interprocess communication IPC**, obviously there are reasons for cooperating process:

- information sharing;
- computation speedup;
- modularity;
- convenience;

Also there are two different way to implement IPC:

- Shared memory: you define a specific space into ram where two process can access and communicate
- Message passing

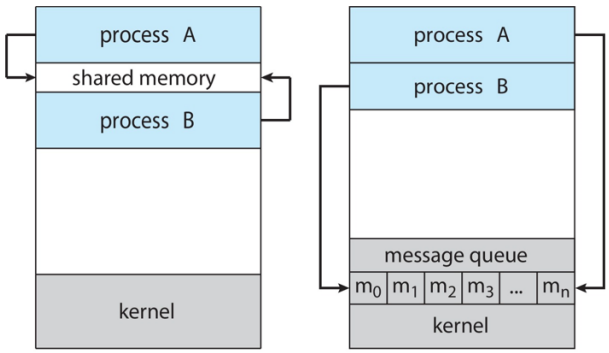


Figure 3.6: shared memory left; message passing right

3.8.1 Producer-consumer problem

Producer process produces information that is consumed by consumer process, two variations:

- unbounded-buffer places no practical limit on the size of buffer:

  Producer never waits

  consumer waits if there is no buffer to consume

- bounded-buffer assumes that there is a fixed buffer size:

  Producer must wait if all buffer are full

  Consumer waits if there is no buffer to consume

3.8.2 IPC - Message Passing

Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.

Example of IPC - POSIX

POSIX Shared Memory: Process P1 create shared memory segment, an area of RAM memory.

```
1 shm_fd = shm_open(name, O CREAT | O RDWR, 0666);
```

and set the size of the existing segment:

```
1 ftruncate(shm_fd, 4096);
```

In this way you create an area of memory and you must use it like a file, calling the default call for file descriptor: open, read, write, close, seek...

Memory-map

If you use the function *mmap()*, you transform the file in array of bytes. You just modified the access to the byets not the content. Using this method you also can access to virtual memory a concept that we will see later.

IPC POSIX Producer

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

#include <string.h>
#include <fcntl.h>

#include <sys/shm.h>
#include <sys/stat.h>

int main() {

    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;

    /* name of the shared memory object */
    const char *name = "OS";

    /* strings written to shared memory */
    const char *message0 = "Hello";
    const char *message1 = "World!";

    /* shared memory file descriptor */
    int shm_fd;

    /* pointer to shared memory object */
    void *ptr;

    /* create the shared memory object */
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

    /* write to the shared memory object */
    sprintf(ptr, "%s", message0);
    ptr += strlen(message0);  // you must do it otherwise you rewrite
    the bytes
    sprintf(ptr, "%s", message1);
    ptr += strlen(message1);

    return 0;
}
```

IPC POSIX Consumer

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

#include <fcntl.h>
#include <sys/shm.h>

#include <sys/stat.h>

int main() {

    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;

    /* name of the shared memory object */
    const char *name = "OS";

    /* shared memory file descriptor */
    int shm_fd;

    /* pointer to shared memory object */
    void *ptr;

    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

    /* read from the shared memory object */
    printf("%s", (char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);  //like close(), you free the memory to OS.

    return 0;
}
```

The only way to get access the S.M. is to know the name of the shared memory object, so it must be **secret**.

3.8.3 IPC - Message Passing

Processes communicate with each other without resorting to shared variables, thus IPC provide two option: **send** and **receive** massage. The size of message is fixed or variable.

If process P and Q wish to communicate, they need:

- Establish a communication link between them
- Exchange messages via send and receive

How to implement this communication? How to establish the link? The link is connected with one ore more processes? Capacity of the link?

3.8.4 Direct Communication

Processes must name each other explicitly:

- send(P, msg) - send a message to process P
- receive(Q, msg) - receive a message from process Q

The proprieties of communication link are: links are established automatically, bi-directional and between each pair exist only one link.

3.8.5 Indirect communication

Indirect communication is different as Direct communication because it uses a mailbox to communicate.

Each mailbox has a unique id and process can communicate only if they share a mailbox.

The proprieties of communication link are: links are established only if processes share a common mailbox, link could be associated with several processes and each process may share several communication links, links may be unidirectional or bi-directional.

- send(A, msg) - send a message to mailbox A
- receive(A, msg) - receive a message from mailbox A

In this way there is a problem. If P1, P2 and P3 share mailbox A and P1 send a message, who receive the message?

Some solutions could be:

- Create link associated only a two processes (like direct communication)
- Allow only one process at a time to execute a receive operation
- Allow the system to select arbitrarily the receiver

3.8.6 Synchronization

The messages could be blocking or non-blocking message:

Blocking is considered synchronous

- **Blocking send** - the sender is blocked until the message is received
- **Blocking receive** - the receiver is blocked until the message is available

Non-blocking is considered asynchronous

- **Non-blocking send** - the sender sends the message and continue
- **Non-blocking receive** - the receiver receives a valid message or null message

Different combinations are possible but if both, sender and receiver, are blocked we have a **rendezvous**.

3.8.7 IPC Systems - Windows

The message passing in Windows pass throw **advanced local procedure call**:

- Only works between processes on the same system
- Uses ports (like mailbox) to established and maintain communication link

Communication works as follows:

1. The client open a handle to the subsystem's connection port object
2. the client sends a connection request
3. the server create two private communication ports and return the handle to the client
4. the client and server use corresponding port handle to send messages and/or callback a to listen for replies

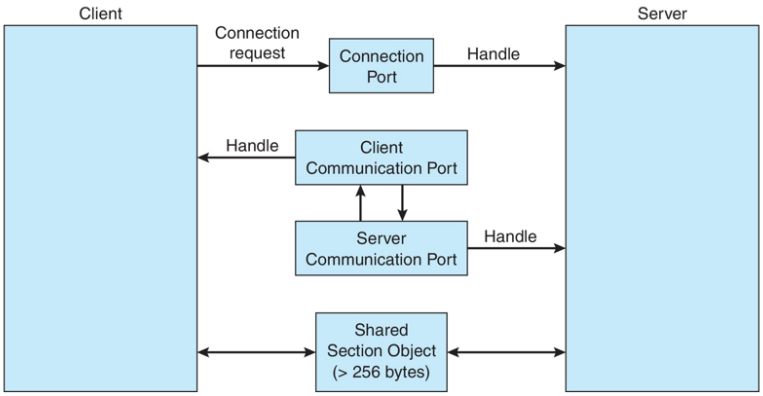


Figure 3.7: Message passing via advanced Local Call LPO

3.9 Pipes

Pipes acts as a conduit allowing to processes to communicate. There are some issues like: are uni or bi directional and in case of two-way communication it is half or full duplex?

There are two different type of pipes:

- **Ordinary pipes** - the pipe cannot be accessed from outside the process that created it. Typical example might be: parent create a process and want to communicate with child that it created.
- **Named pipes** - can be accessed without a parent-child relationship

To create a pipe you must call the *mkfifo("name", 0666)*

3.9.1 Ordinary pipes

This ordinary pipe allow to communicate only if you have a parent-child relationship.

Producer write in one end (write-end) and the consumer read from the another end (read-end). Ordinary pipes are therefore unidirectional.

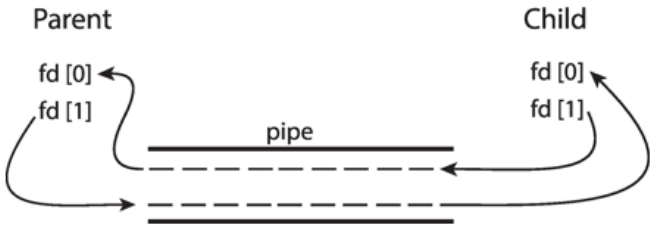


Figure 3.8: Ordinary pipes

Windows calls this pipe **anonymous pipes**.

Ordinary UNIX pipes

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>

#define BUFFER_SIZE 25
#define READ_END  0
#define WRITE_END 1

int main(void){
  char write_msg[BUFFER_SIZE] = "Greetings";
  char read_msg[BUFFER_SIZE];
  pid_t pid;
  int fd[2];

  /* create the pipe */
  if (pipe(fd) == -1) {
    fprintf(stderr,"Pipe failed");
    return 1;
  }

  /* now fork a child process */
  pid = fork();

  if (pid < 0) {
    fprintf(stderr, "Fork failed");
    return 1;
  }

  if (pid > 0) {  /* parent process */
    /* close the unused end of the pipe */
    close(fd[READ_END]);

    /* write to the pipe */
    write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

    /* close the write end of the pipe */
    close(fd[WRITE_END]);
  }
  else { /* child process */
    /* close the unused end of the pipe */
    close(fd[WRITE_END]);

    /* read from the pipe */
    read(fd[READ_END], read_msg, BUFFER_SIZE);
    printf("child read %s\n",read_msg);

    /* close the write end of the pipe */
    close(fd[READ_END]);
  }
  return 0;
}
```

3.9.2 Named pipes

This pipes are more powerful than the ordinary pipes, indeed communication is bidirectional; no parent-child relationship is necessary; several processes can use the named pipe for communication.

This type of pipes are provided on both UNIX and Windows systems.

Named UNIX Pipe read

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#define BUFFSIZE 512
#define err(mess) { fprintf(stderr,"Error: %s.", mess); exit(1); }

void main(){
    int fd, n;
    char buf[BUFFSIZE];

    if ( (fd = open("fifo_x", O_RDONLY)) < 0)
        err("open");
    while( (n = read(fd, buf, BUFFSIZE) ) > 0) {
        if ( write(STDOUT_FILENO, buf, n) != n) {
            exit(1);
        }
    }
    close(fd);
}
```

Named UNIX Pipe write

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#define BUFFSIZE 512
#define err(mess) { fprintf(stderr,"Error: %s.", mess); exit(1); }

void main(){
    int fd, n;
    char buf[BUFFSIZE];

    mkfifo("fifo_x", 0666);
    if ( (fd = open("fifo_x", O_WRONLY)) < 0)
        err("open");
    while( (n = read(STDIN_FILENO, buf, BUFFSIZE) ) > 0) {
        if ( write(fd, buf, n) != n) {
            err("write");
        }
    }
    close(fd);
}
```

3.10 Communication in Client-Server Systems

So far we have talked about the intern communication, now we want to communicate with other processes. To talk about it we introduce the **Soket** system.

3.10.1 Sockets

A socket is defined as an endpoint for communication, concatenate IP address and port to the receiver: 192.168.1.1:80.

All ports below 1024 are **well known** used for standard servicies like mail 25, secure web browsing 443, FTP 20 21 etc.

3.10.2 Remote Procedure Calls

The Remote Procedure Calls RPC abstracs procedures calls between processes on networked systems. Again it use ports for service differentiation.

Data are represented by **External Data Representation, XLD**, thus it can be read by different architecture: Big-endian and Little-endian.

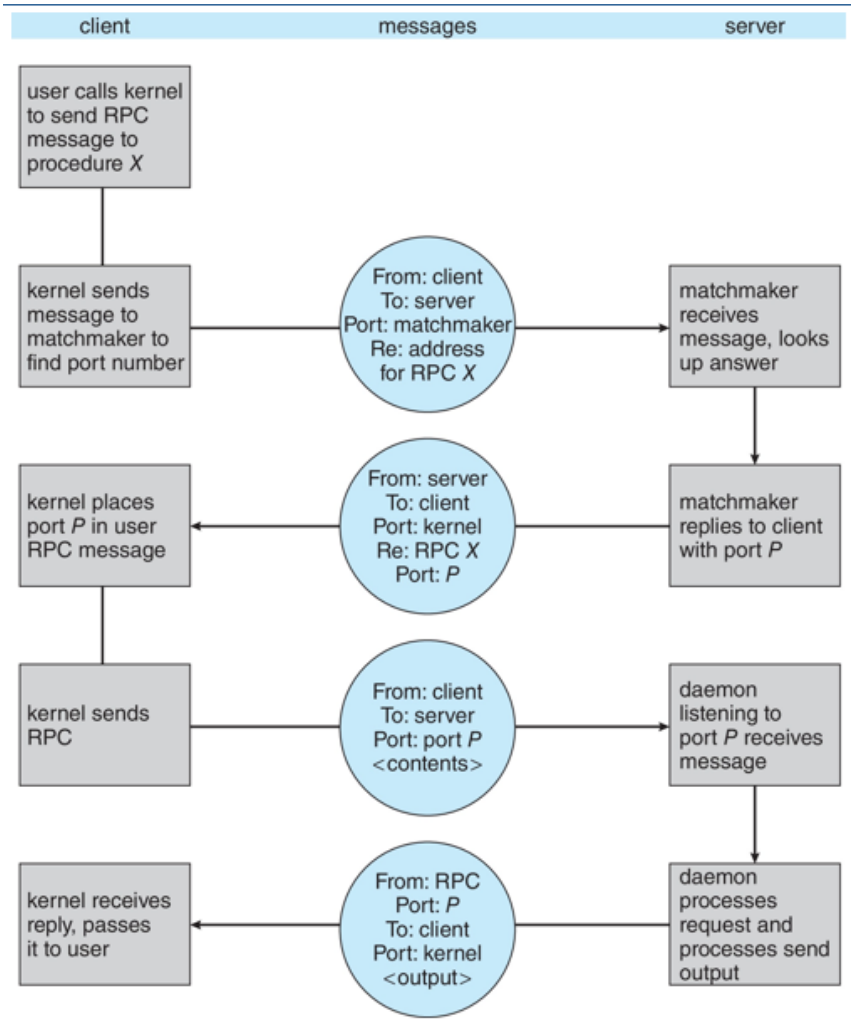


Figure 3.9: Execution of RPC

Chapter 4

Threads & Concurrency

4.1 Process Concept

Task, process and subprocess are all synonym. We talked about how to fork a process, parent con create a child a copy of the parent. They are two different process with one signle thread each.

We also sad that when we switch from process to another the context switch must copy all the data (registers, PC, code, opened files, etc.) and this require time and more small it is more efficient are the OS.

So is not efficient to duplicate all the parent instead of a small part of process.

We would like to reduce this overhead to improve overall performance and the solution is to use **thread**.

4.1.1 Motivation

Most, may be all, modern application are multithreaded, thread run within application. Multiple tasks, in a single application, can be implemented by separated threads:

- Update display;
- Fetch data from file/network;
- Spell checking;
- Send data into the net;
- etc.

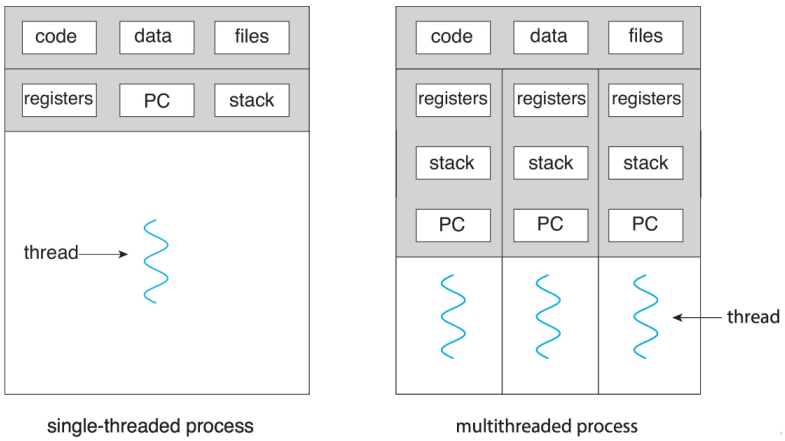
Also they can simplify code and increase efficiency also kernel are generally multithreaded.

Another big difference is: process creation is **heavy-weight**, like I sad before, while thread creation is **light-weight**.

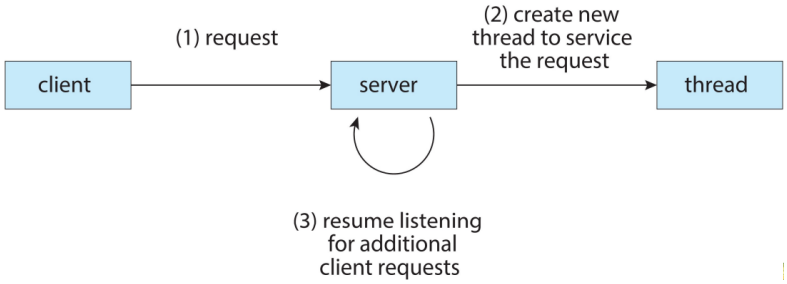
4.2 Multi-Threading

Single and Multithreaded processes

Thread is a basic unit of CPU usage composed of **thrID**, **PC**, **register set**, **stack** and some shared data: **heap data**, **code** (text) and **OS resouces** with other threads belonging to the same process.



Example:



4.2.1 Fork == Thread?

At this point, you may ask yourselves if the UNIX fork spawns a **new process** (child of the caller) or a **Thread**? The answer is: it spawns a child process.

But then, what is the difference between a child process and a thread?

4.2.2 Linux Threads

For Linux systems refers to them as **tasks** rather than **threads**.

Task (process) creation is done through *fork()* but also Task (thread) creation is done through *clone()* system call. This function, depends on the parameters passed, can allows a child task to share the address spaces of the parent task (process).

Flags control behavior, can be used for both:

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

Figure 4.1: Flags to use in clone

By default:

- fork copies all data structures of the parent process to the child,
- while clone points to them

Fork child processes:

Are processes, so its creation is heavier than threads, the purpose is to create a new process, which becomes the child process of the caller. Both processes will execute the next instruction following the *fork()* system call and if the child wants to communicate with the parent, it has to set-up brand new shared memories or message-passing because if the parents change the value of a variable the child does not see it (because is a copy).

Two identical copies of the computer's address space,code, and stack are created one for parent and child. Like "Dolly" sheep.

Threads:

Purpose is to create a new thread in the program which is given the same process of the caller and threads within the same process can communicate using shared memory.

The second thread will share data, open files, signal handlers and signal dispositions, current working directory, user and group ID's. The new thread will get its own stack, thread ID, and registers though.

Continuing the analogy: if it was a sheep, your process grows a fifth head or a second tail when it creates a new thread; those are connected and share the same brain.

In my OS

But then, is my OS scheduling processes or threads?

- The Linux scheduler (on recent Linux kernels, e.g. 3.0 at least) is scheduling schedulable tasks or "ready" tasks.
- A task/job may be:

  a single-threaded process (e.g. the main of a C code, or something created by fork without any thread library);

  any thread inside a multi-threaded process (including its main thread), in particular Posix threads;

  kernel tasks, which are started internally in the kernel and stay in kernel land (e.g. kworker, nfsiod, kjournald ...). These are managed directly by the OS.

In other words, threads inside multi-threaded processes are scheduled like non-threaded -i.e. single threaded- processes.

4.3 Benefits

Using threads improve the overall OS this is some resons:

- **Responsiveness** – may allow continued execution if part of process is blocked, especially important for user interfaces;
- **Resource Sharing** – threads share resources of process, easier than shared memory or message passing;
- **Economy** – cheaper than process creation, thread switching lower overhead than context switching: **Don't need to resent** the cache/CPU is I am switching from a process thread to another thread of the same process;
- **Scalability** – process can take advantage of multicore architectures: This is true also for single-threaded processes if handled correctly.

4.4 Multicore Programming

Multicore or multiprocessor systems put pressure on programmers, challenges include:

Dividing activities

Balance load

Data splitting

Data dependency

Testing and debugging

Concurrency: supports more than one task making progress. Single processor / core, scheduler providing concurrency. Not really concurrent, there is still at most a thread executing at once. However, it

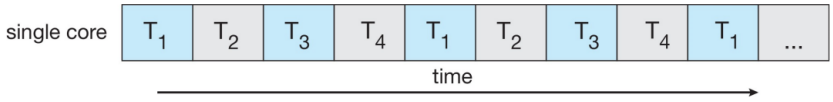


Figure 4.2: Concurrent execution on single-core system

gives the impression that many things are running in parallel.

If there are lots of process the time between the first and the last are more relevant.

Parallelism: implies a system can perform more than one task simultaneously.

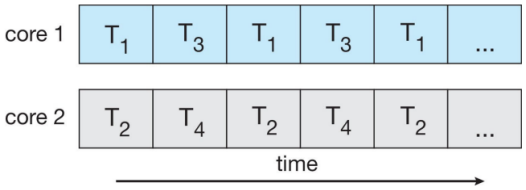


Figure 4.3: Parallel execution on a multi-core system

The are two type of parallelism:

- – **Data parallelism:** distributes subsets of the same data across multiple cores, same operation on each, none data is shared with other tasks;

  e.g. quicksort, margesort.

- – **Task parallelism:** distributing threads across cores, each thread performing **unique operation**, you have some data that it can be shared in different tasks.

  e.g. pipelining

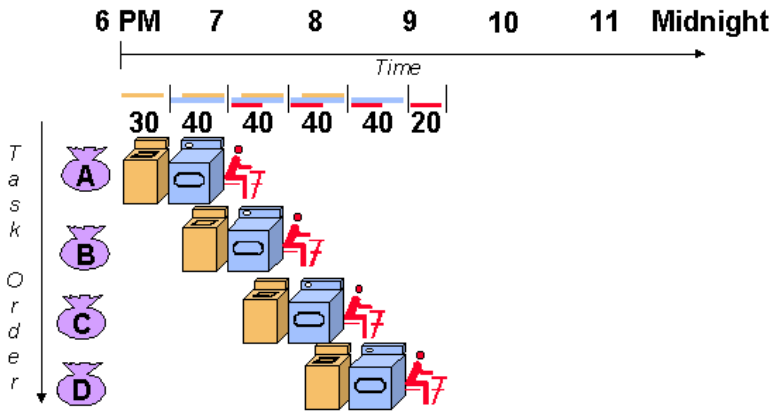


Figure 4.4: Pipelining

Pipelining, a standard feature in RISC processors, is much like an assembly line. Because the processor works on different steps of the instruction at the same time, more instructions can be executed in a shorter period of time.

RISC Pipelines: A RISC processor pipeline operates in much the same way, although the stages in the pipeline are different. While different processors have different numbers of steps, they are basically variations of these five, used in the MIPS R3000 processor:

1. fetch instructions from memory
2. read registers and decode the instruction
3. execute the instruction or calculate an address
4. access an operand in data memory
5. write the result into a register

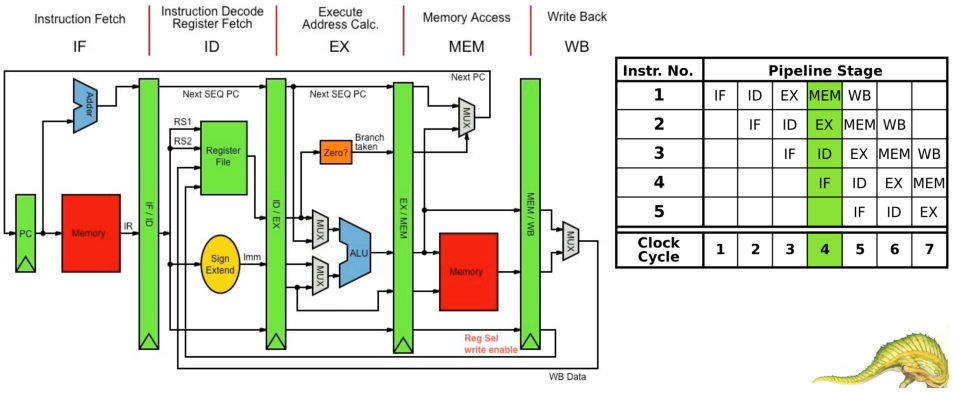


Figure 4.5: ALU

4.5 Amdahl's Law

Identifies performance gains from adding additional cores (**N**) to an application that has both serial (**S**) and parallel (**P**) components.

$$speedup \leq \frac{1}{S + \frac{1-S}{N}}$$

That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times, so not the x2 improvement.

As N approaches infinity, speedup approaches 1/S and if S approaches 1 the speedup approaches 1 (no speedup).

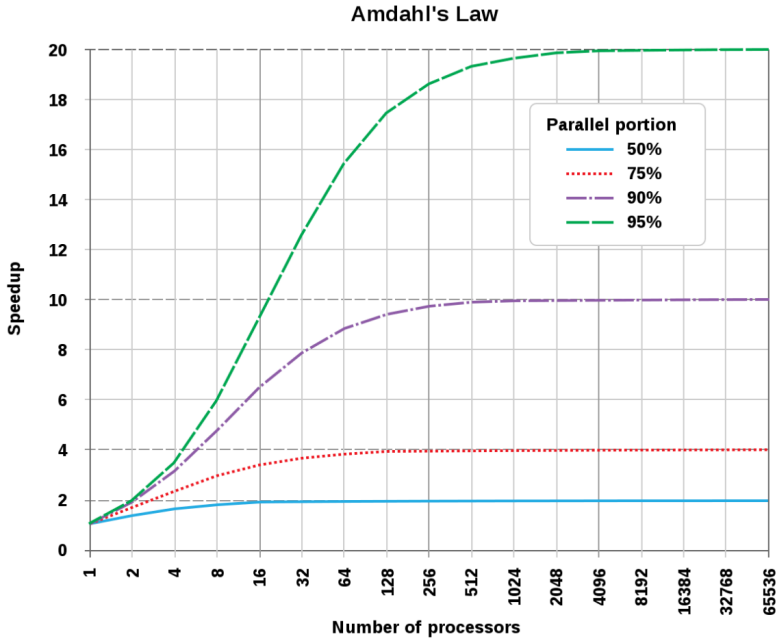


Figure 4.6: Amdahl's Law

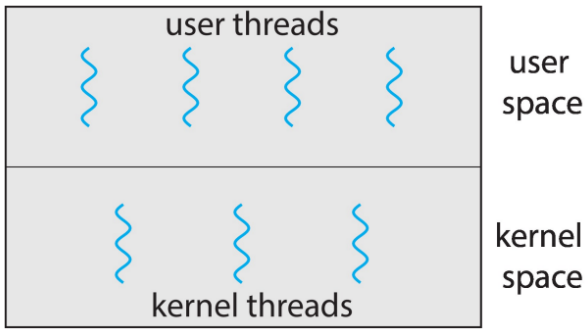
4.6 Multi-thread models

It could be dangerous to create a lot of threads, think about a virus that create lots of threads doing nothing only steel time CPU of others processes.

There are two different threads:

User thread – management done by user-level threads library (pthread, Windows thread, Java runnables and threads)

Kernel threads – Supported by the Kernel



The OS maps user and kernel thread in three way:

- Many-to-One
- One-to-One
- Many-to-Many

4.6.1 Many-to-One

Many user-level threads mapped to single kernel thread, multiple threads may not run in parallel on multicore system because only one may be in kernel at a time.

If one thread blocking causes all to block. This approach is use in a few OS like Solaris Green Threads or GNU Portable Threads.

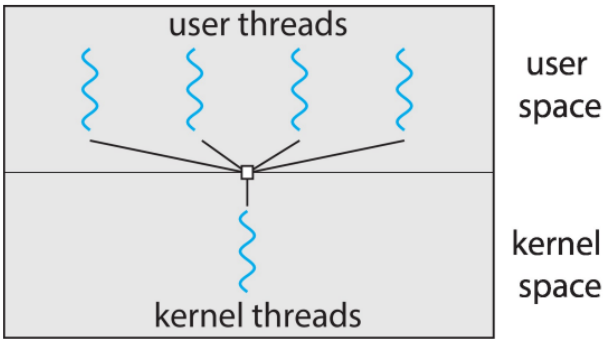


Figure 4.7: Many-to-One thread

4.6.2 One-to-One

The easiest solution, each user-level thread maps to kernel thread this allows to more concurrency than many-to-one.

For the security the number of threads per process sometimes restricted due to overhead.

Some example are: **Windows** and **Linux** systems.

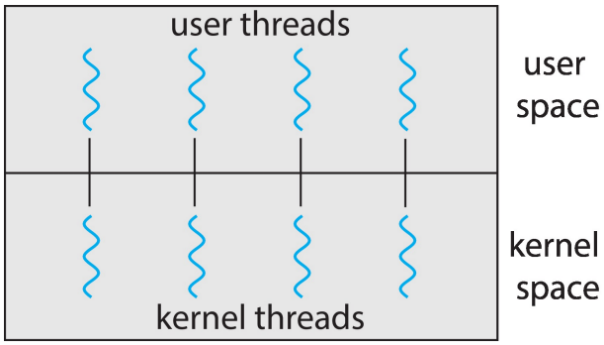


Figure 4.8: One-to-One thread

4.6.3 Many-to-Many

The last solution is a mix of the approach talked before.

Allows many user level threads to be mapped to many kernel threads if I have N user threads, I will be mapped into M ($\leq$ N) kernel threads this M size can be set by the OS, also considering the amount of available cores, or the amount of processes allows the operating system to create a sufficient number of kernel threads without over-creating them

Very complex to implement and manage.

Some example are: Windows with the ThreadFiber package, otherwise not very common.

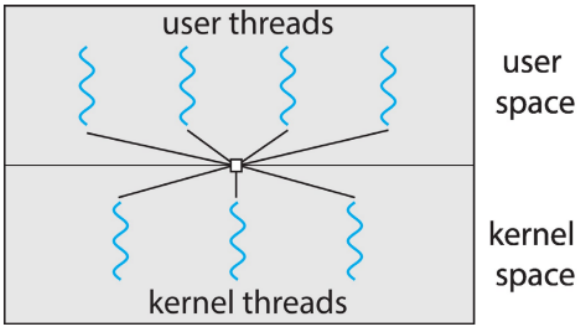


Figure 4.9: Many-to-Many thread

4.6.4 Two-level Model:

similar to many-to-many, except that it allows a user thread to be bound to kernel thread. Not really used in practice.

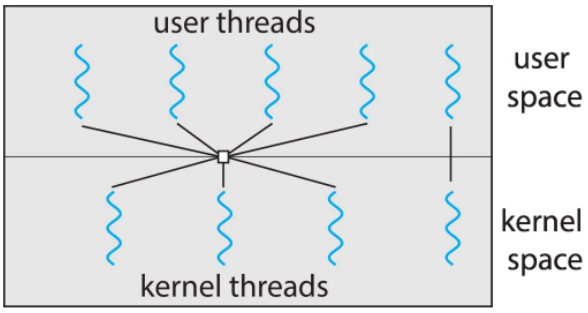


Figure 4.10: Two-level Model

4.7 Thread libraries

A Thread library provides programmer with API for creating and managing threads. Two primary ways of implementing: Library entirely in user space; Kernel-level library supported by the OS.

Pthreads

A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization, is a Specification, not implementation. Infact API specifies behavior of the thread library, implementation is up to development of the library, this is common in UNIX OS (Linux and Mac OS X).

There are around 100 threads procedures, all prefixed pthread_ and they can be categorized into four groups:

- Thread management – creating, joining threads etc.
- Mutexes
- Condition variables
- Synchronization between threads using read write locks and barriers
- Spinlocks

Pthread_attr_init

Initializes thread attributes, needed for creating new threads, by default it creates a structure like:

Detach state = PTHREAD_CREATE_JOINABLE

Scope = PTHREAD_SCOPE_SYSTEM

Inherit scheduler = PTHREAD_INHERIT_SCHED

Scheduling policy = SCHED_OTHER

Scheduling priority = 0

Guard size = 4096 bytes

Stack address = 0x40196000

Stack size = 0x201000 bytes

Pthread_create

Creates a new thread, initializing its tid, the function required some inputs:

- The empty tid value (need to pass the address – by reference);
- The thread attributes (pass by reference, again);
- A pointer to the function to be exercised by the thread;
- Params of the function, as specified in the signature.

```
1 int pthread_create( pthread_t *restrict thread,
2                     const pthread_attr_t *restrict attr,
3                     void *(*start_routine)(void *),
4                     void *restrict arg );
```

In Linux when you call the create() function it means, create and run the thread.

Pthread_join

The pthread_join() function waits for the thread specified by thread to terminate. If that thread has already terminated, then pthread_join() returns immediately. The thread specified by thread must be joinable.

Means that if a process calls the pthread_join and the thread never ends (stuck somewhere, ...) the process will wait endlessly, it is called a **blocking function call**.

```
int pthread_join(pthread_t thread, void **retval);
```

Pthread_exit

The pthread_exit() function terminates the calling thread and returns a value via retval that (if the thread is joinable) is available to another thread in the same process that calls pthread_join().

To access the return value you have to provide a non-NULL pointer as a second parameter of the pthread_join function, this has to be typed as a void**.

```
[[noreturn]] void pthread_exit(void *retval);
```

Example:

```

#include <pthread.h>
#include <stdio.h>

#include <stdlib.h>

int sum; /* this data is shared by the thread(s) */
void *runner (void *param); /* threads call this function */

int main(int argc, char *argv[]) {

    pthread_t tid; /* set the thread identifier */
    pthread_attr_t attr; /* set of thread attributes */

    /* set the default attributes of the thread */
    pthread_attr_init(&attr);
    /* create the thread */
    pthread_create(&tid, &attr, runner, argv[1]);
    /* wait for the thread to exit */
    pthread_join(tid, NULL);

    printf("sum = %d\n", sum);
}


/* The thread will execute in this function */
void *runner (void *param){

    int i, upper = atoi(param);
    sum = 0;

    for (i=1; i <= upper; i++)
        sum += i;

    pthread_exit(0);
}
```

4.7.1 JAVA THREADS

When you write code in Java therads are managed by the JVM, typically implemented using threads model provided by underlying OS.

NOTE: the JVM is a middlewere between your code and the host OS, thus uses OS-specific funztions.

Java thread may be created by:

- Extending Thread class;
- Implementing the Runnable interface;

The standard practise to implement thread is to implement the **Runnable interface**:

```
class Task implements Runnable{

    ...

    public void run(){
        System.out.println("I am a thread.");
    }

    ...
}
```

creation and waiting on a thread:

```
import java.lang.Thread;

class Main {

    public static void main() {

        Thread worker = new Thread(new Task());
        worker.start();

        try{
            worker.join();
        }catch(InterruptedException ie){
            System.out.println("An error occurred.");
        }

    }
}
```

4.7.2 OpenMP

In C, the process of creating a thread costs a lot of time and effort to the programmer. For this reason openMP libraries automatically implement the creation and the waiting.

Is a set of compiler directives and an API for C, C++, FORTRAN; it provides support for parallel programming in shared-memory environments and identifies **parallel regions** – blocks of code that can run in parallel. All of this only writing a single line: `#pragma omp parallel`, it create as many threads as there are cores.

```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[]) {

    /* sequential code */

    #pragma omp parallel
    {                              //you must go to the new line
        printf("I am a parallel region.");
    }


    /* sequential code */

    return 0;
}
```

There are different ways to make parallel code with omp, the code above defines a parallel region, which is code that will be executed by multiple threads in parallel, to enforce a given number of threads you should disable dynamic teams and specify the desired number of threads with either `omp_set_num_threads()`:

```
#include <omp.h>

omp_set_dynamic(0);     // Explicitly disable dynamic teams
omp_set_num_threads(4); // Use 4 threads for all consecutive parallel
    regions

#pragma omp parallel ...
{
  // Code here will be run by 4 threads in parallel
}
```

Or with the `num_threads` OpenMP clause:

```
omp_set_dynamic(0);     // Explicitly disable dynamic teams
// Spawn 4 threads for this parallel region only

#pragma omp parallel... num_threads (4)
{
  // Code here will be run by 4 threads in parallel
}
```

Directives

There are different ways to make parallel code with omp.

- **Pragma omp parallel**

  Defines a parallel region, which is code that will be executed by multiple threads in parallel.

- **Pragma omp for**

  Causes the work done in a for loop inside a parallel region to be divided among threads.

- **Pragma omp sections**

  Identifies code sections to be divided among all threads.

  Each section has to be tagged as pragma omp section.

- **Pragma omp single**

  Lets you specify that a section of code should be executed on a single thread, not necessarily the main thread.

If I only have one line of code within a parallel block, I just mark the single line with the last directives above.

```
1
2  int main(){
3
4      ...
5
6      #pragma omp parallel
7      {
8
9          //parallel
10
11         #pragma omp single
12         {
13             // sequential
14         }
15
16         //parallel
17
18     }
19
20     return 0;
21 }
```

4.8 Implicit threading

Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads. Creation and management of threads done by compilers and run-time libraries rather than programmers (e.g. OpenMP).

This is called implicit threading, because the creation and all the waiting mechanisms of the thread are managed in the background.

Some methods to use implicit thread:

- Thread Pools
- Fork-Join
- OpenMP
- Grand Central Dispatch
- Intel Threading Building Blocks

4.8.1 Thread Pools

This library emulate OpenMP in Java language, it crate a number of threads in a pool where they await work.

Some advantages:

- Usually slightly faster to service a request with an existing thread than create a new thread
- Allows the number of threads in the application(s) to be bound to the size of the pool
- Separating task to be performed from mechanics of creating task allows different strategies for running task

  E.g. Tasks could be scheduled to run periodically

Three factory methods for creating thread pools in Executors class:

```
static ExecutorService newSingleThreadExecutor();
static ExecutorService newFixedThreadPool(int size);
static ExecutorService newCachedThreadPool();
```

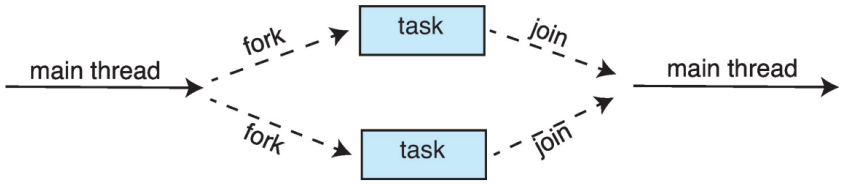
```
import java.util.concurrent.*;

public class ThreadPoolExample {

    public static void main(String[] args) {

        int numTasks = Integer.parseInt(args[0].trim());

        /* Create the thread pool */
        ExecutorService pool = Executors.newCachedThreadPool();

        /* Run each task using a thread in the pool */
        for (int i = 0; i < numTasks; i++)
            pool.execute(new Task());

        /* Shut down the pool once all threads have completed */
        pool.shutdown();
    }
}
```

But also Windows API supports thread pools:

```
DWORD WINAPI PoolFunction(AVOID Param){
    /*
    * This function runs as a separated thread.
    */
}
```

4.8.2 Fork-Join Parallelism

Multiple threads (tasks) are forked, and then joined, this allows to emulate parallelism (not threads) because it is a sub-process. Creation of sub-process is a heavy operation instead to create threads, also each fork have a copy of memory and not shared memory.




```
Task(problem)

if problem is small enough
    then
        solve the problem directly
else
    then
        subtask1 = fork(new Task(subset of problem))
        subtask2 = fork(new Task(subset of problem))
        result1 = join(subtask1)
        result2 = join(subtask2)
endif

return combined results
```

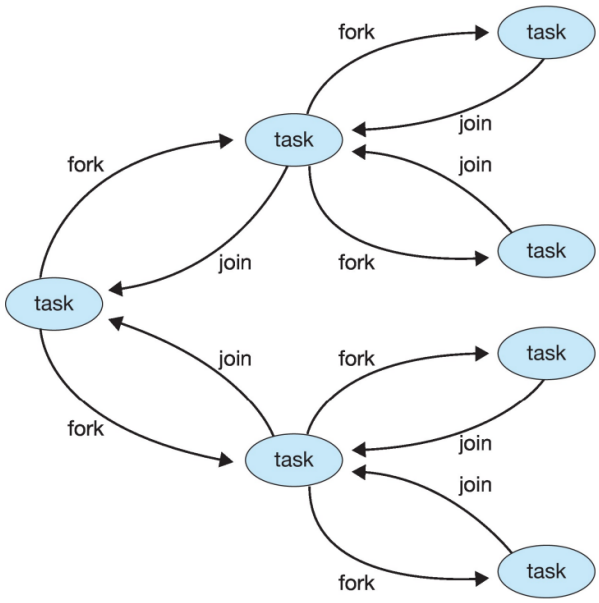


Figure 4.11: Fork-Join Parallelism state machine

4.9 CPU scheduling

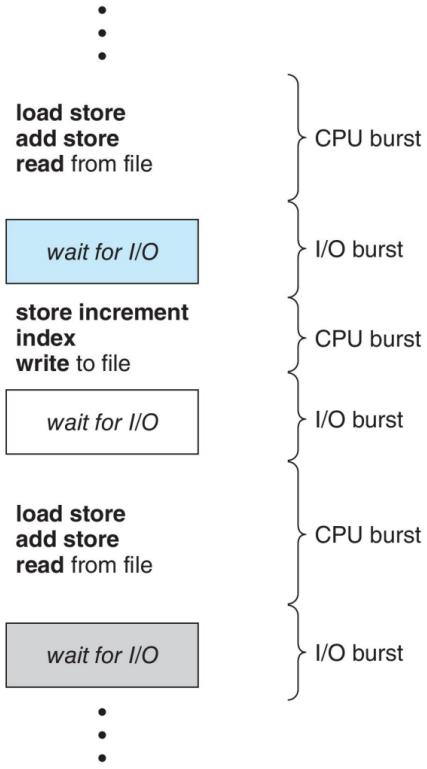
So far we have talked about processes and threads: what they are, how to implement them, how to increase maximum speed, how to use libraries to implement them, etc.

Now we want to know the behavior of all this, how the operating system interacts and handles these activities.

4.9.1 Basic Concepts

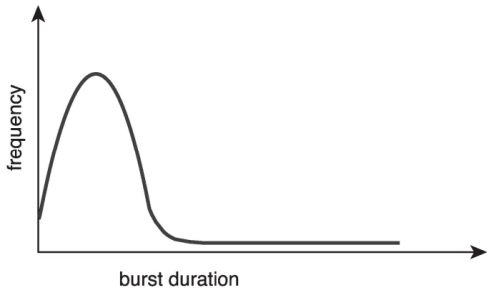
We want to maximize CPU utilization obtained with multiprogramming. The tasks are only CPU burst and I/O burst: Process execution consists of a cycle of CPU execution and I/O wait, in the same order.

The CPU burst distribution is of main concern.



So, if a programme that performs sorting uses the CPU, but when it waits for some I/O the tasks does not use CPU and we want to pull it out from the CPU because others process want use it.

Histogram of CPU-burst Times Large number of short bursts and small number of longer bursts.



4.9.2 CPU Scheduler

The CPU Scheduler working to dispatching tasks to all the cores of the CPU.

The CPU scheduler selects from among the processes in ready queue, and allocates a CPU core to one of them: Queue may be ordered in various ways.

So far, in fact when a task enters the CPU we do not have a method to kick it out and take another one from the ready queue an put it into the cpu, we waiting until it ends.

For this reasons CPU scheduling decisions may take place when a process:

1. Switches from running to waiting state
2. Switches from running to ready state
3. Switches from waiting to ready
4. Terminates

For situations 1 and 4, there is no choice in terms of managing the ready queue. Simply, a new process is taken from the queue.

For situations 2 and 3, however, there is a choice that is up to the scheduler.

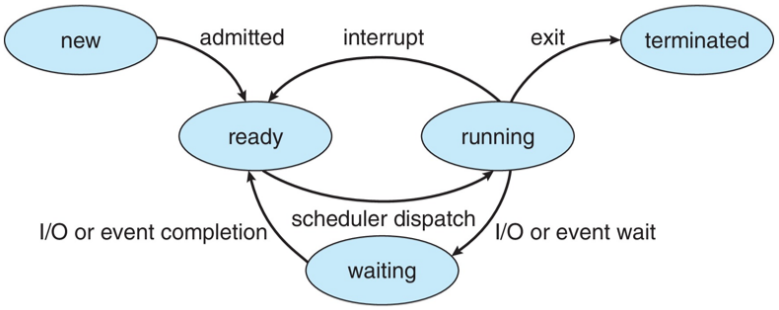


Figure 4.12: Diagram of Process State

4.9.3 Preemptive and Nonpreemptive Scheduling

Under **Nonpreemptive scheduling**, once the CPU has been allocated to a process, the process keeps the CPU until it releases it either by **terminating** or by switching to the **waiting state**.

Otherwise, it is **preemptive**.

Thus, a process could theoretically keep it forever, or, before the OS recognizes this anomalous behavior and kills it.

Virtually all modern operating systems including Windows, MacOS, Linux, and UNIX use preemptive scheduling algorithms. This methods allows to set a priority to each tasks.

Preemptive Scheduling and Race Conditions

Preemptive scheduling can result in race conditions when data are shared among several processes.

Consider the case of two processes that share data. While one process is updating the data, it is preempted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state (data consistency).

```
1 void myFunction(){
2     int myID = thread;
3     myID++;
4 }
```

If it is nonpreemptive the function stuck into the cpu untill the function `myFunction()` terminate (line: 5), otherwise if it is preemptive the CPU scheduler may stopped the execution at line 3 another process could execute the function and the two process have the same myID.

Note: these problems happen only when one process writes and at least another one tries to read it. Read-only variables or atomic block will never have any kind of problems and can be shared without issues.

4.9.4 Dispatcher

Assume that the CPU decided that a given task P1 has to be assigned to a core now running P0. Then what?

Dispatcher module gives control of the CPU to the process selected by the CPU scheduler; this involves:

- Switching context;
- Switching to user mode;
- Jumping to the proper location in the user program to restart that program;

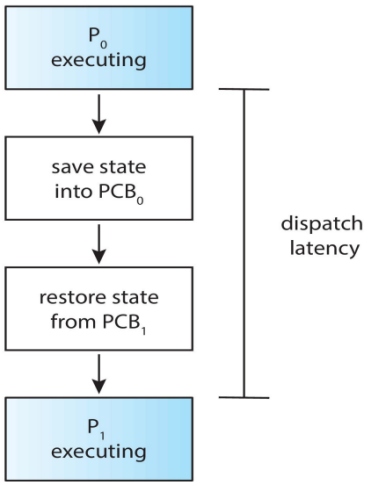


Figure 4.13: Dispatcher

Dispatch latency – time it takes for the dispatcher to stop one process and start another running. The preemptive don't give nothing for free.

4.9.5 Scheduling Criteria

There are some parameters that the CPU scheduler could optimize, but not all of them because some of some of them conflict.

- **CPU utilization** - keep the CPU as busy as possible;
- **Throughput** – # of processes that complete their execution per time unit;
- **Turnaround time** – amount of time to execute a particular process;
- **Waiting time** – the amount of time a process has been waiting in the ready queue;
- **Response time** – amount of time it takes from when a process enters the ready queue to when it gets into the CPU the first time.

The first two are connected.

4.10 Scheduling Algorithms

4.10.1 FCFS

Process	Burst Time
P_1	24
P_2	3
P_3	3

Figure 4.14: First- Come, First-Served

The First-Come, First-Served Scheduling is non preemptive: once assigned, a process finishes is execution.

Waiting time for P1, P2, P3: 0, 24, 27; **Average waiting time**: $(0 + 24 + 27)/3 = 17$.

But if the processes arrive in the order: P2,P3,P1, we obtain other number:
Waiting time for P1, P2, P3: 6, 0, 3; **Average waiting time**: $(6 + 0 + 3)/3 = 3$.
Much better than previous case. This case is called **convoy effect** - short process behind long process.

4.10.2 SJF

Shortest-Job-First Scheduling or, better, shortest-next-CPU-burst. This method minimize the waiting time.

Associate with each process the length of its next CPU burst, but there is a big problem: How do we determine the length of the next CPU burst?

- Could ask the user: unfeasible, introduces delays;
- Estimate but how?

Determining Length of Next CPU Burst

Can be done by using the length of previous CPU bursts (of the same task), using exponential averaging:

$$\tau_{n+1} = \alpha t_n + (1 + \alpha)\tau_n \tag{4.1}$$

where:

- t_n: actual length of the n^{th} CPU burst;
- τ_{n+1}: predicted value for the next CPU burst
- α set to 0.5, $0 \leq \alpha \leq 1$

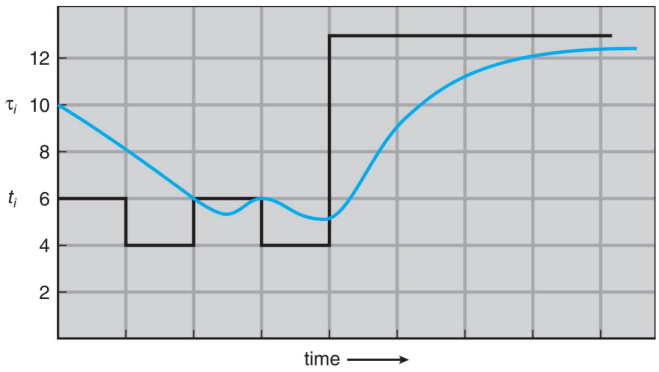


Figure 4.15: Prediction vs Actual CPU Burst

This method is precise if the CPU burst does not change a lot.

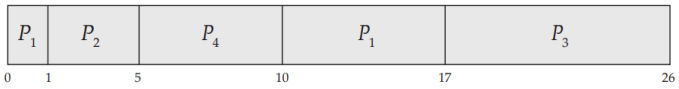
4.10.3 SRT

The Shortest-Remaining-Time-first is the preemptive version of SJN. Whenever a new process arrives in the ready queue, the decision on which process to schedule next is redone using the SJN algorithm.

Is SRT more "optimal" than SJN in terms of the minimum average waiting time for a given set of processes?

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive* SJF Gantt Chart
  - Average waiting time = [(9-0)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5



- Non-preemptive waiting time: [(0)+(8-1)+(17-2)+(12-3)]/4 = 31/4 = 7.8

4.10.4 Round Robin

Each process gets a small unit of CPU time (**time quantum** or **time slice** q), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.

If there are n processes in the ready queue and the time quantum is q, then each process gets 1/n of the CPU time in chunks of at most q time units at once. No process waits more than **(n-1)q** time units.

Timer interrupts every quantum to schedule next process.

Performance:

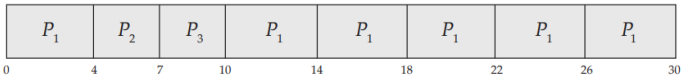
q large: FIFO (FCFS);

q small: RR.

NOTE: the q must be larger with respect to context switch, otherwise overhead is to high, there will be no execution, just context switches after context switches (useless).

RR vs FCFS

Process	Burst Time
P_1	24
P_2	3
P_3	3

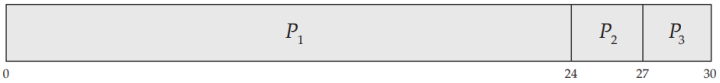


Typically, higher average turnaround than SJF, but better **response**:

Waiting Average = (6 + 4 + 7) / 3 = 5.7

Turnaround = (30 + 3 + 3) / 3 = 12

Response = (4 + 7 + 10) / 3 = 7



Waiting Average = (0 + 24 + 27) / 3 = 17

Turnaround = (24 + 3 + 3) / 3 = 10

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

Turnaround Time Varies With The Time Quantum

Rule of Thumb: 80% of CPU bursts should be shorter than q.

Example: processes P1, P2, P3, P4 with respectively time 6, 3, 1, 7; a good q could be 6.

4.11 Priority Scheduling

A priority number (integer) is associated with each process. The CPU is allocated to the process with the highest priority, smaller integer ≡ highest priority.

- Preemptive: the scheduler kick the task out the CPU for some reasons;
- Non-preemptive: the scheduler must not kick the task out the CPU unless it waiting for an I/O.

But not all the tasks are the same, in fact there are users threads and kernel threads. This latter should not be blocked by the users threads.

SJV is a priority scheduling where priority is the inverse of the predicted next CPU bust. Regardless to all the algorithm chose the tasks that have the higher priority, it means it have the lower CPU burst time.

But there is a problem:

> **Starvation** - low priority tasks may never execute

The solution of this problem is called:

> **Aging** - if tasks does not execute in n-time, its priority (age) is increased, this process is repeated until the execution arrives. The more a process remains in the ready queue, the more his priority (age) increase.

e.g. if the tasks is in the ready queue for 4 CPU burst, I increase its priority.

Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

The scheduler in this situation does not take care about the CPU burst time, it look at the priority of the process.

This priority scheduling has an average waiting time of 8.2. But the burst of CPU of course is important for improving average waiting.

But if the processes are the same priority? There is some mechanism that allow to chose one or the other tasks, we can chose different type of scheduling algorithms:

- Priority scheduling with Round-Robin - same priority, the tasks to execute first is chose by the random algorithm;
- Priority scheduling with FCFS - same priority, the tasks to execute first is chose by the first come first served algorithm;
- Priority scheduling with SJF - same priority, the tasks to execute first is chose by shortest jobs first algorithm;
- Other.

Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

Figure 4.16: Priority Scheduling w/ Round-Robin

4.12 Scheduling in using multiple queues

Having different priority queues for all the priority level in the OS, allow to implement different scheduling algorithms for each queues to decide witch tasks going to be execute first.

4.12.1 Multilevel queue

Multilevel queue scheduler defined by the following parameters:

- Number of queues;
- Scheduling algorithms for each queue;
- Method used to determine which queue a process will enter;
- when that process needs service;
- Scheduling among the queues.

With priority scheduling, have separate queues for each priority.

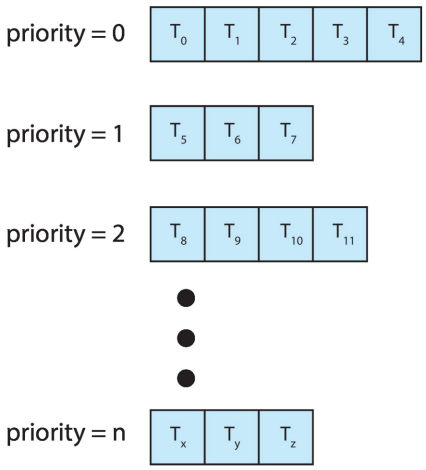


Figure 4.17: Multilevel queue

Aging can be implemented using multilevel feedback queue.

Example of multilevel feedback queue: we look at the CPU bust and not to the priority.

Three queues:

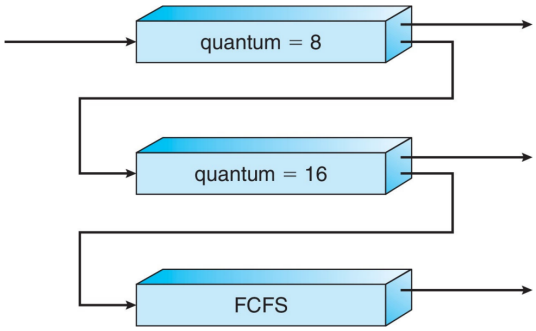
- Q0 – RR with time quantum 8 milliseconds
- Q1 – RR time quantum 16 milliseconds
- Q2 – FCFS

Scheduling: A new process enters queue Q0 which is served in RR

– When it gains CPU, the process receives 8 milliseconds

– If it does not finish in 8 milliseconds, the process is moved to queue Q1

At Q1 job is again served in RR and receives 16 additional milliseconds, if it still does not complete, it is preempted and moved to queue Q2.



4.12.2 Scheduling in multiprocessor systems

CPU scheduling more complex when multiple CPUs are available, in fact multiprocess may be any one of the following architectures:

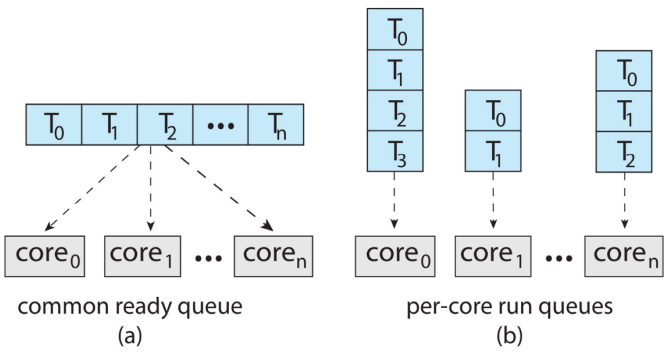
- Multicore CPUs
- Multithreaded cores - the OS think to have double core
- Heterogeneous multiprocessing

When you buy new processor on the box is written: this processor have L1, L2 and L3 cache. It means that the L1 cache is the private cache of the single core, this is divide in 2, L2 is the cache shared by the core, and the L3 cache is the cache shared by all core.

Symmetric multiprocessing

Symmetric multiprocessing (SMP) is where each processor is self scheduling.

- All threads may be in a common ready queue (a) - more easier to implement, all tasks occupy the uncommitted core
- Each processor may have its own private queue of threads (b) - faster in context-switching, virus attack only one core



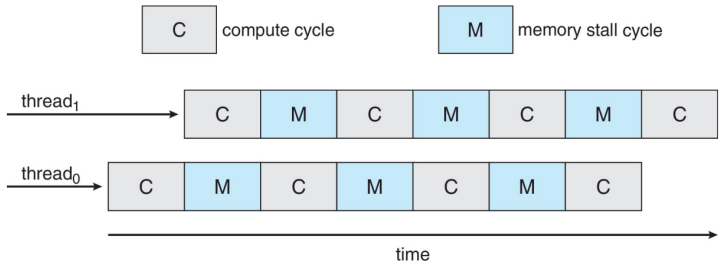
4.12.3 Multi-core Processors

Recent trend to place multiple cores on same physical chip, this approach is faster and consumes less power.

Multiple threads per core (virtual cores) also growing:

- Takes advantage of memory stall to make progress on another thread while memory retrieve happens
- Each core has > 1 hardware threads
- If one thread has a memory stall, switch to another thread!

for those reason the L1 is divide in 2 or more area, witch each contains the data to another thread.



Chip-multithreading (CMT) assigns each core multiple hardware threads. (Intel refers to this as hyperthreading.)

E.g. On a quad-core system with 2 hardware threads per core, the operating system sees 8 logical processors or virtual core.

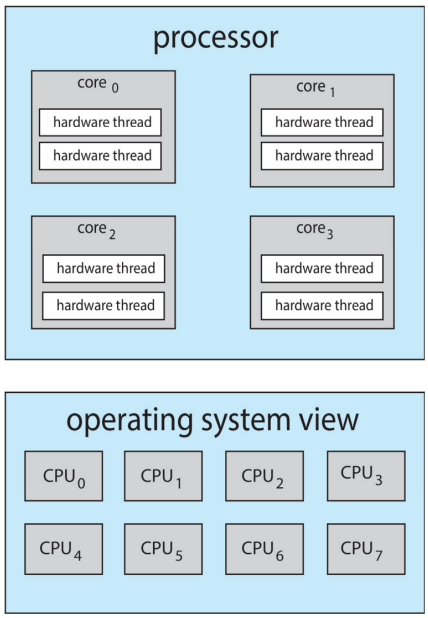
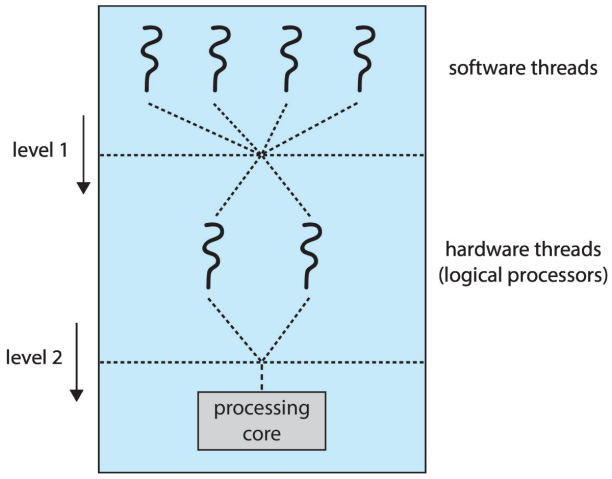


Figure 4.18: Chip-multithreading

Also there are two level of scheduling:

1. The operating system deciding which software thread to run on a logical CPU
2. Each core decides which hardware thread to run on the physical core.



Processor Affinity

When a thread has been running on one processor, the cache contains some information and data and if the process return into the CPU it take the previous data from cache, this benefit in term of latency. We refer to this as a thread having affinity for a processor (i.e. "processor affinity").

Load balancing may affect processor affinity as a thread may be moved from one processor to another to balance loads, yet that thread loses the contents of what it had in the cache of the processor it was moved off of.

- **Soft affinity** – the operating system attempts to keep a thread running on the same processor, but no guarantees
- **Hard affinity** – allows a process to specify a set of processors it may run on.

4.13 Real time CPU scheduling

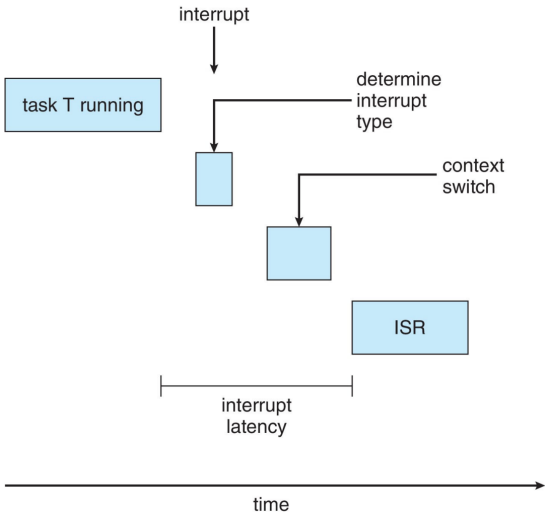
For some reason, the tasks required to be execute in a specific time, or in real-time, e.g. ABS, train tracking and so on. They can present obvious challenges, there are two way to implement this functionality:

- **Soft real-time systems** – Critical real-time tasks have the highest priority, but no guarantee as to when tasks will be scheduled;
- **Hard real-time systems** – task must be serviced by its deadline

4.13.1 Latencies

Event latency – the amount of time that elapses from when an event occurs to when it is serviced. There are two tipes of latencies affect performance:

Interrupt latency – time from arrival of interrupt to start of routine that services interrupt **Dispatch latency** – time for schedule to take current process off CPU and switch to another



4.13.2 Priority-based Scheduling

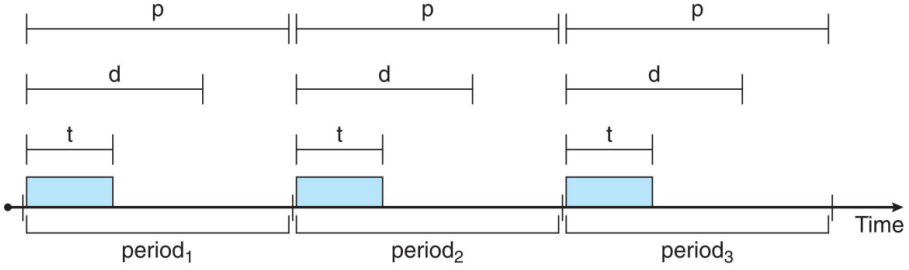
For real-time scheduling, scheduler must support preemptive, prioritybased scheduling. But only guarantees **soft real-time**.

For hard real-time must also provide ability to meet **deadlines**.

Processes have new characteristics: **periodic** ones require CPU at constant intervals; this process has:

- processing time t;
- deadline d;
- period p.

Rate of periodic task is 1/p



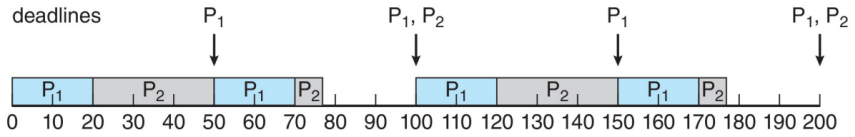
Rate Monotonic Scheduling (RMS)

The priority is assigned based on the inverse of its period, priority = 1/period.

Shorter periods = higher priority;

Longer periods = lower priority.

Example: P1 is assigned a higher priority than P2, P1 has deadline 50, P2 has deadline 100



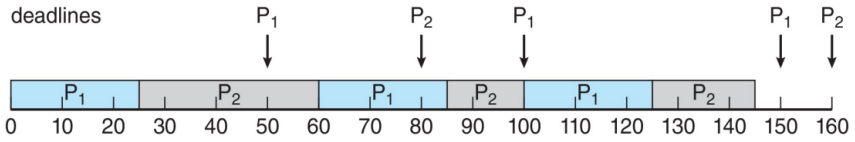
RMS in not optimal solution, P1 is far from its deadline, but P2 is less far from its deadline.

Earliest Deadline First Scheduling (EDF)

In this case the priorities are assigned according to deadlines:

The earlier the deadline, the higher the priority

The later the deadline, the lower the priority



All of this type of scheduling algorithms must have the minimum requirements of core numbers.

4.14 OS scheduling examples

4.14.1 Linux Scheduling in Version 2.6.23+

The Linux scheduling is based on Completely Fair Scheduler (CFS). It is composed of:

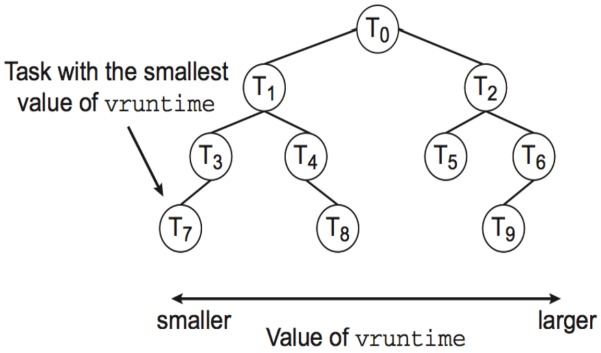
Scheduling classes: each tasks has specific priority, scheduler picks highest priority task in highest scheduling class, rather than quantum based on fixed time allotments, based on proportion of CPU time.
Two scheduling classes included, others can be added:

1. default;
2. real-time.

Virtual run time: CFS scheduler maintains per task virtual run time in variable `vruntime`. Associated with decay factor based on priority of task – lower priority is higher decay rate, normal default priority yields virtual run time = actual run time.

To decide next task to run, scheduler picks task with lowest virtual run time. But how to implement all of this?

To decide witch tasks to run the CFS looks at the `vruntime`, the smartest way to implement this is to see the ready queue like a btree. In this way you have the smallest value always in the same position.



4.14.2 Windows Scheduling

Windows uses priority-based preemptive scheduling. Highest-priority thread runs next. **Dispatcher** is scheduler.

Windows has a 32-level priority scheme, where **Variable class** is 1-15, **real-time class** is 16-31. Priority 0 is memory-management thread.

It is a multilevel queue for each level and if no run-able thread, it runs **idle thread** a task that do noting.

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Figure 4.19: Windows priorities

4.14.3 Solaris

The Solaris OS is used for business. This OS is Priority-based scheduling and has six classes of priority available:

- Time sharing (default) (TS)
- Interactive (IA)
- Real time (RT)
- System (SYS)
- Fair Share (FSS)
- Fixed priority (FP)

Given thread can be in one class at a time and each class has its own scheduling algorithm. Time sharing is multi-level feedback queue, loadable table configurable by sysadmin.

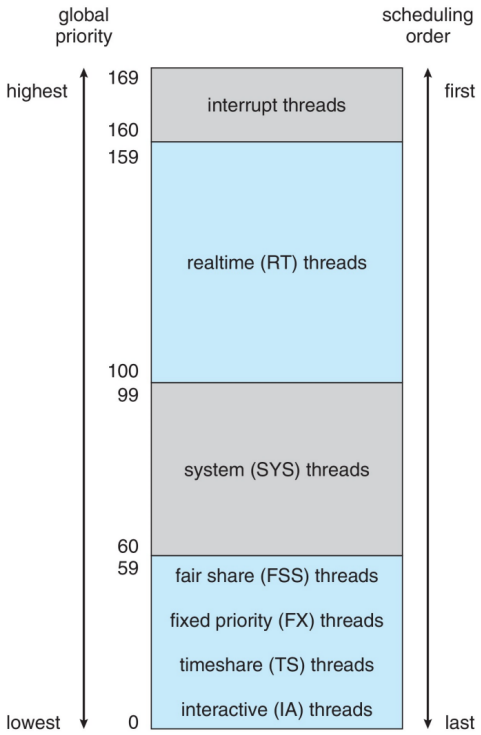


Figure 4.20: Solaris scheduling

4.15 Scheduling evaluation

The scheduling algorithm depends on who is the final users: reading mail, surfing on internet, coding, gaming, render a video, etc. all this tasks required different CPU burst and we would implement a **Deterministic model** to know the CPU burst.

The analytic evaluation is used to takes a particular predetermined workload and defines the performance of each algorithm for that workload.

Consider 5 processes arriving at time 0:

Processes	Burst Time
P1	10
P2	29
P3	3
P4	7
P5	12

The Deterministic Evaluation for each algorithm, calculate minimum average waiting time, simple and fast, but requires exact numbers for input, applies only to those inputs:

- FCS is $28ms$
- Non-preemptive SFJ is $13ms$
- RR is $23ms$

This deterministic model is not applicable in the real-word.

4.15.1 Queueing models

Queueing models are mathematical tools used to analyze systems where customers (or items) wait for service. This model describes the arrival of processes, and CPU and I/O bursts probabilistically. These models help predict things like average queue lengths, waiting times, server utilization, and the probability of having a certain number of customers in the system.

Computer system described as network of servers, each with queue of waiting processes

Little's Formula

Little's law – in steady state, processes leaving queue must equal processes arriving, thus:

$$n = \lambda W$$

Where:

n = average queue length

W = average waiting time in queue

λ = average arrival rate into queue

For example, if on average 7 processes arrive per second, and normally 14 processes in queue, then average wait time per process = 2 seconds; but if I have a CPU that can execute only 6 tasks per second the queue grow.

Problem: very rough estimate.

4.15.2 Simulations

Queueing models limited, **Simulations** more accurate. Is a virtual model, more the model is close to the real behavior and more the simulation is real and the statistics accurate.

Other option is just implement new scheduler and test in the real system, this involves:

- High cost, high risk
- Environments vary

The simulation is one of the possible low cost solution to try new scheduler.

Chapter 5

Synchronization Tools

5.1 Critical section and race condition

For now we know: to create tasks, the tasks have a own life, scheduling algorithms. But how we manage it?

Problem: in a dual core processor each process has own life, if in the same time two process call the fork() call, the sub-processes have the same PID! This may happen because in the dual-core processor the tasks are execute in parallel.

This problem also happen even if we have a single-core CPU, because if the OS has the preemption mechanism in each time the function fork() can be kicked out of the CPU before increase the PID number, the another tasks call the same function and after that we have the same problem saw before: two tasks with same PID.

This is called **race condition**, and implies the inconsistency of the data read. We want to maintaining this consistency of data.

NOTA: the race condition and the inconsistency of the data happen **only** if the variable is in the read/write mode, this not happen if the variable is only in read mode!

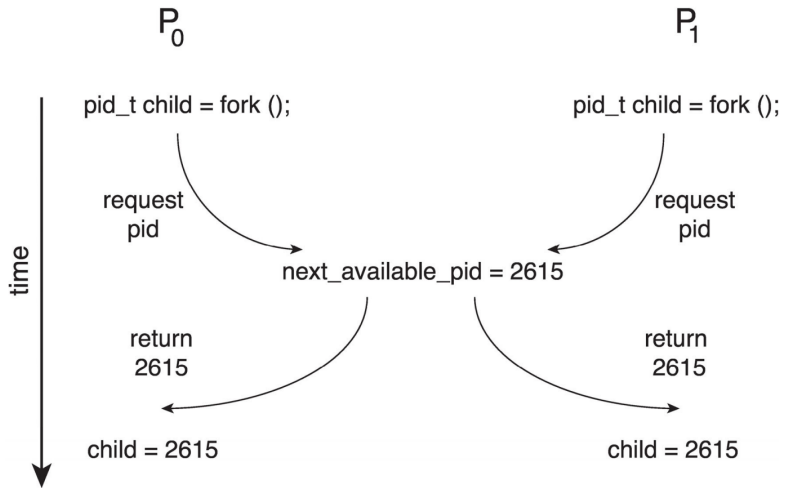


Figure 5.1: Race condition on kernel variable next_available_pid

5.1.1 Critical section

Consider system of n processes, each process has critical section segment of code: process may be changin common variables, updating tables, etc.

When one process in critical section, no other may be in its critical section, **Critical section problem** is to design protocol to solve this.

Each process must:

- ask permission to enter critical section in entry section,
- may follow critical section with exit section,
- then remainder section

```
while( true ){
    ...

    //entry into critical section
        critical section
    //exit critical section

    ...
}
```

5.1.2 Critical section problem

Requirements for solution to critical-section problem:

- **Mutual Exclusion** - If process Pi is executing in its critical section, then no other processes can be executing in their critical sections; no more one in the same critical section;
- **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the process that will enter the critical section next cannot be postponed indefinitely; if no one tasks are into the critical section other process may access into it;
- **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted; if a tasks want to enter into the critical section must waiting to have the access.

If your system has a **single core** CPU and is **nonpreemtive** OS, you never have this problem, but the question is: who in 2024 does not have a multi-core CPU inside a PC? None.

5.2 Software solutions 1

Assume that the **load** and **store** machine-language instructions are atomic; that is, cannot be interrupted.

Two processes share one variable called: int turn, this variable indicates who can access the critical section. Initially, the value of turn is set to i.

```
while (true){
    while (turn = = i);
    /* critical section */
    turn = i;
    /* remainder section */
}
```

```
while (true){
    while (turn = = j);
    /* critical section */
    turn = j;
    /* remainder section */
}
```

Correctness of this Software Solution:

- **Mutual exclusion** is preserved, since Pi enters critical section only if turn == i;
- **Progress requirements** is not respected. The other task can not access into its critical section;
- **Bounded-waiting**, also, is not respected. Process J may be waiting for the critical section endlessly.

5.2.1 Peterson's Solution

Same as before: two process, load and store machine-language instructions are atomic.

The change is: two processes share two variables: int turn and boolean flag[2]. The first indicates which processes enters the critical section, and the flag array is used to indicate if a process is ready to enter the critical section, flag[i] = true -> the process Pi is ready.

```
while (true){
    flag[i] = true;
    turn = j;
    while (flag[j] && turn = = j);

    /* critical section */

    flag[i] = false;

    /* remainder section */
}
```

Correctness of Peterson's Solution:

- **Mutual exclusion** is preserved, since Pi enters critical section only if turn == i, either flag[j] = false or turn = i;
- **Progress requirements** is satisfied.
- **Bounded-waiting**, requirement is met. If it is the turn of the other, but the other is not willing to enter the critical section, the first process will enter (even if not its turn technically).

Although useful for demonstrating an algorithm, Peterson's Solution is not guaranteed to work on modern architectures. To improve performance, processors and/or compilers may reorder operations that have no dependencies.

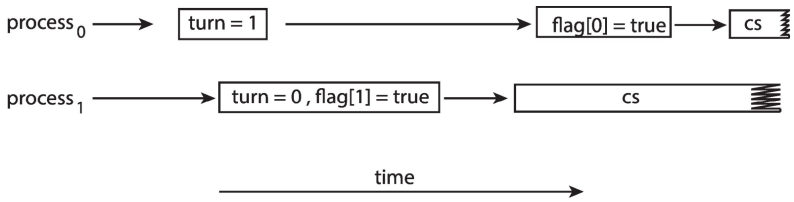
For single-threaded this is ok as the result will always be the same. For multithreaded the reordering may produce inconsistent or unexpected results! Ideally, you want a single instruction for entry/exit section.

5.2.2 Reordering

The compiler transform the high programme language into binary code, but for optimization it happen that some operation are physically distant, and the preemption could cause some error.

5.2.3 Peterson vs Multi-Threading

The effects of instruction reordering in Peterson's Solution



This allows both processes to be in their critical section at the same time, to ensure that Peterson's solution will work correctly on modern computer architecture we must use a **Memory Barrier**.

5.3 Memory Barrier

Memory models may be either:

- **Strongly ordered** – where a memory modification of one processor is immediately visible to all other processors.
- **Weakly ordered** – where a memory modification of one processor may not be immediately visible to all other processors

A memory barrier is an instruction that forces any change in memory to be propagated (made visible) to all other processors.

I need some barrier line that tells to compiler that it can reorder instruction before and after the barrier. In this way the consisntency of the data are ok.

Example: boolean flag = false; int x = 0;

```
//thread one
while (!flag)

memory_barrier();
print x;
```

```
//thread two
while (!flag)

memory_barrier();
print x;
```

For Thread 1 we are guaranteed that that the value of flag is loaded before the value of x.
For Thread 2 we ensure that the assignment to x occurs before the assignment flag.

If we do not write `memory_barrier();` the programm print the value of 0, before the value x is set bt thread 2.

5.4 Hardware instructions

How to implement all of this in the language programm?

Many systems provide hardware support for implementing the critical section code. We will look at three forms of hardware support:

- Hardware instructions
- Atomic variables

5.4.1 Hardware Instructions

Special hardware instructions that allow us to either test-and-modify the content of a word, or to swap the contents of two words atomically:

Test-and-Set instruction

Compare-and-Swap instruction

test_and_set instruction

```
boolean test_and_set (boolean *target){
    boolean rv = *target;
    *target = true;
    return rv:
}
```

Properties:

- Executed atomically
- Returns the original value of passed parameter
- Set the new value of passed parameter to true
- No changes if target is already true

Shared boolean variable lock, initialized to false:

```
do {
    while (test_and_set(&lock)); /* do nothing */
    /* critical section */
    lock = false;
    /* remainder section */
} while (true);
```

After the first process, any process can go in. There is no queue maintained, so any new process that finds the lock to be false again can enter. So bounded waiting is not ensured.

The `lock` is:

- **false** when the critical section is empty (not busy);
- **true** when the critical section is full (busy);

The `Test_and_set` function guarantee the mutual exclusion and the progress, but does not guarantee bounded waiting!

compare_and_swap instruction

This hardware instruction first reed the lock value, then set a specific value.

```
int compare_and_swap(int *value, int expected, int new_value) {
    int temp = *value;
    if (*value == expected)
    *value = new_value;
    return temp;
}
```

Properties:

- Executed atomically
- Returns the original value of passed parameter value
- Set the variable value the value of the passed parameter new_value but only if *value == expected is true.
- That is, the swap takes place only under this condition

Solution using `Test_and_set` and shared integer lock initialized to 0.

```
while (true){
    while (compare_and_swap(\&lock, 0, 1) != 0); /* do nothing */
    /* critical section */
    lock = 0;
    /* remainder section */
}
```

But this solution can resolve the previous problem?

Mutual exclusion: yes

Progress: yes

Bounded Waiting: no

To solve this we can rewrite the code as following:

```
while (true) {
    waiting[i] = true;
    key = 1;
    while (waiting[i] && key == 1)
        key = compare_and_swap(&lock,0,1);
    waiting[i] = false;

    /* critical section */
    j = (i + 1) \% n;
    while ((j != i) && !waiting[j])
        j = (j + 1) \% n;

    if (j == i)
        lock = 0;
    else
        waiting[j] = false;
    /* remainder section */
}
```

Here is where the bounded waiting is guaranteed Scans from i+1 to n to 0 to i-1 for a process waiting in the CS.

In this way we are create a looping queue, so if a process exit to the critical section, it passes the control to the next process and not to the tasks that got the lock before anyone. This guarantees that any process, who willing into the critical section, sooner or later will enter into it.

In this way the **bounded waiting** is guarantee.

5.5 Atomic Variables

Typically, instructions such as compare-and-swap are used as building blocks for other synchronization tools. Using atomic variables provides **atomic** updates on a basic data type such as integer and booleans.

Example: Let **sequence** be an atomic variable, Let **increment()** be operation on the atomic variable sequence. The command `increment(&sequence);` ensures sequence is incremented without interruption:

```
void increment(atomic_int *v){
    int temp;
    do {
        temp = *v;
    }while (temp != (compare_and_swap(v,temp,temp+1));
}
```

5.6 Mutex Locks

Previous solutions are complicated and generally inaccessible to application programmers, OS designers build software tools to solve critical section problem, the simplest is **mutex lock** a boolean variable indicating if lock is available or not. It protect the critical section by:

- First **acquire()** a lock;
- Then **release()** the lock.

Usually implemented via hardware atomic instructions such as compare-and-swap. But this solution requires **busy waiting**, this lock therefore called a **spinlock**.

```
while (true) {
    acquire lock
        critical section
    release lock

    remainder section
}
```

If not free, you will be blocked on "acquire" until it releases. This is the "busy wait", you cant do anything else while you wait, so you are subtract useful work to the CPU.

5.7 Semaphore

I will not subtract useful work to the CPU, so we want someone who notify to the task when a specific resource, in this case the critical section, i free to entry.

Semaphore is a **synchronization tool**, and not only a way to enter into a critical section, that provides more sophisticated ways, than mutex locks, for processes to synchronize their activity. With semaphores we can solve various synchronization problems.

Semaphores can only be accessed via two atomic operations:

- wait()
- signal()

5.7.1 Definition of the wait operation

```
wait(S) {
    while (S <= 0); // busy wait
    S--;
}
```

5.7.2 Definition of the signal operation

```
signal(S) {
    S++;
}
```

The are two different type of semaphores:

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1, this is equivalent to mutex lock

It is possible to implement a counting semaphores as a binary semaphores.

Semaphores usage example: Solution to CS problem creating a semaphores "mutex" initialized to 1

```
wait(mutex);
    CS
signal(mutex);
```

Consider P1 and P2 that with two statements S1 and S2 and the requirement that S1 to happen before S2. Create a semaphore “synch” initialized to 0.

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

All works good but there are still busy waiting, **spinlock**! We can do better!

5.7.3 Semaphore Implementation with no Busy waiting

With each semaphore there is an associated waiting queue, a list of a processes willing to enter into the critical section.

Each entry in a waiting queue has two data items:

- Value (of type integer);
- Pointer to next record in the list, single(ahead)-pointer linked list.

Also there are two operations:

- **block** – place the process invoking the operation on the appropriate waiting queue
- **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}
```

```
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

The struct of semaphores with waiting queue is:

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

5.7.4 Problems with semaphores

If a programmer do not take care when use signal and wait, the semaphores work in a bad way! Some example could be:

Omitting the wait: you forget that you are using the critical section;

Omitting the signal: you exit the critical section without notify that.

5.8 Monitors

Another high-level abstraction tools used for synchronization is called **monitors**.

Pseudocode syntax of a monitor:

```
monitor monitor-name{
    // shared variable declarations
    procedure P1 (...) { ... }
    procedure P2 (...) { ... }
    procedure Pn (...) { ... }
    initialization code (...) { ... }
}
```

You can see it as a service that you can call:

- It offers some functions;
- You don't worry about critical sections, parallelism, whatsoever;
- The monitor itself is managing them (mutual exclusion, progress);
- It will answer eventually (bounded waiting).

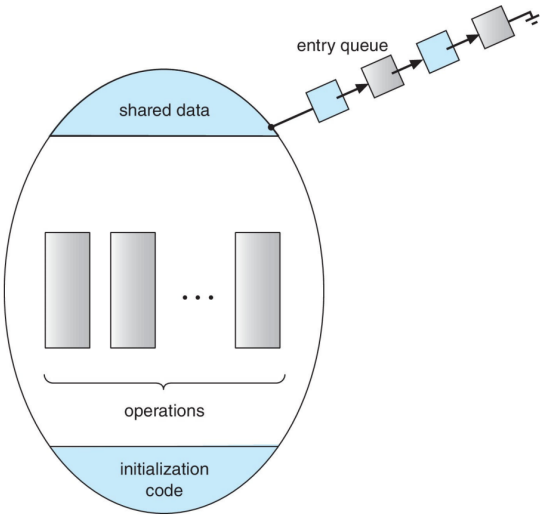


Figure 5.2: Schematic view of a Monitor

5.8.1 Monitor Implementation Using Semaphores

Variables: semaphore mutex, mutex = 1.

Each procedure P is replaced by :

```
wait(mutex);
    ...
    body of P;
    ...
signal(mutex);
```

Mutual exclusion within a monitor is ensured.
However, what if procedures need to interact in the critical section?
Maybe a procedure can be ran until a given point but then needs some action by another process?

5.8.2 Condition Variables

Monitors allow for the creation of condition variables: **condition x, y;**
Two operations are allowed on a condition variable:

- **x.wait()** – a process that invokes the operation is suspended until x.signal();
- **x.signal()** – resumes one of processes (if any) that invoked x.wait(), if not any then it has no effect on the variable

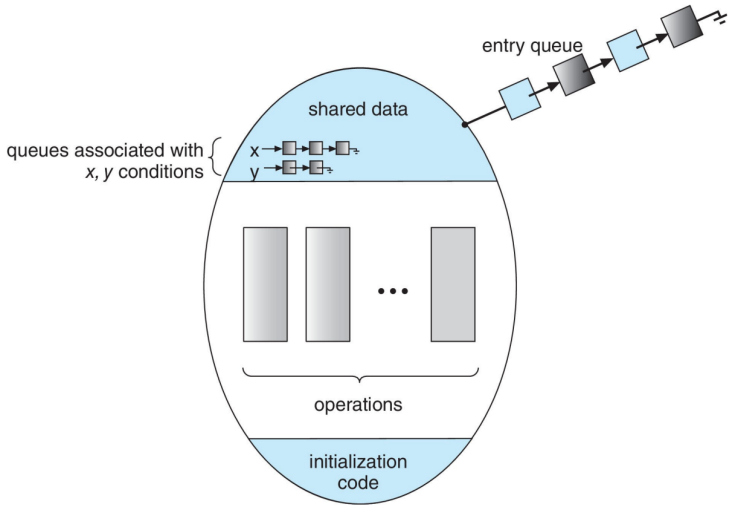


Figure 5.3: Monitor with Condition Variables

Usage of condition variable example:

Consider P1 and P2 that that need to execute two statements S1 and S2 and the requirement that S1 to happen before S2.

```
F1:
    S1;
    done = true;
    x.signal();
F2:
    if done = false
    x.wait()
    S2;
```

Example: variables

semaphore mutex; // (initially = 1)

semaphore next; // (initially = 0)

int next_count = 0; // number of processes waiting inside the monitor

Each function P will be replaced by

```
wait(mutex);
    ...
body of P;
    ...
if (next_count > 0)
    signal(next)
else
    signal(mutex);
```

Mutual exclusion within a monitor is **ensured**.

For each condition variable x, we have:

- semaphore x_sem; // (initially = 0)
- int x_count = 0;

Implementation of x.wait()

```
x_count++;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x_count--;
```

Implementation of x.signal()

```
if (x_count > 0) {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}
```

5.8.3 Resuming Processes within a Monitor

If several processes queued on condition variable x, and x.signal() is executed, which process should be resumed? **FCFS** may not be enough we can decide to pass a parameter inside the wait function.

x.wait(c)

where c is an integer (called the priority number). The process with lowest number (highest priority) is scheduled next. In this way the bounded waiting can be not respected. We can solve this be using ageing.

5.9 Single Resource allocation

Allocate a single resource among competing processes using priority numbers that specifies the maximum time a process plans to use the resource.

The process with the shortest time is allocated the resource first

```
R.acquire(t);
    ...
    access the resurce;
    ...
R.release;
```

Where R is an instance of ResourceAllocator, and t is the maximum time a process plans to use the resource.

```
monitor ResourceAllocator{
    boolean busy;
    condition x;

    void acquire(int time) {
        if (busy)
            x.wait(time);
        busy = true;
    }
    void release() {
        busy = false;
        x.signal();
    }
    initialization code() {
        busy = false;
    }
}
```

5.10 Liveness

Processes may have to wait indefinitely while trying to acquire a synchronization tool such as a mutex lock or semaphore. Waiting indefinitely violates the progress and bounded-waiting criteria discussed at the beginning of this chapter.

Liveness refers to a set of properties that a system must satisfy to ensure processes make progress: **Indefinite waiting** is an example of a liveness failure because the bounded waiting is no satisfied.

5.11 Deadlock

Two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.

Example: let S and Q be two semaphores initialized to 1.

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

Consider if P0 executes wait(S) and P1 wait(Q). When P0 executes wait(Q), it must wait until P1 executes signal(Q)

However, P1 is waiting until P0 execute signal(S).

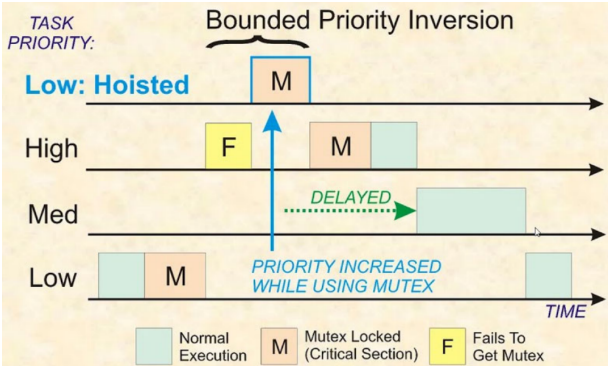
Since these signal() operations will never be executed, P0 and P1 are deadlocked.

5.11.1 Starvation

Indefinite blocking. Process may never be removed from the semaphore queue in which it is suspended

5.11.2 Priority Inversion

Scheduling problem when lower-priority process holds a lock needed by higher-priority process. This is solved via priority-inheritance protocol: when a task blocks one or more higher-priority tasks, it ignores its original priority assignment and executes its critical section at the highest priority level of all the tasks it blocks



Chapter 6

Synchronization Examples

6.1 POSIX Synchronization

POSIX API proviedes:

- mutex locks
- semaphores
- condition variable

Widely used on UNIX, Linux, and macOS.

6.2 POSIX Mutex Locks

Creating and initializing the lock

```c
#include <pthread.h>

pthread_mutex_t mutex;

/* create and initialize the mutex lock */
pthread_mutex_init(&mutex, NULL);
```

Acquiring and releasing the lock

```c
/* acquire the mutex lock */
pthread_mutex lock (&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

6.3 POSIX Semaphores

POSIX provides two versions – **named** and **unnamed**. Named semaphores can be used by unrelated processes, unnamed cannot, similar to named / unnamed pipes.

6.3.1 POSIX Named Semaphores

Creating an initializing the semaphore:

```
#include <semaphore.h>
sem_t *sem;

/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);
```

Another process can access the semaphore by referring to its name SEM

Acquiring and releasing the semaphore:

```
/* acquire the semaphore */
sem_wait(sem);

/* critical section */

/* release the semaphore */
sem_post(sem);
```

6.3.2 POSIX Unnamed Semaphores

Creating an initializing the semaphore:

```
#include <semaphore.h>

sem_t sem;

/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);
```

Acquiring and releasing the semaphore:

```
/* acquire the semaphore */
sem_wait(&sem);

/* critical section */

/* release the semaphore */
sem_post(&sem);
```

6.4 POSIX Condition Variables

Since POSIX is typically used in C/C++ and these languages do not provide a monitor, POSIX condition variables are associated with a POSIX mutex lock to provide mutual exclusion: Creating and initializing the condition variable:

```
#include <pthread.h>

pthread_mutex_t mutex;
pthread_cond_t cond_var;

/* Initialize the mutex lock and condition variable */
pthread_mutex_init(&mutex, NULL);
pthread_cond_init(&cond_var, NULL);
```

Thread waiting for the condition a == b to become true:

```
pthread_mutex_lock(&mutex);
while (a != b)
    pthread_cond_wait(&cond_var, &mutex);

pthread_mutex_unlock(&mutex);
```

Thread signaling another thread waiting on the condition variable:

```
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```

NOTE: pthread_cond_wait unlocks the mutex just before it sleeps, but then it re-acquires the mutex (which may require waiting) when it is signalled, before it wakes up. So if the signalling thread holds the mutex (the usual case), the waiting thread will not proceed until the signalling thread also unlocks the mutex.

This can generate a deadlock!

6.5 Java Synchronization

Java creates an entry queue that contains the threads that want to access the critical section. Java provides rich set of synchronization features:

- Java monitors
- Reentrant locks
- Semaphores
- Condition variables

Every Java object has associated with it a single lock.

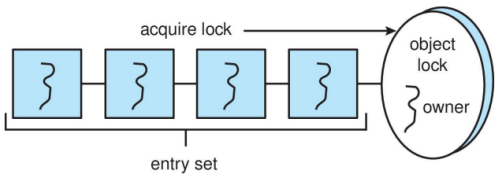
If a method is declared as synchronized, a calling thread must own the lock for the object.

If the lock is owned by another thread, the calling thread must wait for the lock until it is released.

Locks are released when the owning thread exits the synchronized method.

```
public class BoundedBuffer<E> {

  private static final int BUFFER_SIZE = 5;

  private int count, in, out;
  private E[] buffer;

  public BoundedBuffer() {
    count = 0;
    in = 0;
    out = 0;
    buffer = (E[]) new Object[BUFFER_SIZE];
  }

  /* Producers call this method */
  public synchronized void insert(E item) {
    // ...
  }

  /* Consumers call this method */
  public synchronized E remove() {
    // ...
  }
}
```

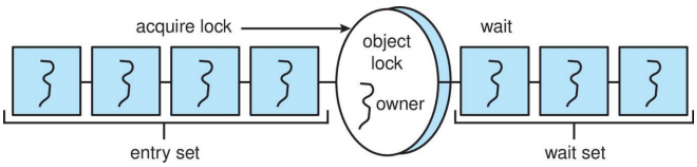
A thread that tries to acquire an unavailable lock is placed in the object's entry set:



Similarly, each object also has a wait set.

When a thread calls wait():

1. It releases the lock for the object
2. The state of the thread is set to blocked
3. The thread is placed in the wait set for the object



A thread typically calls **wait()** when it is waiting for a condition to become true.
But the question is: how does a thread get notified?

When a thread calls **notify()**:

- An arbitrary thread T is selected from the wait set;
- T is moved from the wait set to the entry set;
- Set the state of T from blocked to runnable.

T can now compete for the lock to check if the condition it was waiting for is now true.

6.5.1 Java Semaphores

Constructor:

```
Semaphore(int value);
```

Usage:

```
Semaphore sem = new Semaphore(1);

try{

    sem.acquire();

    /* Cristical section */

}catch (InterruptedException ie) {
    // do something
}finally{
    sem.release();
}
```

6.6 OpenMP Synchronization

OpenMP is a set of compiler directives and API that support parallel progamming.

```
void update(int value){

    #pragma omp critical
    {
        count += value
    }
}
```

NOTE: the curly brackets must be in a new line.

The code after the `#pragma omp critical` contains a critical section and it will be performed atomically.

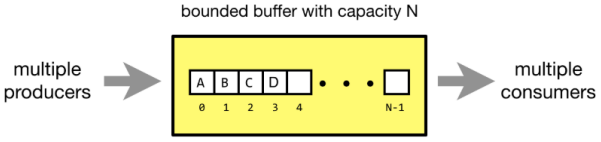
6.7 Bounded buffer

The bounded-buffer problems, a.k.a. the producer-consumer problem, is a classic example of concurrent access to a shared resource. A bounded buffer lets multiple producers and multiple consumers share a single buffer.

Producers write data to the buffer and consumers read data from the buffer.

Producers must block if the buffer is full.

Consumers must block if the buffer is empty



How can we solve the problem with semaphores?

1. Semaphore mutex initialized to the value 1, to avoid concurrent accesses to the buffer;
2. Semaphore elements initialized to the value 0, to know if there is something to read;
3. Semaphore free initialized to the value n to track free spaces.

The structure of the producer process:

```
while (true) {
        ...
        /* produce an item in next_produced */
        ...
    wait(free);
    wait(mutex);
        ...
        /* add next produced to the buffer */
        ...
    signal(mutex);
    signal(elements);
}
```

The structure of the consumer process:

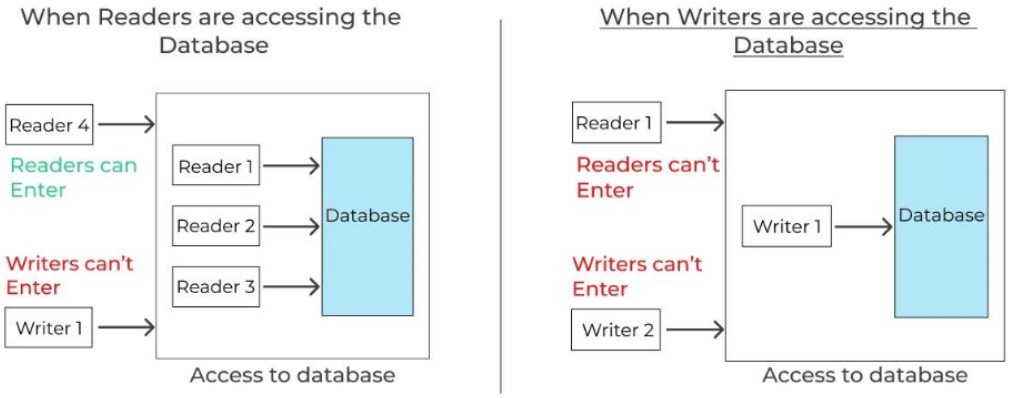
```
while (true) {
     wait(elements);
     wait(mutex);
          ...
          /* remove an item from buffer to next_consumed */
          ...
     signal(mutex);
     signal(free);
          ...
          /* consume the item in next consumed */
          ...
}
```

6.8 Readers-Writers Problem

A data set is shared among a number of concurrent processes:

- Readers – only read the data set; they do not perform any updates
- Writers – can both read and write

Problem – allow multiple readers to read at the same time, only one single writer can access the shared data at the same time.



We want to access into a shared data: Data set, how we can do?

- Semaphore rw_mutex initialized to 1, this is to deal with the fact that there should not be two writers or a reader and a writer at the same time
- Integer read_count initialized to 0, this is to keep track of how many readers are there
- Semaphore mutex initialized to 1, this is to update read_count safely

The structure of a writer process

```
while (true) {
    wait(rw_mutex);
        ...
        /* writing is performed */
        ...
    signal(rw_mutex);
}
```

The structure of a reader process

```
while (true){
    wait(mutex);
    read_count++;
    if (read_count == 1) // first reader needs lock
        wait(rw_mutex);
    signal(mutex);
    ...
    /* reading is performed */
    ...
    wait(mutex);
    read_count--;
    if (read_count == 0) // last reader releases lock
        signal(rw_mutex);
    signal(mutex);
}
```

There are some problems: The solution can result in a situation where a writer process never writes. It is referred to as the "First reader-writer" problem.

The "Second reader-writer" problem is a variation the first reader-writer problem that state: Once a writer is ready to write, no "newly arrived reader" is allowed to read.

Both the first and second may result in **starvation**. Leading to even more variations. The problem is solved on some systems by kernel providing **reader-writer locks**.

6.9 Dining-Philosophers Problem

This is a conceptual problem. N philosophers' sit at a round table with a bowel of rice in the middle

Each philosopher to eat must take the right and the left chopstick. But if everyone take before the left chopstick, none can eat and it produce a starvation.

In the case of 5 philosophers, the shared data

- Bowl of rice (data set)
- Semaphore chopstick [5] initialized to 1

Semaphore Solution, the structure of Philosopher i :

```
while (true){
    wait (chopstick[i] );
    wait (chopStick[ (i + 1) \% 5] );

        /* eat for awhile */

    signal (chopstick[i] );
    signal (chopstick[ (i + 1) \% 5] );

        /* think for awhile */
}
```

There is a problem: if each philosopher takes the i-th chopstick and then waits for the other they can not! **DEADLOCK**.

Easy solution: one philosopher behaves asymmetrically, willing to take the i+1-th chopstick first.

6.9.1 Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers{
    enum {THINKING; HUNGRY, EATING} state [5];
    condition self [5];

    void take_fork (int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING)
        self[i].wait;
    }
    void put_fork (int i) {
        state[i] = THINKING;
        // test left and right neighbors
        test((i + 4) \% 5);
        test((i + 1) \% 5);
    }
    void test (int i) {
        if ((state[(i + 4) \% 5] != EATING) && (state[i] == HUNGRY)
   &&(state[(i + 1) \% 5] != EATING) ) {
            state[i] = EATING ;
            self[i].signal () ;
        }
    }
    initialization_code() {
        for (int i = 0; i < 5; i++)
            state[i] = THINKING;
    }
}
```

Each philosopher "i" invokes the operations take_fork() and put_fork() in the following sequence:

```
DiningPhilosophers.take_fork(i);
    /** EAT **/
DiningPhilosophers.put_fork(i);
```

No deadlock, but starvation is possible.

Chapter 7

Deadlocks

Nowadays the OS has more instance of the same resource for example think about to the CPU, normally in 2024, it contains more than 4 core.

Each tasks require a lot of different resources:

- CPU time;
- network;
- read & write from the SSD;
- etc.

and for each the process must: request, use and release the resource. I would to find a general role to check before if it could be a deadlock.

Example: If we have P1 and P2 and each 2 chopstick, we understood that there will never be a deadlock.

Deadlock can arise if four conditions hold simultaneously:

- **Mutual exclusion**: only one thread at a time can use a resource, sharing variable is risky BUT NOT in all cases, only in R/W case!;
- **Hold and wait**: a thread holding at least one resource is waiting to acquire additional resources held by other threads, chopstick;
- **No preemption**: a resource can be released only voluntarily by the thread holding it, after that thread has completed its task;
- **Circular wait**: there exists a set T0, T1, ..., Tn of waiting threads such that T0 is waiting for a resource that is held by T1, T1 is waiting for a resource that is held by T2, ..., Tn–1 is waiting for a resource that is held by Tn, and Tn is waiting for a resource that is held by T0. ($n > 1$)

7.1 Resource-Allocation Graph

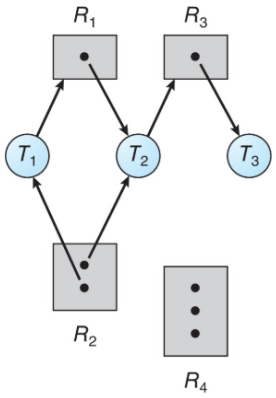


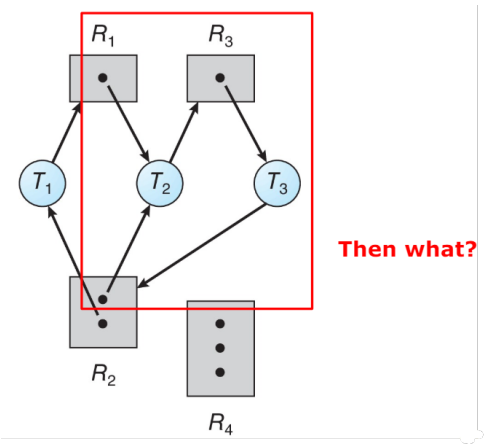
Figure 7.1: Resource Allocation Graph Example

One instance of R1, Two instances of R2, One instance of R3, Three instance of R4.

T1 holds one instance of R2 and is waiting for an instance of R, T2 holds one instance of R1, one instance of R2, and is waiting for an instance of R, T3 is holds one instance of R3.

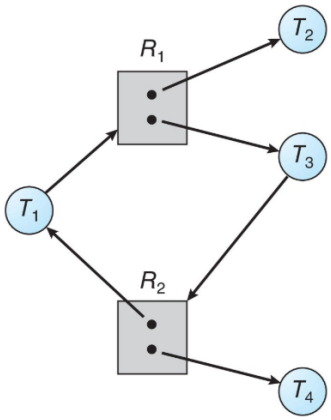
In this situation there are **none deadlock**, no circular waiting.

Example of deadlock



In this case there is a deadlock, because there are two circular wait!

Example of no deadlock



No deadlock here, eventually T4 will release R2, making it available to T3 who is requesting it.

Summary

- If graph contains no cycles → no deadlock
- If graph contains a cycle →

  if only one instance per resource type involved, then deadlock

  if several instances per resource type, possibility of deadlock

7.2 Methods for Handling Deadlocks

We want to handle the deadlock that happened. Is difficult for a system recover a deadlock that happened, thus we ensure that the system will never enter a deadlock state:

- Deadlock prevention
- Deadlock avoidance

7.3 Deadlock Prevention

Invalidate one of the four necessary conditions for deadlock:

- Mutual Exclusion – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- Hold and Wait – must guarantee that whenever a thread requests a resource, it does not hold any other resources
  - Require threads to request and be allocated all its resources before it begins execution or allow thread to request resources only when the thread has none allocated to it.
  - Cons: Low resource utilization; starvation possible
- No Preemption:
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - Preempted resources are added to the list of resources for which the thread is waiting
  - Thread will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
- Circular Wait:
  - Impose a total ordering of all resource types, and require that each thread requests resources in an increasing order of enumeration, Not always flexible, but doable

7.4 Circular Wait

Invalidating the circular wait condition is most common. Simply assign each resource (i.e., mutex locks) a unique number.

Resources must be **acquired in order**. All the process has the same structure like this:

```
wait(M1)
wait(M2)

/* execution */

signal(M2)
signal(M1)
```

7.5 Deadlock Avoidance

Requires that the system has some additional a priori information available.

Simplest and most useful model requires that each thread declare the maximum number of resources of each type that it may need.he deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.

Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes

This is **feasible in real-time systems**, where tasks have to be allocated in advance for schedulability evaluation, and also to check for deadlocks.

7.6 (Thread-)Safe State

When a thread requests an available resource, system must decide if immediate allocation leaves the system in a safe state.

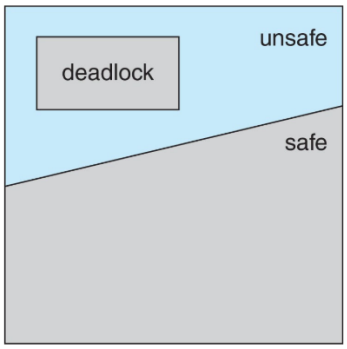
System is in safe state if a sequence of thread wanting to execute a task has ALL the necessary resource.

That is:

- If Ti resource needs are not immediately available, then Ti can wait until all Tj have finished;
- When Tj is finished, Ti can obtain needed resources, execute, return allocated resources, and terminate;
- When Ti terminates, Ti +1 can obtain its needed resources, and so on.

7.7 Recap

- If a system is in safe state → no deadlocks;
- If a system is in unsafe state → possibility of deadlock;
- Avoidance → ensure that a system will never enter an unsafe state, thus, ensure that there will be no deadlocks!



Chapter 8

Main memory

This section provide a smart way to use all the powerful of the hardware you bought, the main focus is on the **main memory**.

We saw at the begin of the course that a program, stored into the disk, became a process when it be brought from disk into main memory.

Main memory and registers are the only storage that the CPU can access directly. Memory unit only sees a stream of:

- addresses + read requests, or
- address + data and write requests

The CPU can access to the registers in one, or less, CPU clock. However, when the CPU takes data from the main memory it use many clock cycles, causing **stall**.

Cache sits between the main memory and the registers to provide a buffer of the needed data, we don't want a bottleneck.

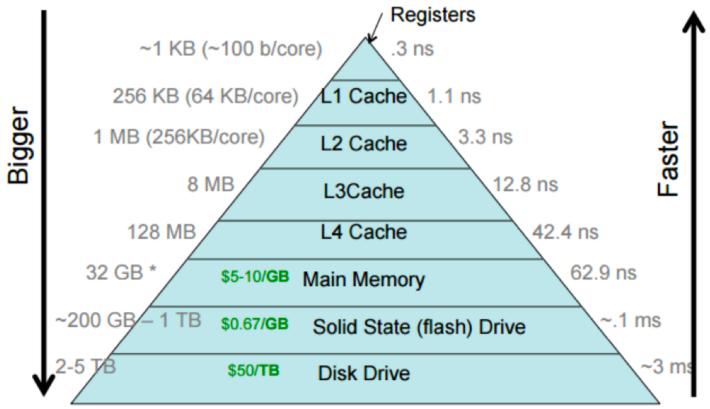


Figure 8.1: The Memories at a Glance, price and latency is higher than 2024

All this levels contains a copy of the same subset of data.

8.0.1 Spatial Locality and Temporal Locality

Spatial Locality: Spatial Locality means that if I read a variable or instruction in memory, all the nearby data have a high chances to be fetched, so when I reed, I reed in blocks and store all the data into a faster memory.

Temporal Locality: Temporal Locality means that if I read a variable the probability of reuse it is high, so I keep it in the cache memory.

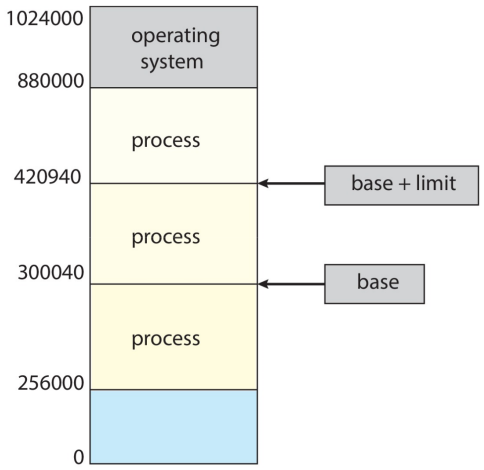
S.No.	Spatial Locality	Temporal Locality
1.	In Spatial Locality, nearby instructions to recently executed instruction are likely to be executed soon.	In Temporal Locality, a recently executed instruction is likely to be executed again very soon.
2.	It refers to the tendency of execution which involve a number of memory locations .	It refers to the tendency of execution where memory location that have been used recently have a access.
3.	It is also known as locality in space.	It is also known as locality in time.
4.	It only refers to data item which are closed together in memory.	It repeatedly refers to same data in short time span.
5.	Each time new data comes into execution.	Each time same useful data comes into execution.
6.	Example : Data elements accessed in array (where each time different (or just next) element is being accessing).	Example : Data elements accessed in loops (where same data elements are accessed multiple times).

8.1 Protection

When a program has became a process, all or most of its data are store in main memory. We need a way to:

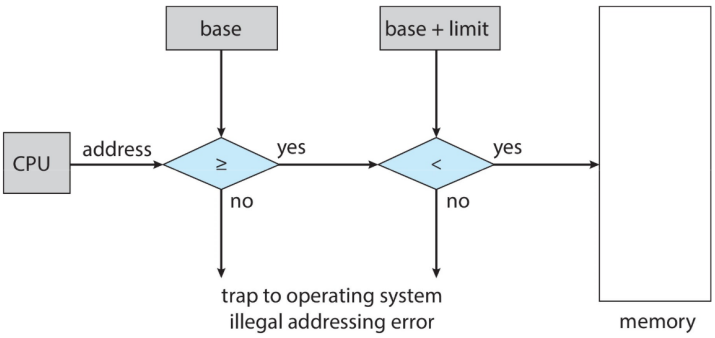
- Split the memory to all the process;
- Need to ensure that a process can access only the addresses in its address space.

We can provide to solve these problems by using a pair of **base** and **limit** registers define the logical address space of a process.



8.1.1 Hardware Address Protection

CPU/OS must check every memory access generated in user mode to be sure it is between base and limit for that user.



The instructions to loading the base and limit registers are privileged.

8.1.2 Address Binding

When we crate a C program we do not take care about where the variable is in memory, we know only that it is in memory.

- Source code addresses usually symbolic;
- Compiled code addresses bind to relocatable addresses, '14 bytes from beginning of this module', Instead of "byte 14';
- Linker or loader will bind relocatable addresses to absolute addresses;
- Each binding maps one address space to another.

In the compile code we see how many space we need and not the hard coded address. We want a dynamically bind and not static.

Binding of Instructions and Data to Memory

When a program is started, the OS give an amount of space for the program.

Address binding of instructions and data to memory addresses can happen at three different stages:

- **Compile time:** If memory location known a priori, absolute code can be generated; must recompile code if starting location changes, used in embedded systems, or systems where the tasks are known a priori and are of a small number;
- **Load time:** Must generate relocatable code if memory location is not known at compile time; Widely used strategy for old OSs or simple systems like dishwasher...
- Execution time: Binding delayed at run time if the process can be moved while executing from one memory segment to another; Need hardware support (e.g., base and limit registers) and this is the De-facto standard for modern OSs.

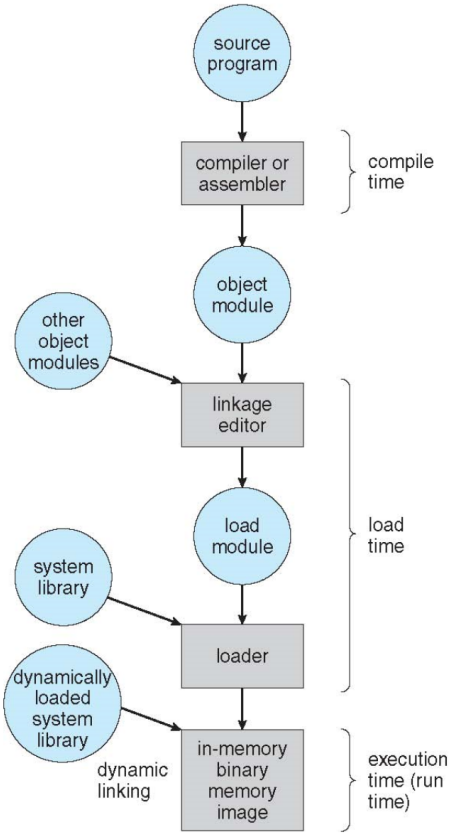


Figure 8.2: Multistep Processing of a User Program

8.2 MEMORY MANAGEMENT UNIT - MMU

I need a method to pass from virtual memory to hardware memory and vice versa, the **binding process**.

8.2.1 Logical vs. Physical Address Space

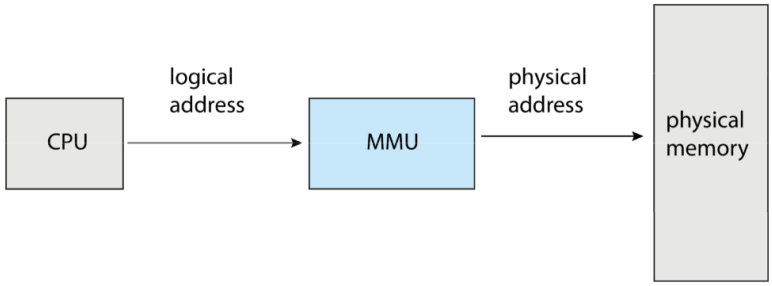
The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management.

- **Logical address** – generated by the CPU; also referred to as **virtual address**;
- **Physical address** – address seen by the memory unit.

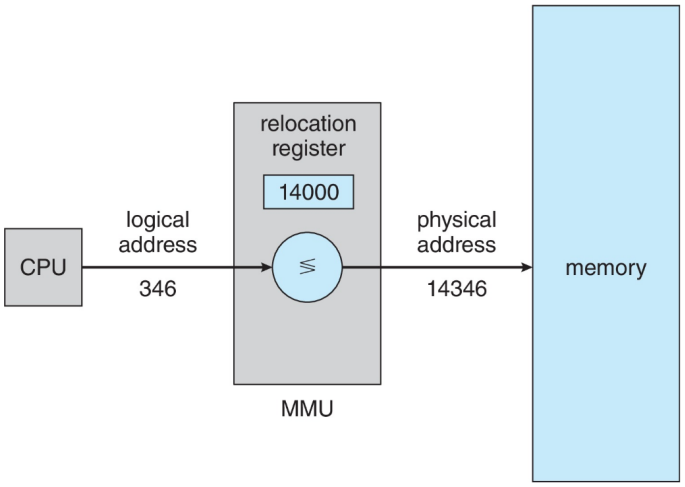
Logical and physical addresses are the same in compile-time and load-time address-binding schemes, but Logical (virtual) and physical addresses differ in execution-time address-binding scheme.

8.2.2 MMU

Hardware device that at run time maps virtual to physical address



The base register now called relocation register. The value in the relocation register is added to every address generated by a user process at the time it is sent to memory.



The user program deals with logical addresses; it **never sees the real physical addresses**. If you print the address of a variable in C program, it print the logical value and not the hardware real value!

8.2.3 Big programs + No Memory Space

What if I have to execute a program that needs 1GB of memory, and I have only 200MB available?

The easiest solution is: throw an error like "No more main memory". But this is not a solution!

8.3 LINKING AND LOADING

8.3.1 Dynamic loading

The entire program does need to be in memory to execute. Routine (part of program) is not loaded until it is called this improve memory-space utilization; unused routine is never loaded. OS can help by providing libraries to implement dynamic loading.

8.3.2 Static vs Dynamic Linking

Static linking – system libraries and program code combined by the loader into the binary program image at compile-time.

Dynamic linking – linking postponed until execution time.

Small piece of code, stub, used to locate the appropriate memoryresident library routine. Stub replaces itself with the address of the routine, and executes the routine at runtime, Operating system checks if the routine is in processes' memory address, if not in address space, add to address space.

Dynamic linking is particularly useful for libraries, System also seen as "shared libraries", this provide small programme because the library are shared by other programme.

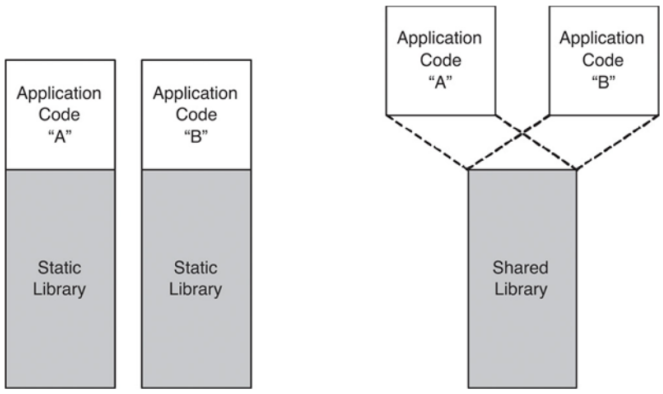


Figure 8.3: .ddl file

Also this method provide an advantage: when the Windows update one library we must compile only this code and not all the other applications that use it.

8.4 Program allocation

8.4.1 Contiguous Allocation

We want to manage efficiently the main memory because main memory must support both OS and user processes and limited resource, must allocate efficiently.

Contiguous allocation is one early method, arrays and linked lists.

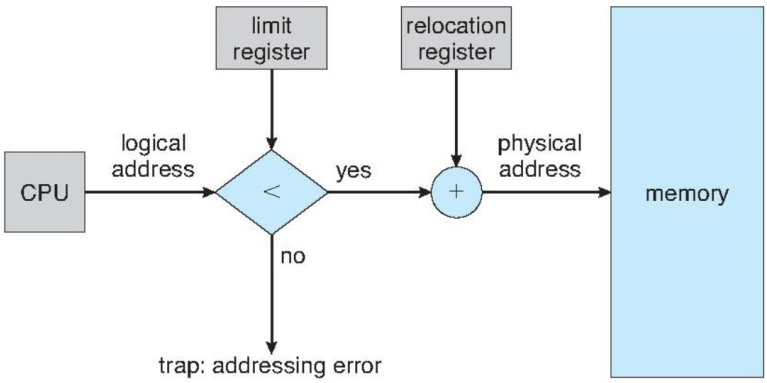
Main memory usually into two partitions:

- Resident operating system, usually held in low memory with interrupt vector;
- User processes then held in high memory

Each process contained in single contiguous section of memory.

Relocation registers used to protect user processes from each other, and from changing operating-system code and data

- Base register contains value of smallest physical address
- Limit register contains range of logical addresses – each logical address must be less than the limit register
- MMU maps logical address dynamically



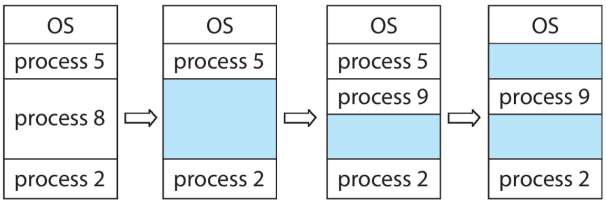
8.4.2 Variable Partition

The OS can decide the Multiple-partition allocation. Degree of multi-programming limited by the number of partitions.

- **Variable-partition** - sizes for efficiency (sized to a given process' needs).
- **Hole** – block of available memory; holes of various size are scattered throughout memory. Undesired, on paper, you want a continuous block of available memory.

When a process arrives, it is allocated memory from a hole large enough to accommodate it, when a process exiting frees its partition, adjacent free partitions combined and all of this is managed by the operating system that maintains information about:

- allocated partitions
- free partitions (hole)



8.4.3 Dynamic Storage-Allocation Problem

How to satisfy a request of size n from a list of free holes?

- First-fit: Allocate the first hole that is big enough

  May produce many small leftover holes

- Best-fit: Allocate the smallest hole that is big enough; must search entire list, unless ordered by size

  Produces the smallest leftover hole, this hole could be not filled anymore.

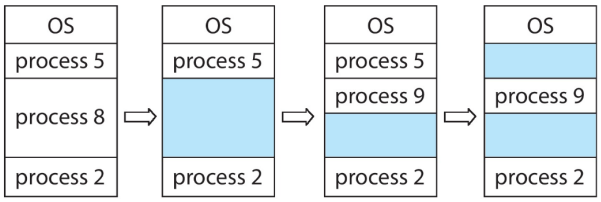
- Worst-fit: Allocate the largest hole; must also search entire list

  Produces the largest leftover hole, it can be filled by other processes.

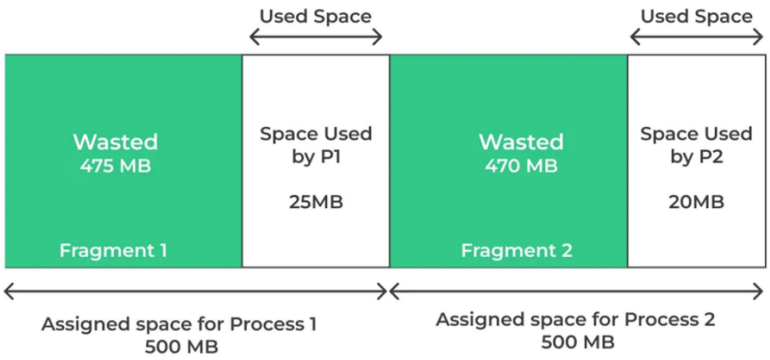
First-fit and best-fit better than worst-fit in terms of speed and storage utilization.

8.4.4 Fragmentation

External Fragmentation – total memory space exists to satisfy a request, but it is not contiguous. Whether you apply a first-fit or best-fit memory allocation strategy it'll cause external fragmentation.



Internal Fragmentation – allocated memory may be slightly larger than requested memory, the minumim size is defined by the OS and the difference between the C programm and this minimum size is defined Internal Fragmentation. If we use dynamic partitioning to allot space to the process, this issue can be solved.



8.4.5 Compaction

Reduce external fragmentation by **compaction**, shuffle memory contents to place all free memory together in one large block. Compaction is possible only if relocation is dynamic, and is done at execution time.

Although the compaction technique is very useful in making memory utilization efficient and reduces external fragmentation of memory, the problem with it is that a large amount of time is wasted in the process and during that time the CPU sits idle hence reducing the efficiency of the system.

8.5 Paging

Suppose that we allow the physical address space of a process can be noncontiguous, the benefit is more. Thus, the process is allocated physical memory whenever the main memory is available. We avoid external fragmentation and varying sized memory chunks.

The paging process consists to divide the physical memory into fixed-sided blocks called **frame**, the size is a power of 2, between 512 bytes and 16 MiB, also the logical memory is divide in the same size blocks called **pages**.

NOTE: is indifferent at this point to talk about if it is a SSD or RAM, is the same, only change the latency and the volatile.

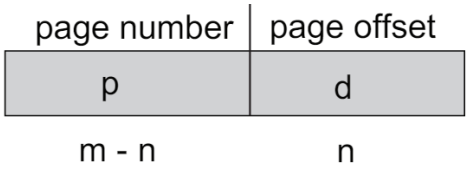
This can allow to keep track all free frames and this can use to run a program of size N pages, need to find N free frames and load program. To do this we must we set up a page table to translate logical to physical addresses.

We still have Internal fragmentation.

8.5.1 Address Translation Scheme

(Virtual) Address generated by CPU is divided into:

- **Page number (p)** – used as an index into a page table which contains base address of each page in physical memory;
- **Page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit.



For given logical address space 2^m and page size 2^n.

8.6 Paging Hardware

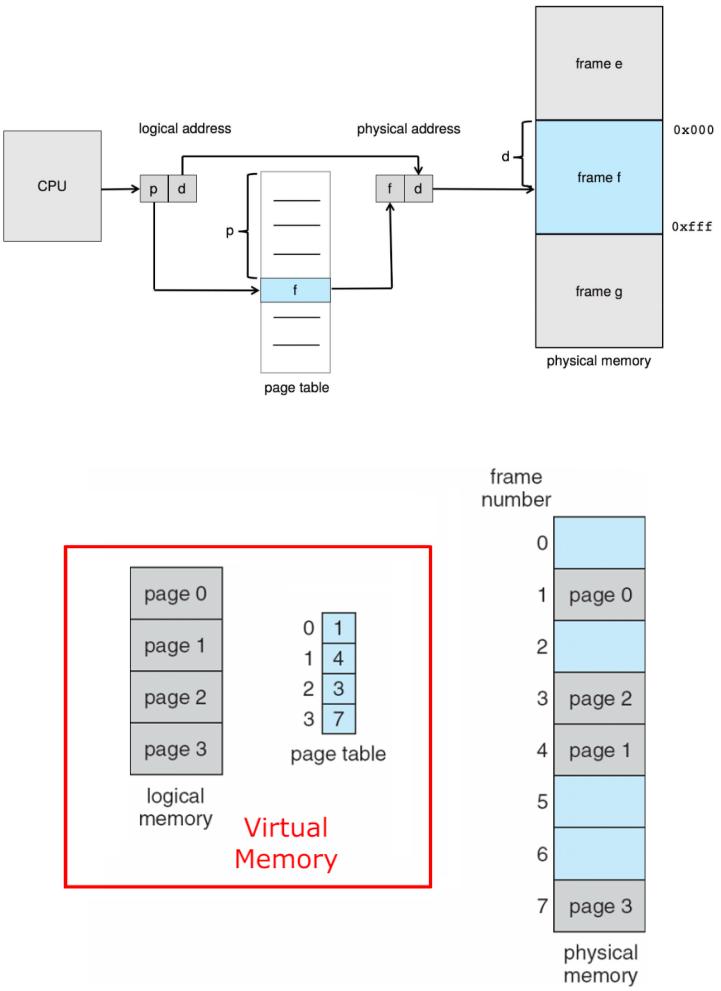
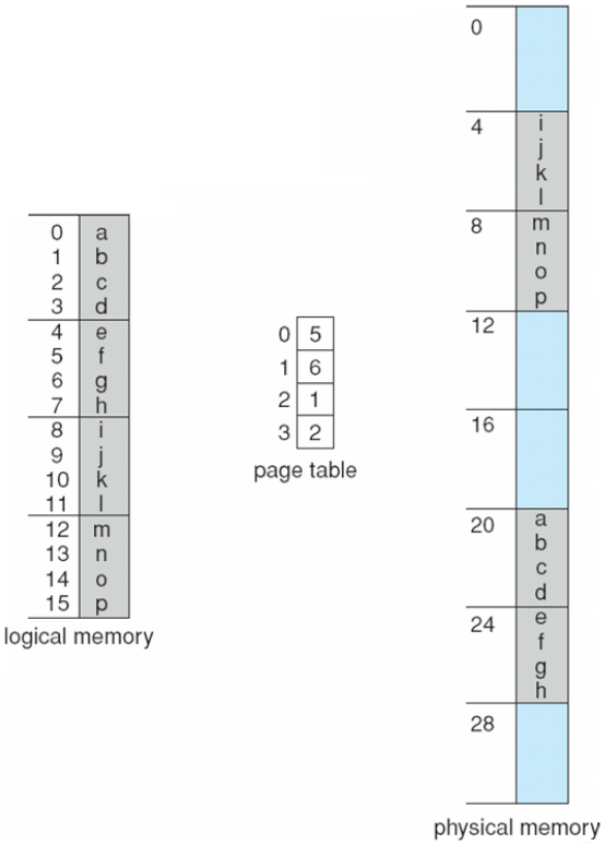


Figure 8.4: Paging Model of Logical and Physical Memory

Paging Example

Logical address: n = 2 (offset bits), n-m = 3 (page bits). Using a page size of 4 bytes and a physical memory of 32 bit (2^m = 8 pages).



Paging - Calculating internal fragmentation

Page size = 2,048 bytes; Process size = 72,766 bytes, we use 35 pages + 1,086 bytes, the internal fragmentation of 2,048 - 1,086 = 962 bytes.

Worst-case scenario = frame size – 1 byte, on average fragmentation = 1 / 2 frame size.

Thus, small frame sizes allow for small internal fragmentation...but the page size became bigger because each page table, per process, takes memory to track: Small page size = many many pages = many memory allocated to track the page and unused for the user/OS tasks.

Page sizes growing over time:

- Solaris supports two page sizes – 8 KB and 4 MB
- Win10 supports two page sizes – 4 KB and 2 MB
- Linux supports two page sizes – 4 KB and custom huge page size

8.6.1 How many pages?

Suppose 4KB (212 bytes) of page size and 32bit hardware CPU. The page table entry is equal to 32 bits. 2^{32} pages, each page is 4KiB, the total of memory addressed is 16 TiB ($2^{12} * 2^{10}$, first is the paging addresses and the second is the offset).

Another example:

$$\text{m} = \text{length address} = 8 \text{ bits.}$$

$$\text{n} = log_2(pagesize) = log_2(32bytes) = 5 \text{ bits of offset.}$$

$$\text{m - n} = 3 = \text{number of page that I can addresses.}$$

If you reduce the size of the page you can addresses more pages, and the offset decreases. Normally m = 32 and the page size is 4KiB, like in the previous example.

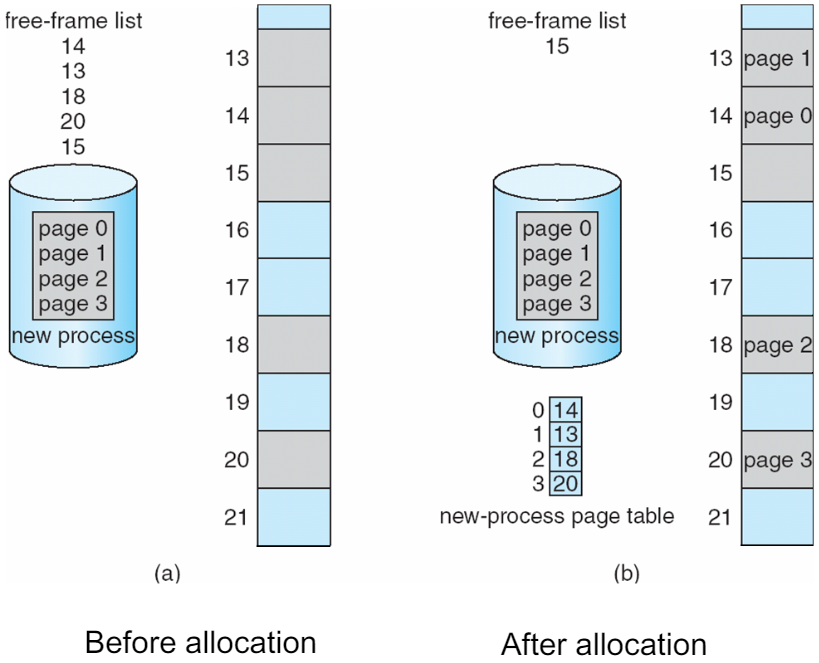


Figure 8.5: Free Frames

8.6.2 Implementation of Page Table

Page table, as we said before, is kept in main memory. It is the main "artifact" of the virtual memory management:

Page-table base register (PTBR) points to the page table

Page-table length register (PTLR) indicates size of the page table

In this scheme every data/instruction access requires two memory accesses! One for the page table and one for the data / instruction.

The two-memory access problem can be solved by the use of a special fast-lookup hardware cache called translation look-aside buffers, **TLBs**.

Exploits **Temporal locality**: the tendency of programs to use data items over and again during the course of their execution.

8.6.3 Hardware solution - Translation Look-Aside Buffer

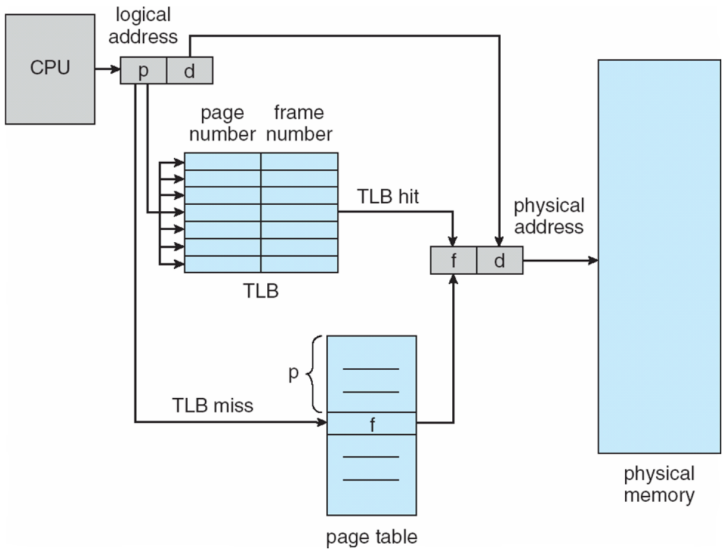
TLBs store **address-space identifiers** (ASIDs) in each TLB entry – uniquely identifies each process to provide addres-space protection for that process, Otherwise need to flush at every context switch.

TLBs typically small, 64 to 1024 entries, but faster, thus, it is a fully-associative cache.

On a TLB miss, value is loaded into the TLB for faster access next time, replacement policies must be considered some entries can be wired down for permanent fast access.

It does not speed up everything:

- Average case: access in TLB + access in memory for data;
- Worst case: no data in TLB, thus access in memory for page table + access in memory for data.



8.6.4 Effective Access Time

Hit ratio – percentage of times that a page number is found in the TLB. An 80% hit ratio means that we find the desired page number in the TLB 80% of the time.

Suppose that you need 10 nanoseconds to access memory. If we find the desired page in TLB then a mapped-memory access take 10 ns, otherwise we need two memory access so it is 20 ns.

Effective Access Time, ETA = 0.80 x 10 + 0.20 x 20 = 12 nanoseconds, implying 20% slowdown in access time.

Consider a more realistic hit ratio of 99%, EAT = 0.99 x 10 + 0.01 x 20 = 10.1ns, implying only 1% slowdown in access time.

8.7 C code and page faults

Think to have the RAM full, only one page is empty, and in this page we want to do this:

```
int main(){
    int[128,128] data;

    ...

    for (j = 0; j <128; j++)
        for (i = 0; i < 128; i++)
            data[i,j] = 0;

    //how many page faults? 128 * 128 = 16384

    ...

    for (i = 0; i < 128; i++)
        for (j = 0; j < 128; j++)
            data[i,j] = 0;

    //how many page faults? only 128

}
```

Page faults - A page fault occurs when a program attempts to access data or code that is in its address space, but is not currently located in the system RAM.

When we access to something into ram the OS brought a block of data from SSD into ram. In the first case the OS take a block but the C code use only the first column; in the second for the OS take a block of data and use all of the column.

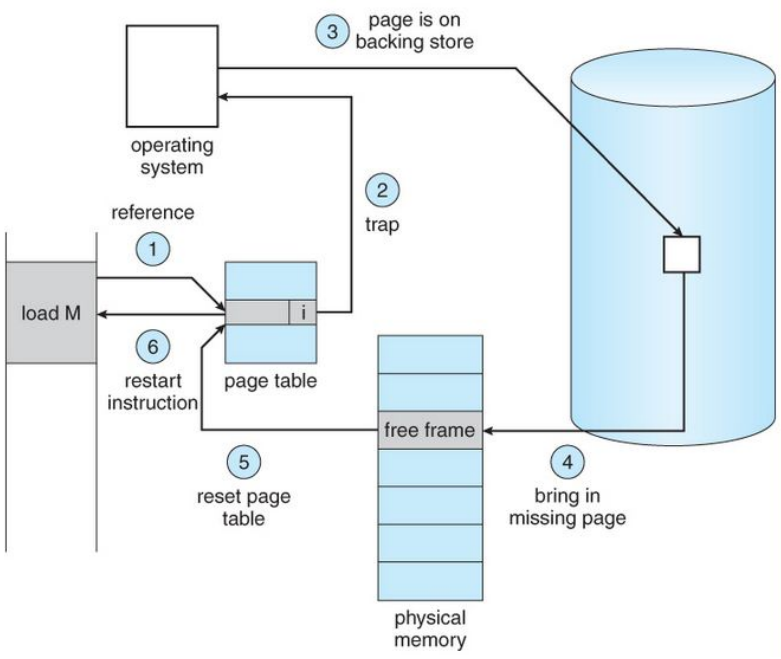


Figure 8.6: Page faults

8.8 Page Faults

What the page faults is? Data is typically stored in non-volatile big and slow memories and then loaded into RAM, it means that data required for execution (a page) may exist only in the secondary storage due to dynamic loading. Thus it needs to be loaded in the RAM, then cache, then registers, this is a **page fault**!

We can not solve it but we try to mitigate this slow event.

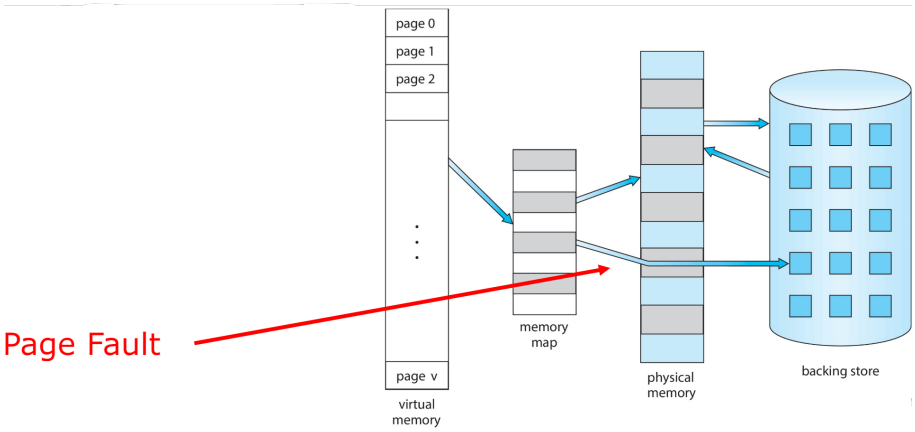
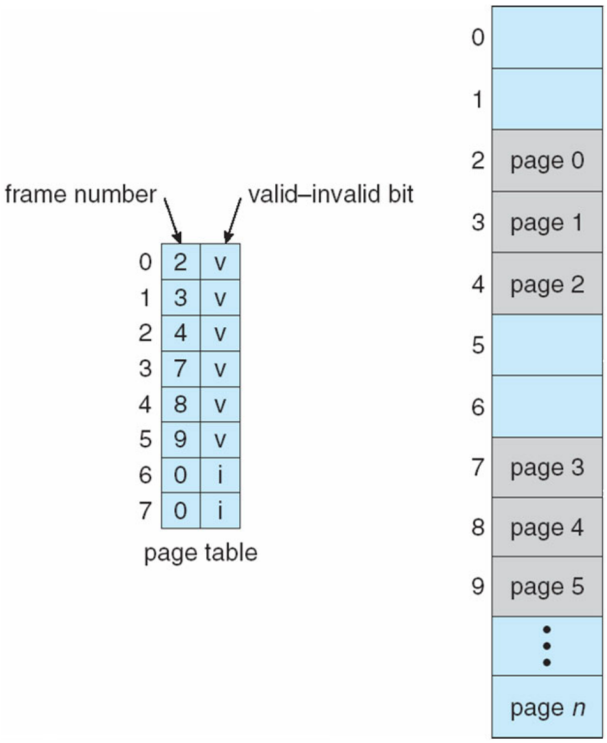


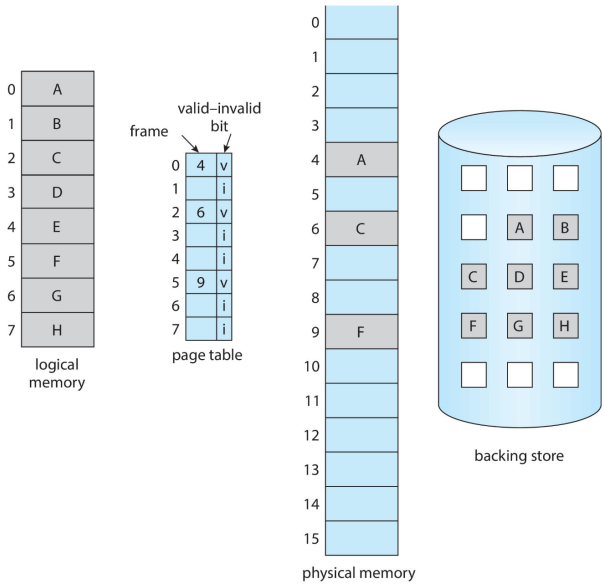
Figure 8.7: Page fault

8.8.1 How detecting page faults?

We can use one bit, **valid-invalid**, to track is the address is in RAM or in secondary memory.

- valid (1): indicate that the associated page is in the process logical address space, and exists in RAM already;
- invalid (0): indicates that the page is not in the RAM, it means **page faults** will happen.





8.8.2 Steps in handling page faults

1. If there is a reference to a page, first reference to that page will trap to operating system = page faults;
2. If there is a reference to a page, first reference to that page will trap to operating system;

   Invalid reference → abort

   Just not in memory, need to be loaded from disk
3. Find a free frame in memory;
4. Swap page into frame via scheduled disk operation, the slowest part of the process;
5. Change tables to indicate page now in memory, Set valid bit = v;
6. Restart the instruction that caused the page fault;

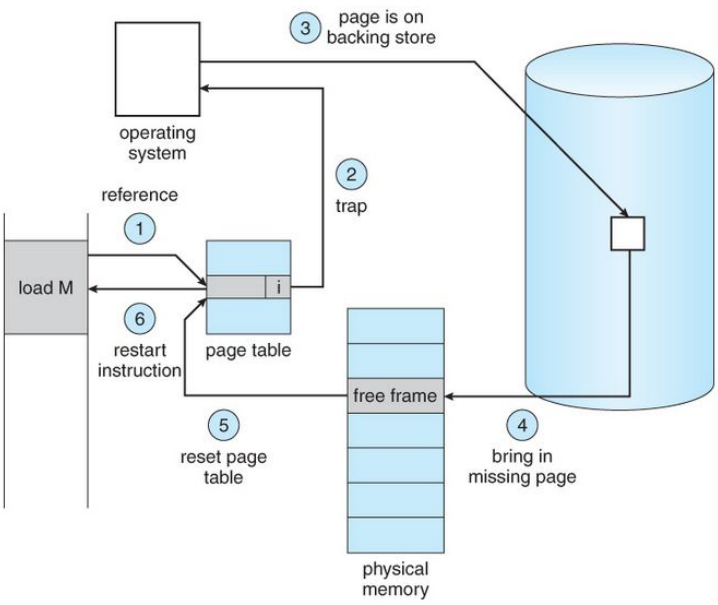


Figure 8.8: Page faults

8.8.3 Cost of a page fault

Assume of Page Fault Rate $0 < p < 1$, if $p = 0$ no page faults, if $p = 1$, every reference is a fault.
Effective Access Time:

$$\text{EAT} = (1 - p) \text{ x memory access} + p \text{ (page fault overhead)}$$

Memory access time = 200 nanoseconds; Average page-fault service time = 8 milliseconds, One access out of 1,000 causes a page fault. It seems a small number but it is not!

Assuming of RAM Memory access time = 200 nanoseconds, average page-fault service time = 8 milliseconds:

$$\text{EAT} = (1 - p)200 + p \text{ (8 milliseconds)} = 200 + p7{,}999{,}800$$

If one access out of 1,000 causes a page fault, then EAT = 8.2 microseconds.
This is a slowdown by a factor of 8.2 / 0.2 = 41!! Not very good!

This is probably the primary source of slowdowns when computing something.

8.9 Page replacement strategy

When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory.

Most operating systems maintain a free-frame list – a pool of free frames for satisfying such requests.

Operating system typically allocate free frames using a technique known as zero-fill-on-demand – the content of the frames zeroedout before being allocated. When a system starts up, all available memory is placed on the freeframe list.

8.9.1 What Happens if There is no Free Frame?

Like for scheduling algorithms we can implement the easy solution using random, took a frame in memory put into the HD and the needed page load into RAM, this solution is not the optimal solution!

This technique, beyond the algorithm, is called **page replacement** - find some page in memory not really in use, push it out and load the needed page.

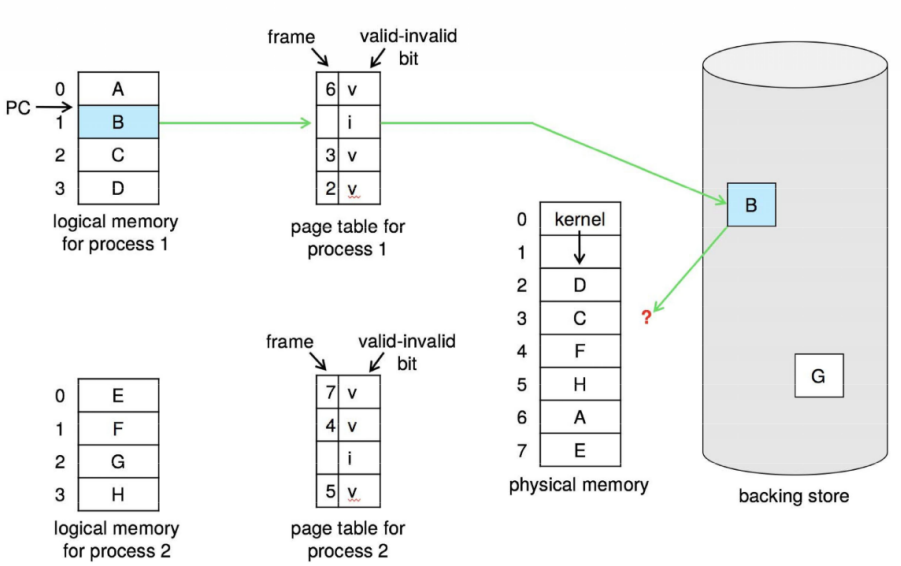
Prevent over-allocation of memory by modifying page-fault service routine to include page replacement.

We can do it using **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk.

If the dirty bit is set means that the value in ram change and before load the new page I need to write the value in the HD.

Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory.

This process means we accesses the HD **two times**.



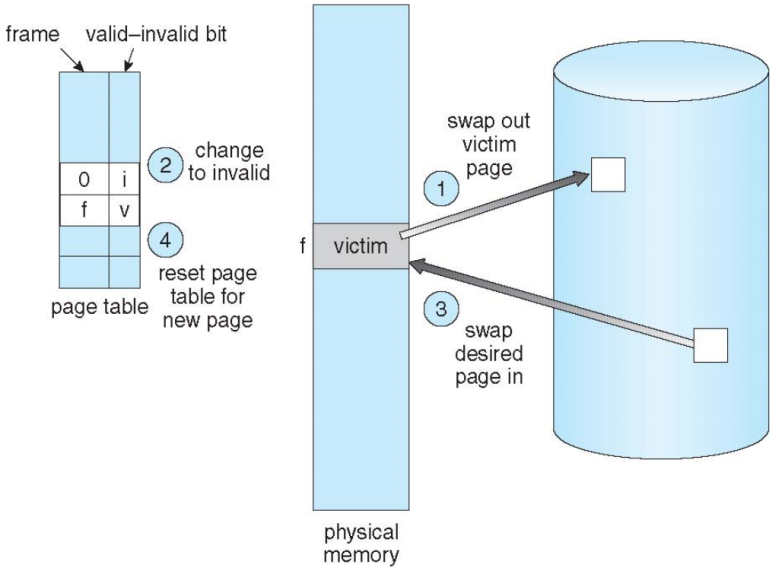
8.9.2 Basic Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame:

   If there is a free frame, use it

   If there is no free frame, use a page replacement algorithm to select a victim frame

   Write victim frame to disk if dirty (= modified while in RAM)
3. Bring the desired page into the (newly) free frame; update the pageand frame tables
4. Continue the process by restarting the instruction that caused the trap



8.10 Page and Frame Replacement Algorithms

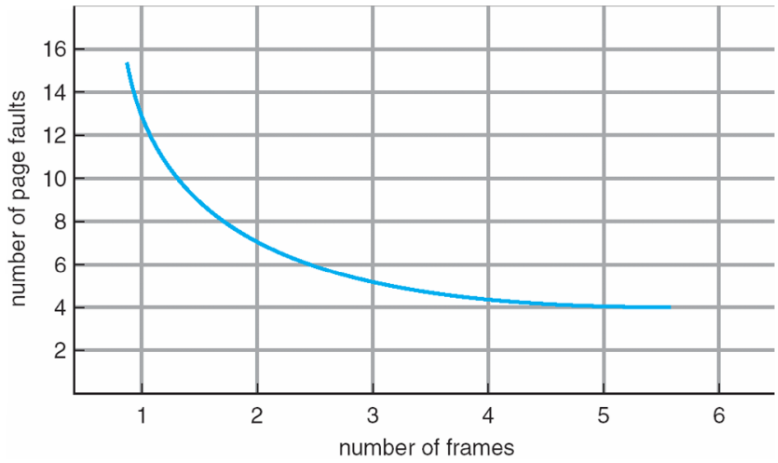
The Frame-allocation algorithm determines :

How many frames to give each process

Which frames to replace, ideally I replace only the page that I do not need soon (but It do not predict the future)

Page-replacement algorithm:

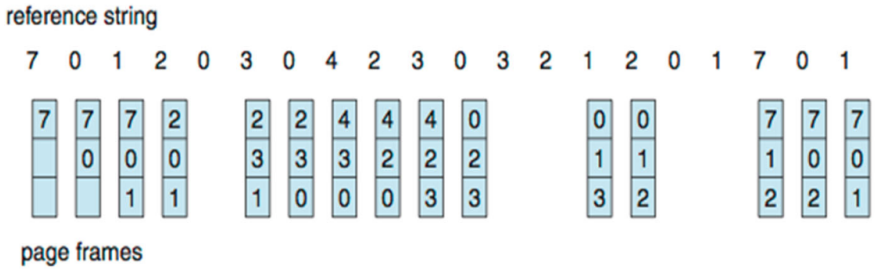
Want lowest page-fault rate on both first access and re-access



In this image we can see that the more frame slots associated to each process, the least the probability of having page faults.

8.10.1 FIFO algorithm

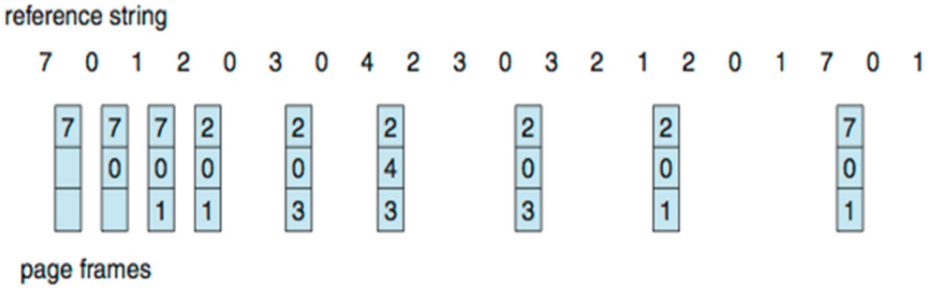
Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1;
3 frames (3 pages can be in memory at a time per process)



With this methods we have 15 page faults, we can do better?

8.10.2 Optimal (?) Algorithm

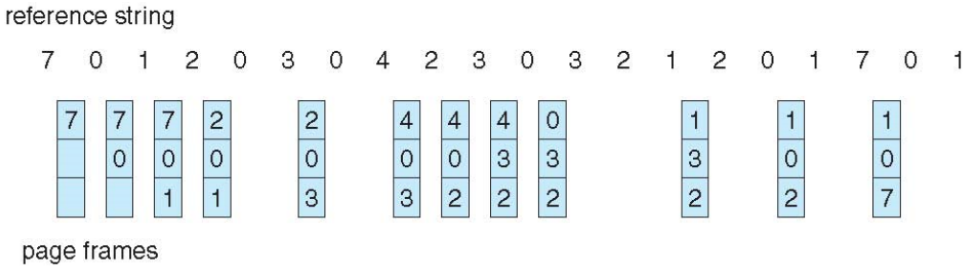
Replace page that will not be used for longest period of time, 9 page faults is optimal best for the example. But like sad before we can't read the future, BUT this is the optimal solution so we can measuring how well the algorithm performs.



8.10.3 Least Recently Used (LRU) Algorithm

We have seen that the program can not read the future, but we can use the past knowledge rather than future!

Replace page that has not been used in the most amount of time, exploits principle of temporal locality. Associate time of last use with each page.



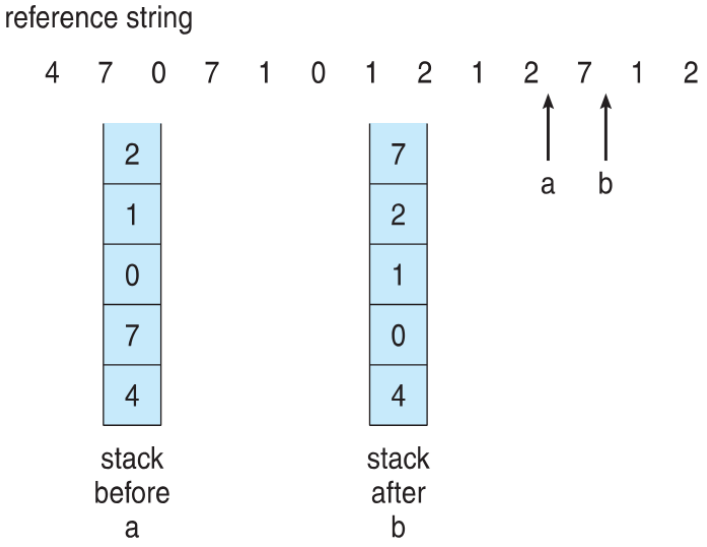
12 faults – better than FIFO but worse than OPT; generally good algorithm and frequently used. But how to implement?

Counter implementation: Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter. When a page needs to be changed, look at the counters to find smallest value, search through table needed.

Stack implementation: Keep a stack of page numbers in a double link form. Page referenced: move it to the top and requires 6 pointers to be changed.

But each update more expensive and no search for replacement.

LRU and OPT are cases of stack algorithms that don't have Belady's Anomaly. Use of a stack to record most recent page references.



8.10.4 LRU Approximation Algorithms

LRU needs special hardware and still slow and memory (4 bytes each entry).

We can simplify using only one bit: **Reference bit**. Initially each page we associate a bit with value 0, than when the page is referenced bit is set to 1. If the OS require memory for process it can replace any page have the bit to value 0 (if one exists).

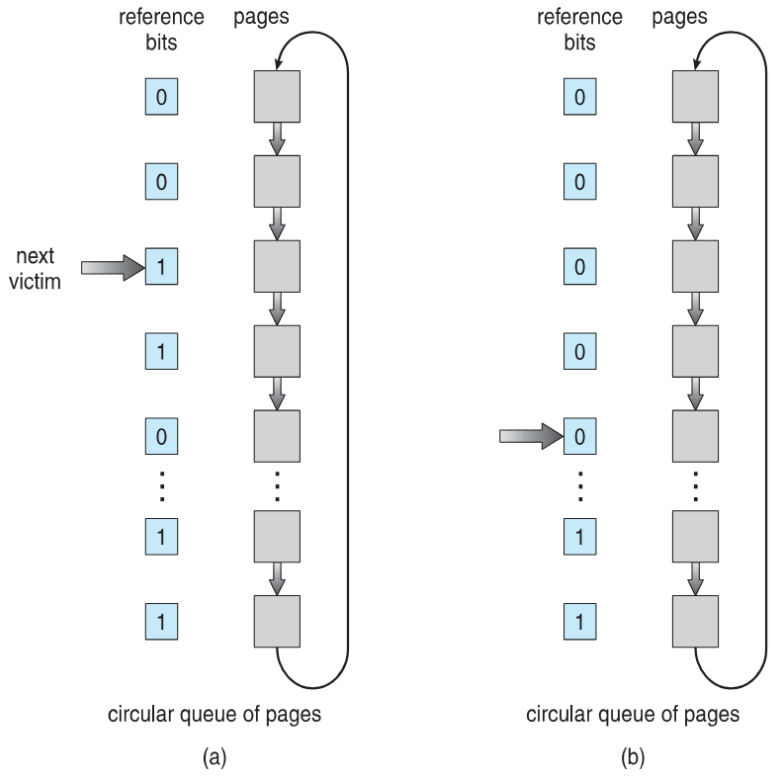
Worst case scenario, the algo. scroll all the list.

Second-chance algorithm: Generally FIFO, plus hardware-provided reference bit with **Clock** replacement. If page to be replaced has:

- Reference bit = 0 -> replace it
- Reference bit = 1 then:

  set reference bit 0, leave page in memory

  replace next page, subject to same rules



8.10.5 Counting Algorithms

Keep a counter of the number of references that have been made to each page, not common.

Least Frequently Used (LFU) Algorithm:

- Replaces page with smallest count
- May have the problem of always replacing the "newly arrived pages" that have the counter set to 1

Most Frequently Used (MFU) Algorithm:

Based on the argument that the page with the smallest count was probably just brought in and has yet to be used

8.11 Allocation of frames to precesses

Each process needs minimum number of frames, the Maximum of course is total frames in the system.

Two major allocation schemes: fixed allocation and priority allocation, but also many variations.

8.11.1 Fixed Allocation

For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames. Keep some as free frame buffer pool.

8.11.2 Proportional Allocation

Allocate according to the size of process, dynamic as degree of multiprogramming, process sizes change

In each two allocation there are no way to account for high/low priority tasks. Whereas ideally you want them to finish early, thus giving them more memory/CPU

8.11.3 Global vs. Local Allocation

Global replacement – process selects a replacement frame from the set of all frames.

- one process can take a frame from another
- process execution time can vary greatly
- greater throughput so more common

Local replacement – each process selects from only its own set of allocated frames

- More consistent per-process performance
- possibly underutilized memory
- Thus, smaller throughtput

8.11.4 Reclaiming Pages

A strategy to implement global page-replacement policy is: all memory requests are satisfied from the free-frame list, rather than waiting for the list to drop to zero before we begin selecting pages for replacement, the Page reclaiming is triggered by the kernel when the list falls below a certain threshold.

The kernel process that manages it is called **reaper**.

This strategy attempts to ensure there is always sufficient free memory to satisfy new requests, or the free-frame list is never empty.

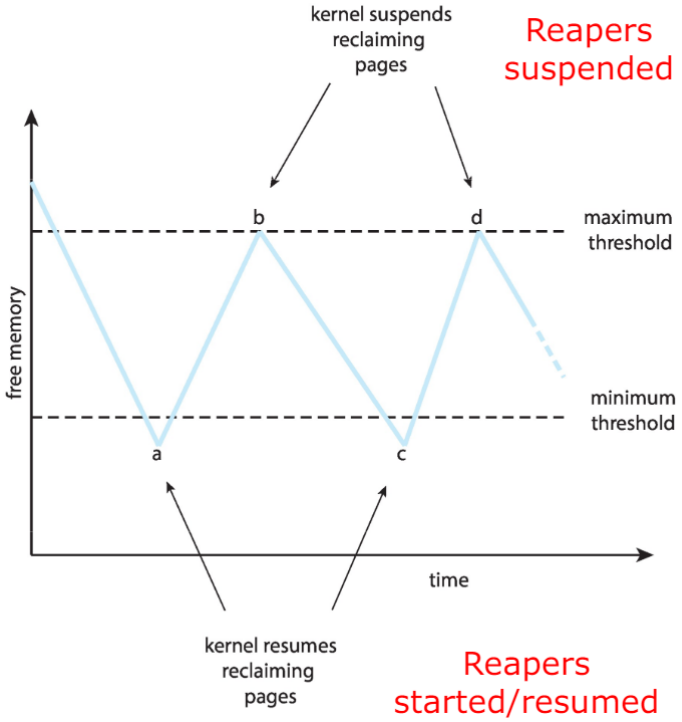


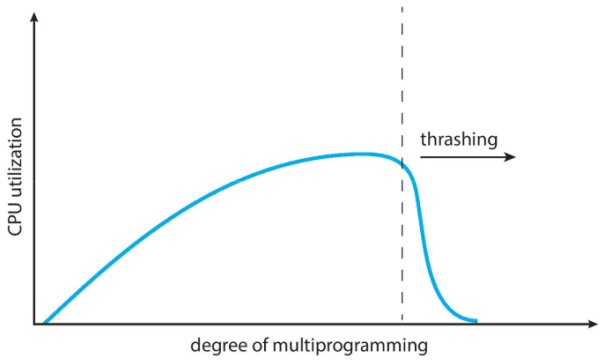
Figure 8.9: Reclaiming Pages Example

8.11.5 Thrashing

Thrashing - a process is busy swapping pages in and out.

If a process does not have "enough" pages, the page-fault rate is very high: **Page fault** to get page, **Replace** existing frame, But quickly need **replaced frame** back.
All of this cycles leads to:

- Low CPU utilization
- Operating system thinking that it needs to increase the degree of multiprogramming
- Another process added to the system



8.11.6 Page-Fault Frequency

Establish "acceptable" page-fault frequency (PFF) rate and use local replacement policy.

If actual rate too low, process loses frame

If actual rate too high, process gains frame

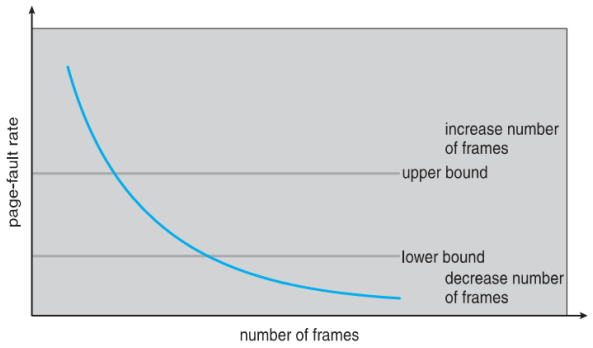


Figure 8.10: PPF

Chapter 9

More on Virtual Memory

9.1 Page swapping

A process can be swapped temporarily out of memory to HD and then brought back into memory for continued execution. The total logical memory space of processes can exceed physical memory.

HD - fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images.

Roll out, roll in - swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed.

Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped. System maintains a **ready queue** of ready-to-run processes which have memory images on disk.

Does the swapped out process need to swap back in to same physical addresses?
Usually not, but depends on address binding method, plus consider pending I/O to / from process memory space.

Each OS can decide how to manage the swap method:

- Swapping normally disabled
- Started if more than threshold amount of memory allocated
- Disabled again once memory demand reduced below threshold

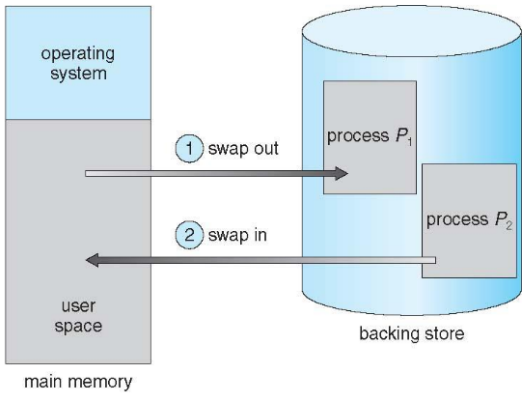


Figure 9.1: Swapping process

9.1.1 Context Switch Time including Swapping

If next processes to be put on CPU is not in memory and RAM is full, you need to swap out a process and swap in target process. The context switch time can then be very high i.e. latency for getting data to HD.

Example: 100MB process swapping to hard disk with transfer rate of 50MB/sec:

Swap out time of 2 seconds

Plus swap in of same sized process

Total context switch swapping component time of 4 s

Can reduce if reduce size of memory swapped – by knowing how much memory really being used. System calls to inform OS of memory use via `request_memory()` and `release_memory()`.

Other constraints: if a task is pending I/O can't swap out as I/O would occur to wrong process, or always transfer I/O to kernel space, then to I/O device but this cause as double buffering, adds overhead.

So how does it work in **modern operating systems**?

Swap only when free memory is extremely low

When using memory pages, entire pages are swapped in and out

Swapped out pages are usually stored in a "paging file"

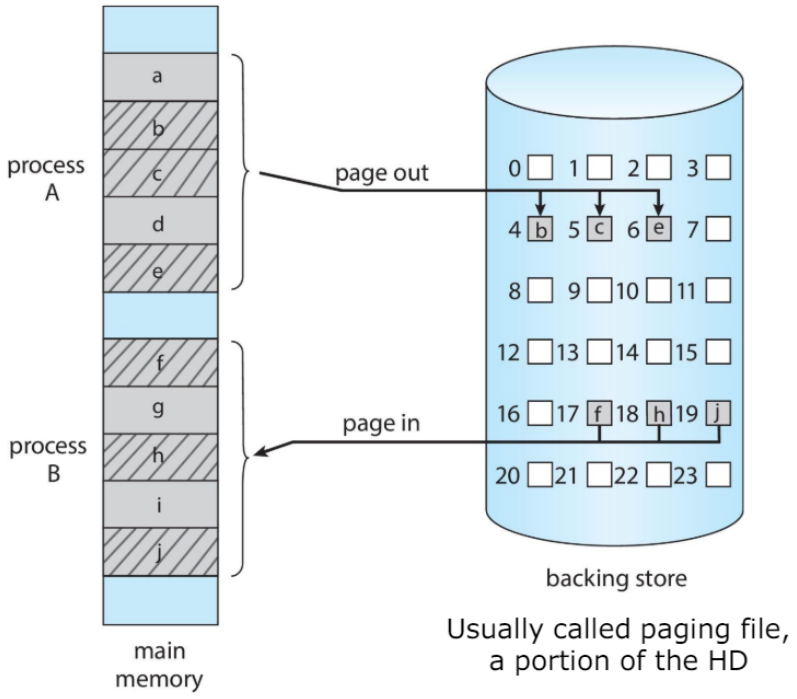


Figure 9.2: Swapping with Paging

9.1.2 Swapping on Mobile Systems

The swapping is not typically used because the storage is small, the flash drive has limited number of write cycles and the bus between the flash memory and the CPU is limited.

So instead of using swapping when the free memory is low:

- **iOS** asks apps to voluntarily relinquish allocated memory, Read-only data thrown out and reloaded from flash if needed. Failure to free can result in termination.
- **Android** terminates apps if low free memory, but first writes application state to flash for fast restart.

Both OSes support paging as discussed below.

9.2 Structure of the Page Table

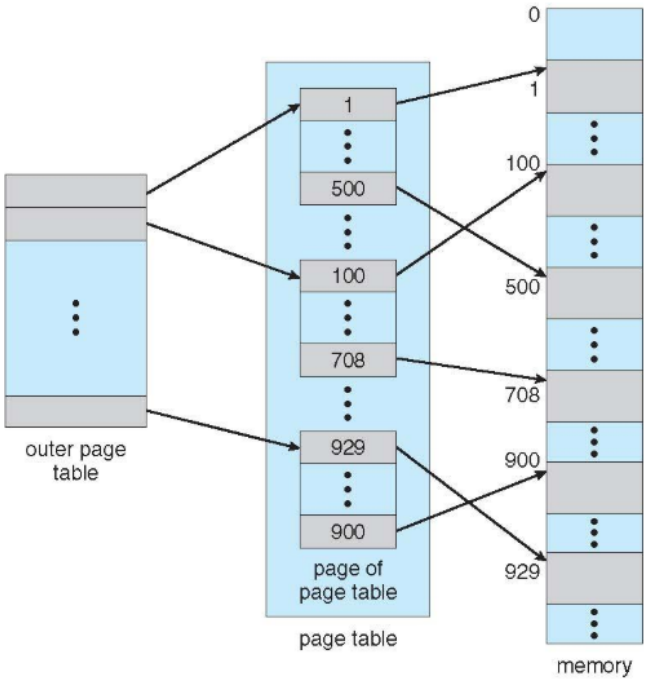
Memory structures for paging can get huge using straight-forward methods. Consider a 32-bit logic address space, page size of 4KB, the page would have 1 million entry the total space for EACH process is 4MB only for get the physical address space.

One simple solution is to divide the page table into smaller units, and loading it only partially:

1. Hierarchical Paging
2. Hashed Page Tables
3. Inverted Page Tables

9.2.1 Hierarchical Page Tables

In this methods we break up the logical address space into multiple page tables. A simple technique is a two-level page table, in this way we load only the section we are interested in.



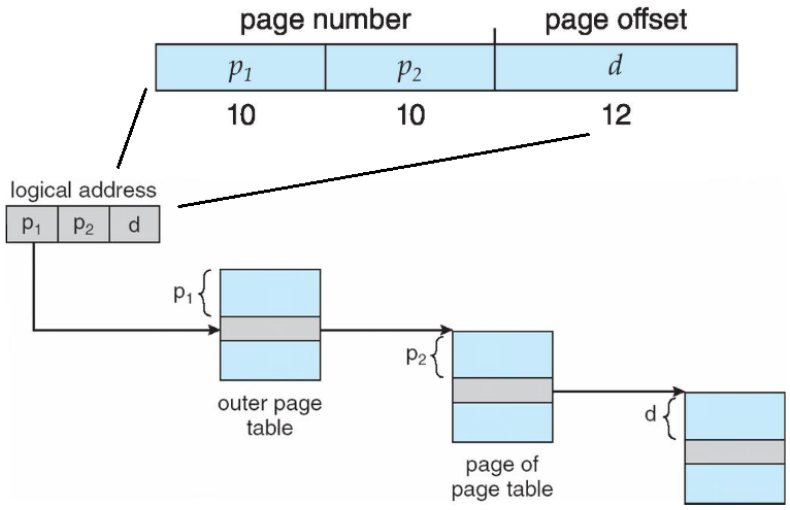
Now the 32-bit with 4K page size, is divides into:

- a page number consisting of 20 bits is divided in two piece of 10-bit each:

  p1 to access the page of page tables, the **outer page**

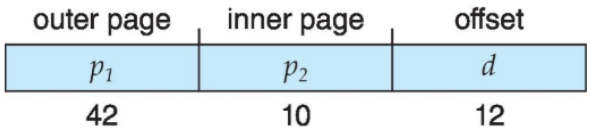
  p2 to access the specific row of the "chunk" of page table, the **inner page**

- a page offset consisting of 12 bits

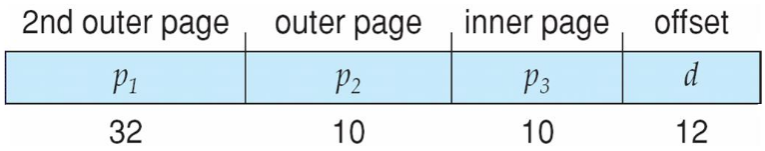


64-bit Logical Address Space

Even two-level paging scheme may not be sufficient, if page size is still 4 KB the page table has 2^{52} entries, the inner page tables could be 2^{10}:



The outer page table has 2^{42} entries witch is 4 terabytes per process! One solution could be add another layer of outer page table:



The problem here is the 4 memory access to get one physical memory location, this is not a brilliant idea.

9.2.2 Hashed Page Tables

Common address spaces is > 32-bit. The virtual page number is hashed into a page table, this page table contains a list of elements hashed to the same location.

Each elements contains:

- the virtual page number
- the value of the mapped page frame
- a pointer to the next element

If you have and address space of 64-bit, 12 are offset, so 52 for page address. You can use an hash table of 4Kb (2^{12}), which typically fits in a single page and always stays in memory for each process.

Then go to the specific page through it. If all the memory is used by the process, this is not efficient. However, in the vast majority of cases no process will fully use all 2^{52} pages.

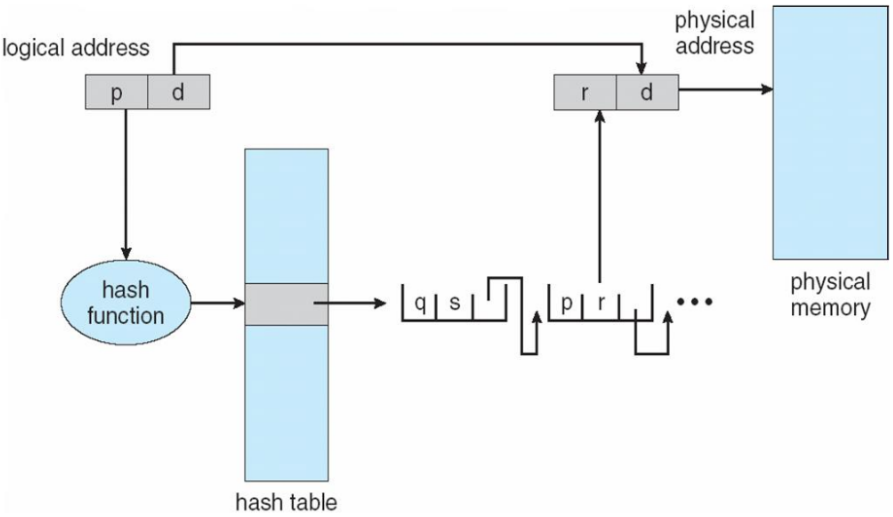


Figure 9.3: Hashed Page Table

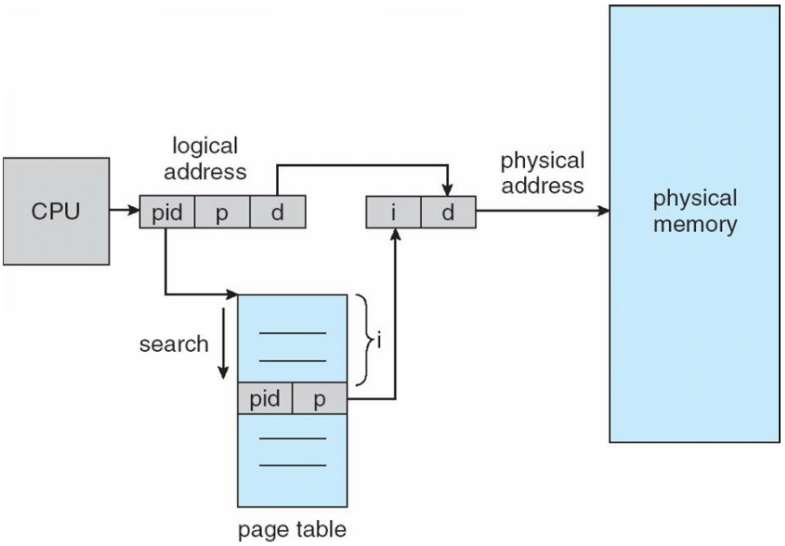
9.2.3 Inverted Page Table

Rather than each process having a page table and keeping track of all possible logical pages, one entry for each real page of memory!

Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page

Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs.

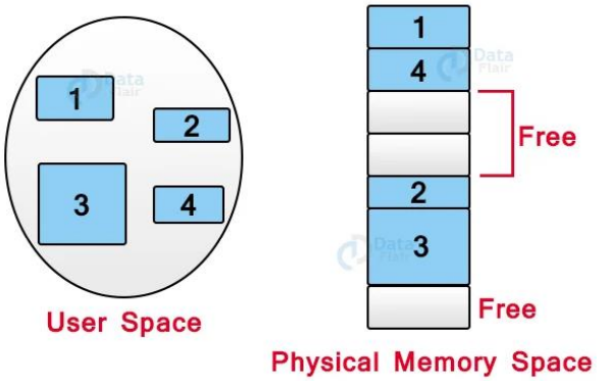
One solution could be using hashing to limit the search to one, also TLB can accelerate access.



9.3 Segmentation

Using pages of fixed sizes opens the problem of internal fragmentation, each process is matched to a given amount of pages rather than memory it needs.

So, is there a way to assign exactly the amount of memory a process needs, thus avoiding any kind of memory waste? The answer is using **segmentation**.



A process is divided into chunks, the chunks that a program is divided into, which are not necessarily all of the exact sizes, are called segments.

- A chunk for stack
- A chunk for heap
- A chunk for library 1

Each process is divided into several segments, all of which are loaded into memory at run time, though not necessarily contiguously.

A table stores the information about all such segments: the base and the limit (length).

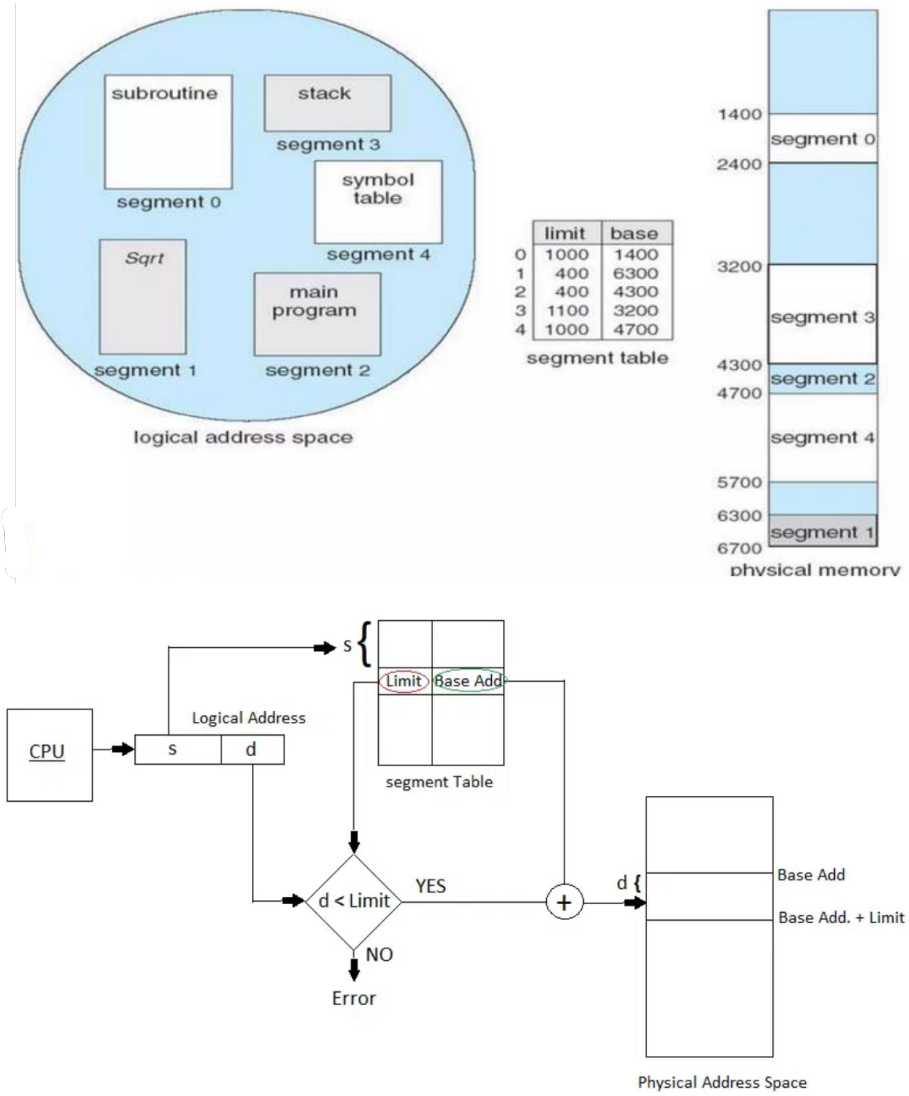
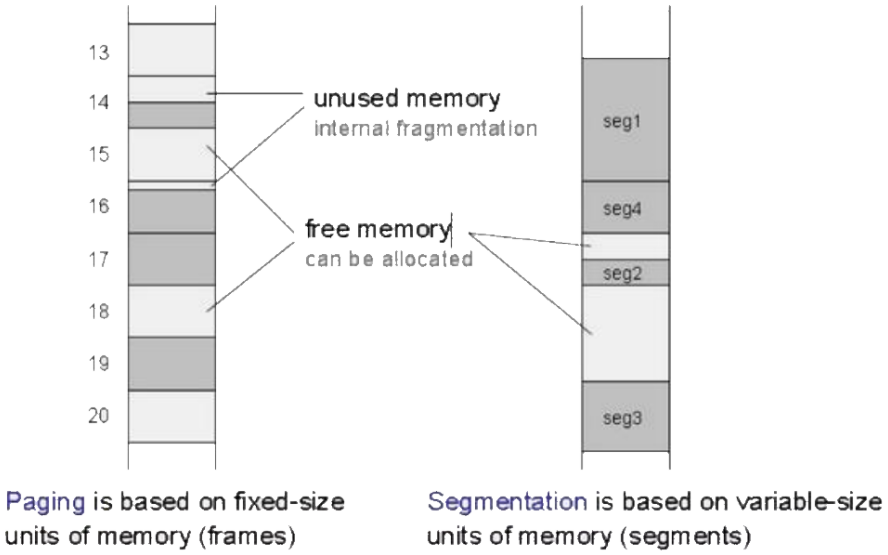


Figure 9.4: Segmentation: logical to physical

9.3.1 Paging vs Segmentation

Feature	Paging	Segmentation	Example
Usage	Systems with a small amount of memory to prevent external fragmentation.	Systems with a large amount of memory to provide better security and flexibility.	A word processing program with a small memory footprint would be a good candidate for paging. A database program with a large memory footprint would be a good candidate for segmentation.
Working	The program is divided into pages and stored in the main memory. When the program needs to access a page, the operating system transfers it from the secondary memory to the main memory.	The program is divided into segments and stored in the secondary memory. When the program needs to access a segment, the operating system transfers it from the secondary memory to the main memory.	A program that is 100 pages long is divided into 4KB pages in paging. In segmentation, the program is divided into 3 segments: code, data, and stack.
Page size	Fixed	Variable	In paging, the page size is typically 4KB. In segmentation, the segment size can be any size.
Loading of pages	Dependent	Independent	In paging, all pages of a process must be loaded into memory before the process can be executed. In segmentation, pages can be loaded into memory independently.
Prevention of internal fragmentation	No	Yes	Paging can lead to internal fragmentation because a free page may not be large enough to accommodate a requesting page. Segmentation can prevent internal fragmentation because segments can be of different sizes.
Prevention of external fragmentation	Yes	No	Paging can prevent external fragmentation because pages are loaded into memory in blocks. Segmentation cannot prevent external fragmentation because segments can be loaded into memory independently.
Memory management overhead	Lower	Higher	The memory management overhead in paging is lower than in segmentation because paging is a simpler technique.
Security	Lower	Higher	Paging cannot provide as much security as segmentation because segments can be loaded into memory independently.
Flexibility	Lower	Higher	Paging is less flexible than segmentation because pages must be loaded into memory in blocks.
Performance	Faster	Slower	Paging is faster than segmentation because it is a simpler technique.



Example of Intel IA-32 Architecture

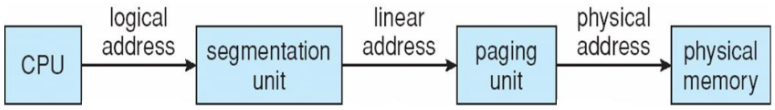
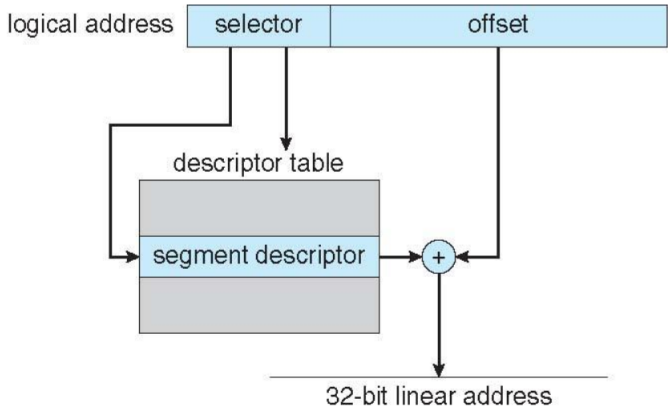


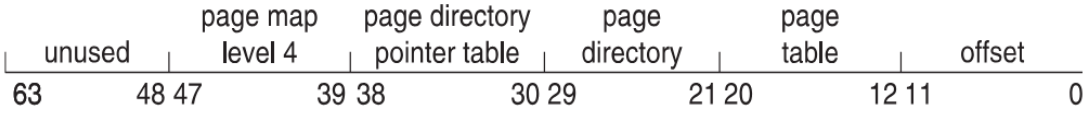
Figure 9.5: Memory address conversion

Paging units form equivalent of MMU, pages sizes can be 4 KB or 4 MB, two-level page tables.



Example of Intel Intel x86-64

Current generation of CISC architecture has 64 bits, this is ginormous (> 16 exabytes). In practice only implement 48 bit addressing.



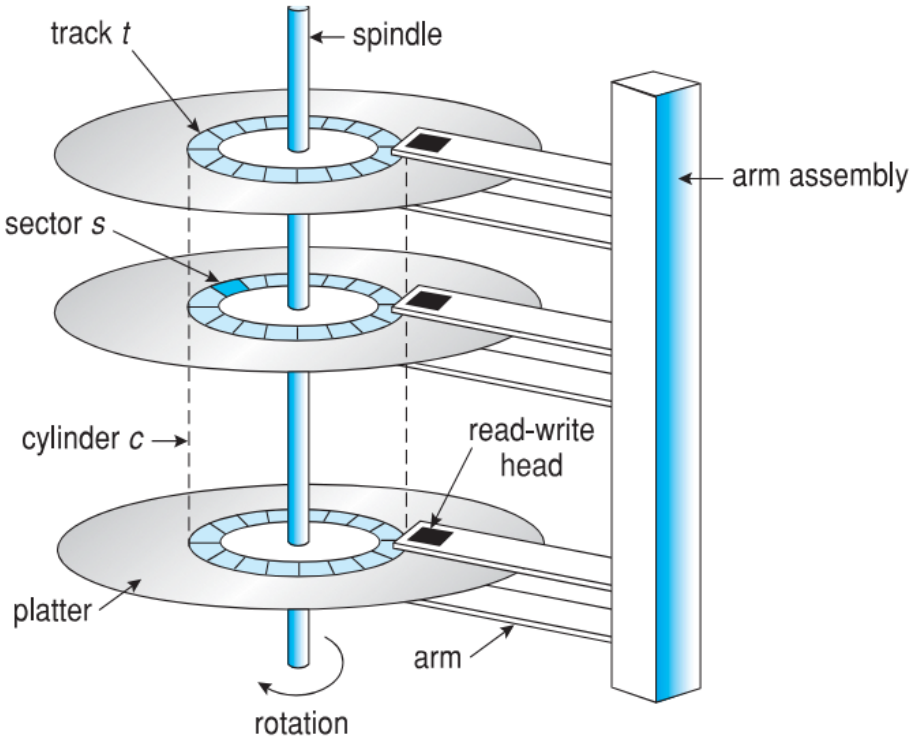
Chapter 10

Mass-Storage Systems

Bulk of secondary storage for modern computers is **hard disk drives**, HDD, and **nonvolatile memory**, NVM, devices.

HDDs spin platters of magnetically-coated material under moving read-write heads:

- Drives rotate at 60 to 250 times per second
- Transfer rate is rate at which data flow between drive and computer, $\approx$ 1 Gb/sec
- Positioning time (random-access time) is time to move disk arm to desired cylinder (seek time) and time for desired sector to rotate under the disk head (rotational latency), from 3ms to 12ms
- Head crash results from disk head making contact with the disk surface – That's bad



Access Latency (Average access time) = average seek time + average latency

For fastest disk: 3 ms + 2 ms = 5 ms.

Average I/O time = average access time + (amount to transfer / transfer rate) + controller overhead

Esample: o transfer a 4KB block on a 7200 RPM disk with a 5ms average seek time, 1Gb/sec transfer rate with a .1ms controller overhead.

At 7200 RPM = 120 rev/sec = 8.333ms/rev, in the worst case, we have to wait 1 full revolution, but on average we have to wait a half-revolution or 4.17ms.
Transfer time = 4KB / 1Gb/s * 8Gb / GB * 1GB / 10242KB = 32 / (1024^2) = 0.031 ms
Average I/O time for 4KB block = 5ms + 4.17ms + 0.1ms + .031ms = 9.301ms

10.1 Nonvolatile Memory Devices

The NVM is intendef for the solid-state drives and the USB drive. This device are more reliable and **faster** than HDD but more expensive.

Read and written in "page" increments (think sector) but can't overwrite in place, must first be erased, and erases happen in larger "block" increments. This device can be erased a limit number of times called **drive writes per day**, DWPD.

The SSD is a complex device, it mount processor to look if all sector is in good conditions, erase and write block ecc.

10.2 HD Scheduling

The operating system is responsible for using hardware efficiently, for the disk this means having a fast access time and disk bandwidth. The **bandwidth** is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.

There are many sources of disk I/O request: OS, System processes, Users processes; all includes write/read, disk address, memory address, number of sectors to transfer. The OS maintains a queue of request per device.

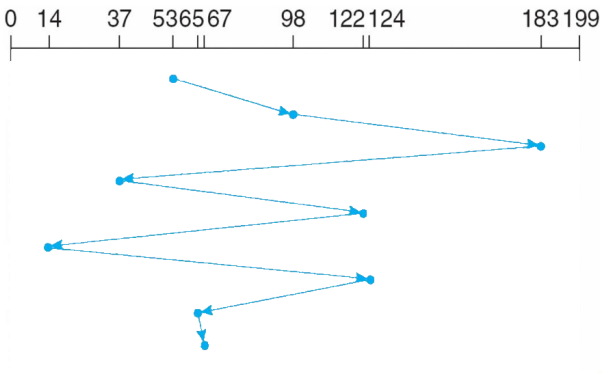
In the past the HD is not sophisticated like now, before the OS must take care of all of thinks: queue management and disk drive head scheduling. Now the control is up to the HD, the OS must pass only the block addresses and sorting the request.

Esample: we want to manage the following list of request using different scheduling algorithm:

98, 183, 37, 122, 14, 124, 65, 67

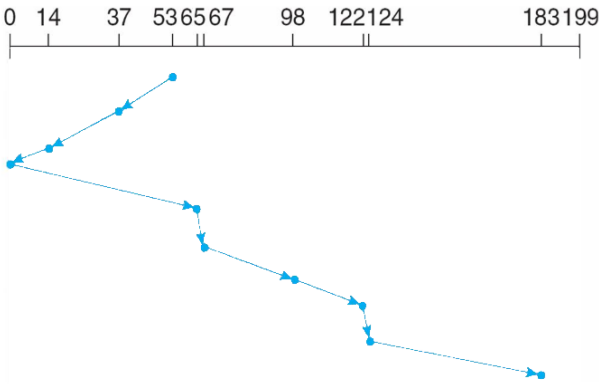
Head point is set to 53 at beginning

10.2.1 FCFS



10.2.2 SCAN - elevator algorithm

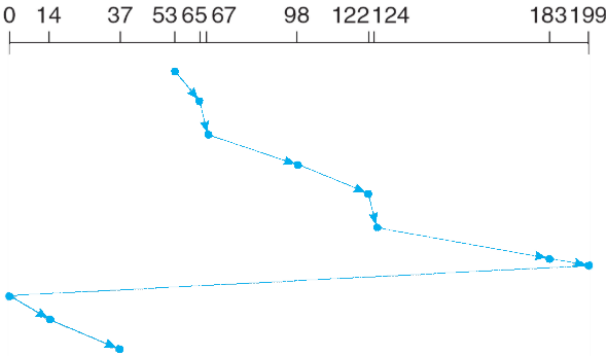
The disk arm start at one end of the disk, and moves toward the other end. This is not equal, the last value have to wait a lot of time.



NOTE: there are no request from 0-65, we can just skip it.

10.2.3 C-SCAN

Provides a more uniform wait time than SCAN, the head moves from one end of the disk to the other, servicing requests as it goes. When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip.



10.2.4 Chooseing Disk-Scheduling Algorithm

- FCFS is common and has a natural appeal
- SCAN and C-SCAN perform better for systems that place a heavy load on the disk

Linux implements deadline scheduler

- Maintains separate read and write queues
- By default, read batches take precedence over write batches, because applications are more likely to block on read I/O operations.
- Schedules read and write batches for execution in increasing logical block addressing (LBA)
- After each batch, it naturally select read batches if any, but also checks how long write operations have been waiting (to avoid starvation)
- schedules the next read or write batch as appropriate
- Typically, if read batch is waiting for more than 500ms, gets priority over read

This scheduler is suitable for most use cases, but particularly those in which the write operations are mostly **asynchronous**.

10.3 NVM Scheduling

No disk heads or rotational latency but still room for optimization. NVM best at random I/O, HDD at sequential, the Input/Output operations per second (IOPS) much higher with NVM (hundreds of thousands vs hundreds).

10.4 Device management

One fundamental aspect of many parts of computing is the **Error Detection and Correction**.

The **Error detection** determines if there a problem has occurred, frequently used via parity bit.

Parity one form of checksum and another error-detection method is cyclic redundancy check (CRC) which uses hash function to detect multiple-bit errors.

Error-correction code, ECC, not only detects, but can correct some errors, soft errors correctable, hard errors detected but not corrected.

10.4.1 Storage Device Management

Low-level formatting, or **physical formatting** - dividing a disk into sectors that the disk controller can read and write. Each sector can hold header information, plus data, plus error correction code. Usually 512 bytes of data but can be selectable.

To use a disk to hold files, the operating system still needs to record its own data structures on the disk:

- **Partition** the disk into one or more groups of cylinders, each treated as a logical disk
- **Logical formatting** or "making a file system"

To increase efficiency most file systems group blocks into clusters. Disk I/O done in blocks, File I/O done in clusters.

Boot/root partition contains the OS, other partitions can hold other Oses, other file systems, or be raw. The partition can be **mounted** at boot time, or automatically or manually.

At mount time, file system consistency checked, all metadata correct?

If not, fix it, try again

If yes, add to mount table, allow access

Boot block, or a boot management program for multi-os booting, can point to boot volume or boot loader set of blocks that contain enough code to know how to load the kernel from the file system.

Boot block initializes system, it is stored in ROM, firmware. The Bootstrap loader program stored in boot blocks of boot partition, master boot record MBR.

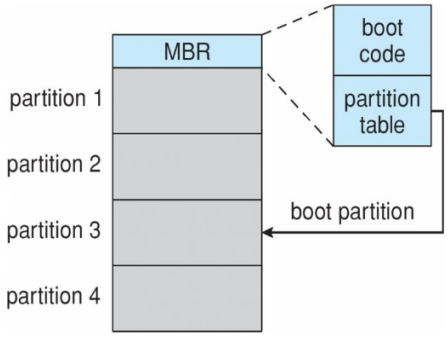


Figure 10.1: Booting from secondary storage in Windows

10.4.2 Swap-Space Management

Used for moving entire processes or pages from DRAM to secondary storage when DRAM not large enough for all processes. The Operating system provides **swap space management**:

- Secondary storage slower than DRAM, so important to optimize performance
- Usually multiple swap spaces possible – decreasing I/O load on any given device
- Best to have dedicated devices
- Can be in raw partition or a file within a file system (for convenience of adding)

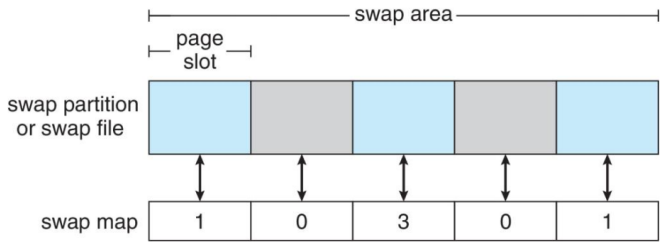


Figure 10.2: Data structures for swapping on Linux systems

10.4.3 Host Storage Attachment

Computers access storage in three ways:

- host-attached
- network-attached
- cloud

Host attached access through local I/O ports, using one of several technologies to attach many devices (USB, LAN...).

Network-Attached Storage: NAS

Network-attached storage (NAS) is storage made available over a network rather than over a local channel (such as a bus). **NFS** and **CIFS** are common protocols, implemented via remote procedure calls (RPCs) between host and storage over typically TCP or UDP on IP network.

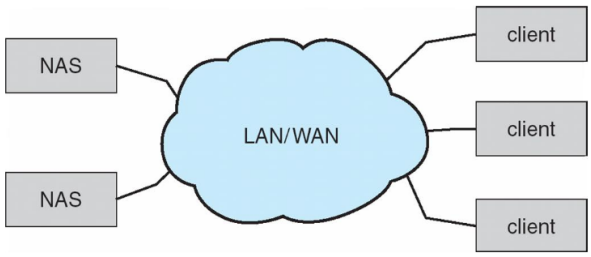


Figure 10.3: NAS

10.4.4 Cloud Storage

Similar to NAS, provides access to storage across a network. Unlike NAS, accessed over the Internet or a WAN to remote data center.

NAS presented as just another file system, while cloud storage is API based, with programs using the APIs to provide access. Use APIs because of latency and failure scenarios (NAS protocols wouldn't work well).

10.4.5 Redundant Array of Independent Disks - RAID

RAID - multiple disk drives provides reliability via redundancy.

- Increases the **mean time to failure**.
- **Mean time to repair** – exposure time when another failure could cause data loss
- **Mean time to data** loss based on above factors

If mirrored disks fail independently, consider disk with 100,000 mean time to failure and 10 hour mean time to repair. The mean time to data loss is:

$$\text{MTDL} = 100\,000^2/(2 * 10) = 500 * 10^6 \text{ hours or } 57\,000 \text{ years!}$$

Several improvements in disk-use techniques involve the use of multiple disks working cooperatively.

Disk **striping** uses a group of disks as one storage unit. RAID is arranged into six different levels.

RAID schemes improve performance and improve the reliability of the storage system by storing redundant data

- **Mirroring** or shadowing (**RAID 1**) keeps duplicate of each disk
- Striped mirrors (**RAID 1+0**) or mirrored stripes (**RAID 0+1**) provides high performance and high reliability
- Block interleaved parity (**RAID 4, 5, 6**) uses much less redundancy

RAID within a storage array can still fail if the array fails, so automatic replication of the data between arrays is common. Frequently, a small number of hot-spare disks are left unallocated, automatically replacing a failed disk and having data rebuilt onto them.

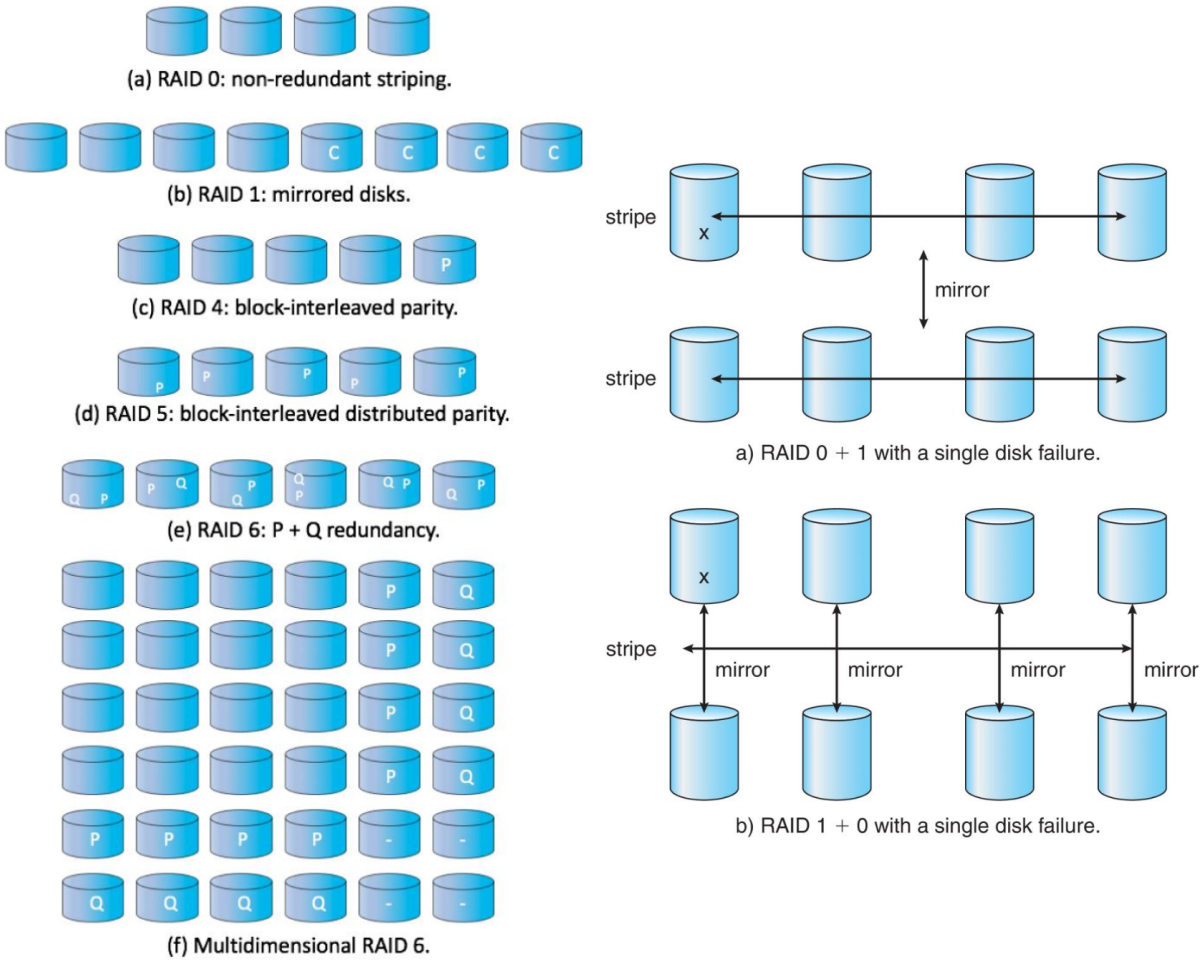


Figure 10.4: RAID schemes

Chapter 11

File-System Interface

11.1 What's a File?

Contiguous logical address space. Types:

- Data File
  - Numeric
  - Character
  - Binary
- Program File

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

Figure 11.1: Type of files

11.1.1 File Attributes

- **Name** – only information kept in human-readable form
- **Identifier** – unique tag (number) identifies file within file system
- **Type** – needed for systems that support different types
- **Location** – pointer to file location on device
- **Size** – current file size
- **Protection** – controls who can do reading, writing, executing
- **Time**, **date**, and **user identification** – data for protection, security, and usage monitoring

Information about files are kept in the **directory structure**, which is maintained on the disk. Many variations, including extended file attributes such as file checksum.

11.1.2 File Structure

- None - sequence of words, bytes
- Simple record structure - Lines, Fixed length, Variable length...
- Complex Structures - Formatted document Relocatable load file

Can simulate last two with first method by inserting appropriate control characters.
The decision of the structure is up to the **OS** or **Program**.

11.2 File access and operation

Several pieces of data are needed to manage **open files**:

- **Open-file table**: tracks open files
- File pointer: pointer to last read/write location, per process that has the file open
- **File-open count**: counter of number of times a file is open – to allow removal of data from open-file table when last processes closes it
- Disk location of the file: cache of data access information
- Access rights: per-process access mode information

11.2.1 File Locking

Provided by some **operating systems** and file systems. Similar to reader-writer locks.
Shared lock similar to reader lock – several processes can acquire concurrently **Exclusive lock** similar to writer lock

Mandatory – access is denied depending on locks held and requested **Advisory** – processes can find status of locks and decide what to do

Example: try to write a file which is already opened in Excel, you get and error; if you try to open a file in gedit, usually not.

11.2.2 Access Methods

A file is fixed length **logical record(s)**:

- Sequential Access
- Direct Access
- Other Access Methods

Sequential Access

Operations: read next, write next, Reset, no read after last write (rewrite), No "seek".

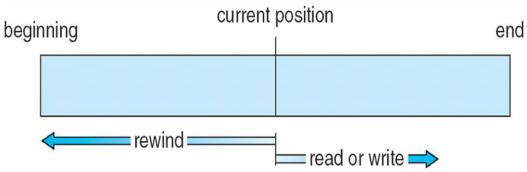


Figure 11.2: Sequential Access

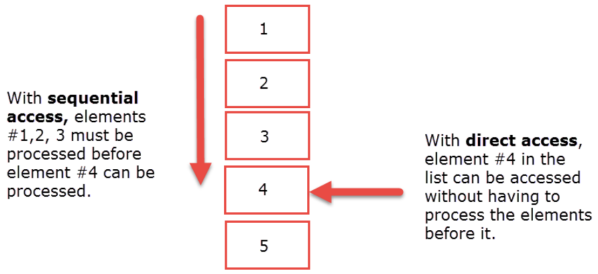
Direct Access

Operations: read n, write n, Reset, Position (seek) to n. n = relative block number

sequential access	implementation for direct access
reset	*cp* = 0;
read next	*read cp*; *cp* = *cp* + 1;
write next	*write cp*; *cp* = *cp* + 1;

Figure 11.3: Direct Access

Sequential vs Direct access



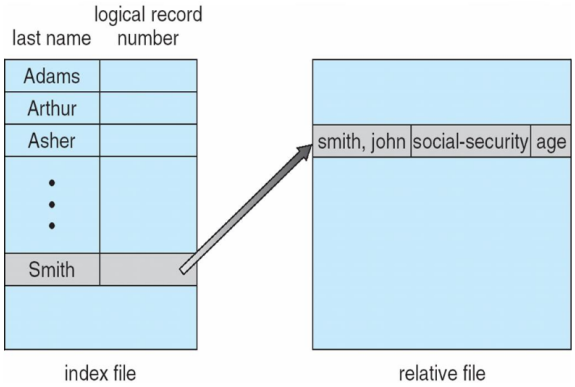
Other Access Methods

Can be other access methods built on top of base methods. Generally involve creation of an index for the file. Keep index in memory for fast determination of location of data to be operated on (consider Universal Produce Code (UPC code) plus record of data about that item).

If the index is too large, create an in-memory index, which an index of a disk index.

IBM indexed sequential-access method (ISAM)

- Small master index, points to disk blocks of secondary index
- File kept sorted on a defined key
- All done by the OS
- A similar engine, MyISAM, is a way of indexing MySQL DBs



11.3 Memory-Mapped Files

Rather than accessing data files from the HD with every file access, data files can be paged into memory the same as process files, resulting in much faster accesses.

You can see it as "paging" a file = a page table per file, except of course when page-faults occur. This is known as memory-mapping a file.

A file is mapped to an address range within a process's virtual address space, and then paged in as needed using the ordinary demand paging system. Can be used to load 10GB file when only 2GB RAM available!

File writes are made to the memory page frames, and are not immediately written out to disk. This is also why it is important to "close( )" a file when done writing to it.

So that the data can be safely flushed out to disk and so that the memory frames can be freed up for other purposes.

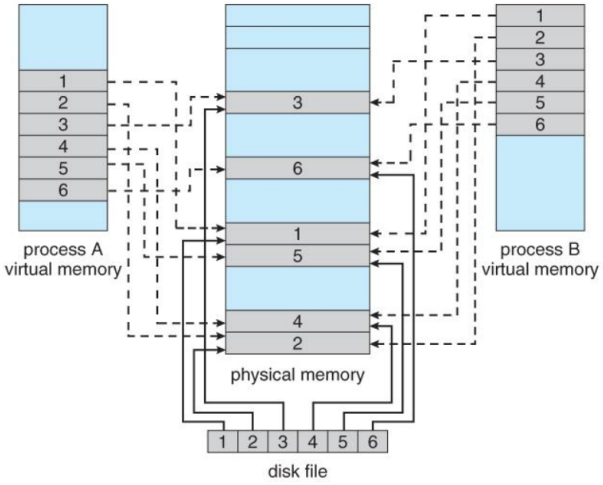


Figure 11.4: Memory-Mapped Files

11.4 Disk and File Systems

From last lecture:

- Disks can be subdivided into partitions
- Disks or partitions can be RAID protected against failure
- Partitions also known as minidisks, slices

A file system or filesystem (often abbreviated to FS or fs) governs file organization and access.
Entity containing file system is known as a volume, a formatted disk or partition is a volume.
Disk or partition can be used raw – without a file system, or formatted with a file system. Each volume containing a file system also tracks that file system's info in device directory or volume table of contents. In addition to general-purpose file systems there are many special-purpose file systems, frequently all within the same operating system or computer.

11.4.1 Types of File Systems

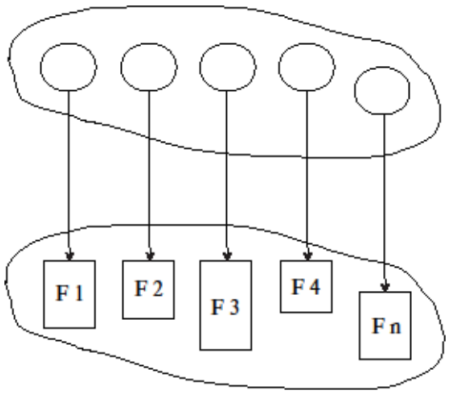
We mostly talk of general-purpose file systems, but systems frequently have many file systems, some general- and some special- purpose.
Consider Solaris has:

- tmpfs – memory-based volatile FS for fast, temporary I/O
- objfs – interface into kernel memory to get kernel symbols for debugging
- ctfs – contract file system for managing daemons
- lofs – loopback file system allows one FS to be accessed in place of another
- procfs – kernel interface to process structures
- ufs, zfs – general purpose file systems

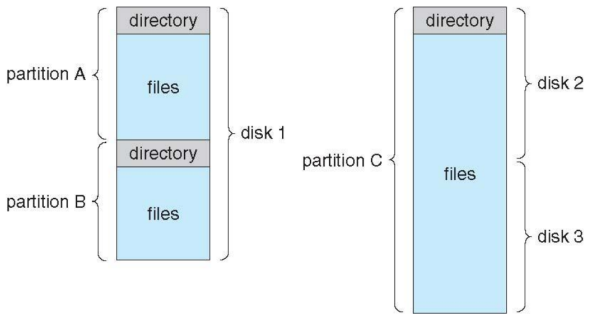
And also Linux has multiple filesystem.

11.5 Directory Structure

A collection of nodes containing information about all files



Both the directory structure and the files reside on disk. Directory information are usually on top of a partition.
The operation performed on Directory are a lot: Search for a file, Create a file, Delete a file, List contents of a directory, Rename a file, Traverse the file system to sub-directories or parent directories.



11.5.1 Directory Organization

The directory is organized logically for performance, usability and access control Particularly:

- Efficiency – locating a file quickly
- Naming – convenient to users: Two users can have same name for different files; The same file can have several different names
- Grouping – logical grouping of files by properties, (e.g., all Java programs, all games ...)

11.5.2 Single-Level Directory

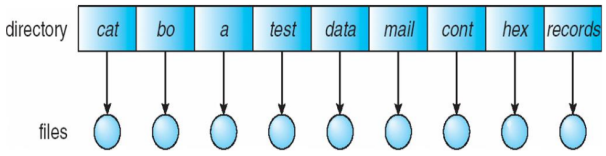


Figure 11.5: A single directory for all users

Lot of problems: Naming problem and Grouping problem are some example.

11.5.3 Two-Level Directory

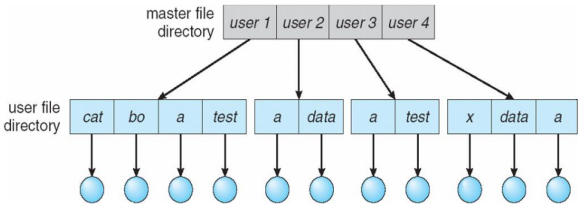
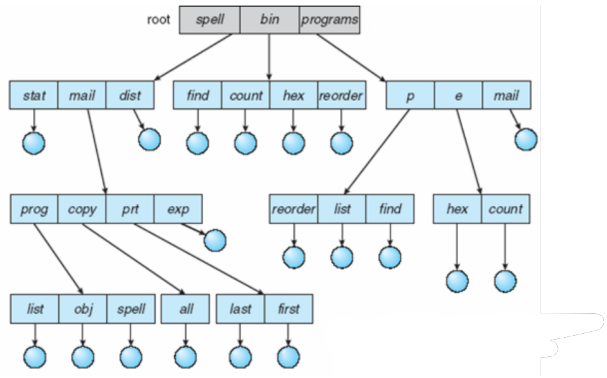


Figure 11.6: Separate directory for each user

Features: Long Path name (bad), Can have the same file name for different user (neutral), Efficient searching (good), No grouping capability (bad)

11.5.4 Tree-Structured Directories

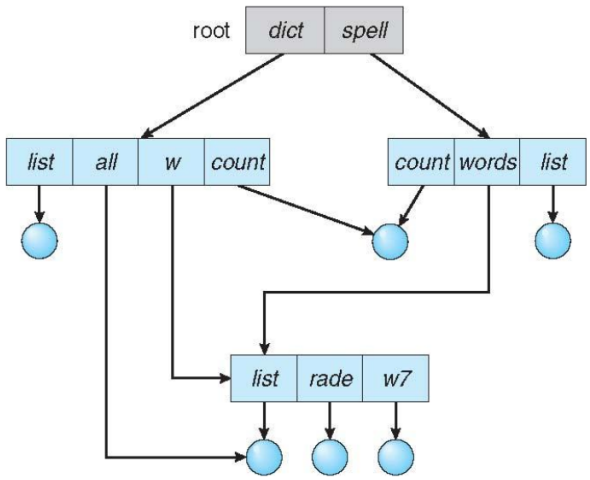


Pro	Cons
General	Inefficient
Scalable	No file sharing
Easy to Search	File duplication in multiple directories

Figure 11.7: RAID schemes

11.5.5 Acyclic-Graph Directories

Have shared subdirectories and files with different names (aliases), but may refer to the same data, thus sharing is possible.



When deleting a directory, you get a dangling pointer that it points to memory which is no longer allocated, and your dereference of it constitutes undefined behaviour.

Solutions:

- Backpointers, so we can delete all pointers to the deleted folder, variable size records a problem
- Backpointers using a daisy chain organization
- Entry-hold-count solution

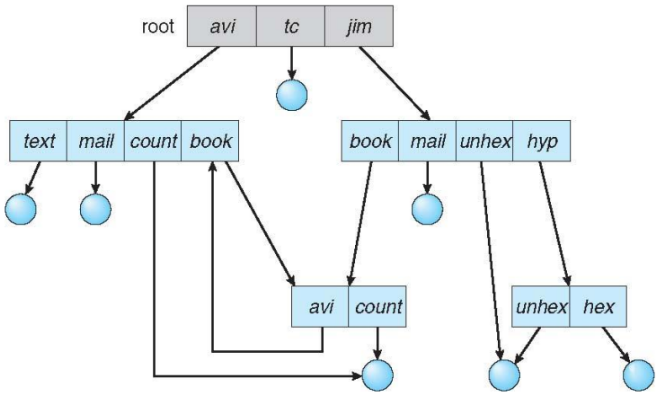
New directory entry type:

Link – another name (pointer) to an existing file

Resolve the link – follow pointer to locate the file

11.5.6 General Graph Directory

How do we guarantee no cycles? Allow only links to files not sub-directories; **Garbage collection**; Every time a new link is added use a cycle detection algorithm to determine whether it is OK.



11.6 Protection

The problem: You don't want others modifying, reading or moving your data/programs.

File owner/creator should be able to control: What can be done/seen, by whom. Type of access: - Read - Write - Execute - Append - Delete - List.

All the OS provide this basic option.

Chapter 12

File System Implementation

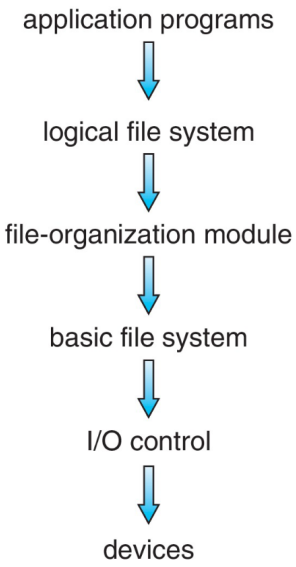
12.1 File-System Structure

File structure: Logical storage unit and collection of related information.
File system provides:

- user interface to storage, mapping logical to physical
- efficient and convenient access to disk by allowing data to be stored, located retrieved easily
- Interface to HW: Disk provides in-place rewrite and random access - I/O transfers performed in blocks or sectors (usually 512 bytes)

File control block (**FCB**) - storage structure consisting of information about a file.
Device driver controls the physical device - Many activities: a File System can be organized into layers.

12.2 Layered File System



12.2.1 Device drivers

Device drivers manage I/O devices at the I/O control layer.
Given commands like: read drive1, cylinder 72, track 2, sector 10, into memory location 1060.
Outputs low-level hardware specific commands to hardware controller

12.2.2 Basic file system

Basic file system given command like "retrieve block 123" translates to device driver.
Also manages memory buffers and caches (allocation, freeing, replacement):

- Buffers hold data in transit
- Caches hold frequently used data

12.2.3 File organization module

File organization module understands files, logical address, and physical blocks.
Translates logical block # to physical block #.
Manages free space, disk allocation.

12.2.4 Logical file system

Logical file system manages metadata information:

- Translates file name into file number, file handle, location by maintaining file control blocks (inodes in UNIX)
- Directory management
- Protection

Layering useful for reducing complexity and redundancy, but adds overhead and can decrease performance.
Logical layers can be implemented by any coding method according to OS designer.
The logical file system is directly called by **applications** e.g., C open()

12.3 File System Types

Many file systems, sometimes many within an operating system, each with its own format:

- CD-ROM is ISO 9660;
- Unix has UFS, FFS;
- Linux has more than 130 types, with extended file system ext3 and ext4 leading;
- Windows has FAT, FAT32, NTFS as well as floppy, CD, DVD Blu-ray.

New ones still arriving – ZFS, GoogleFS, Oracle ASM, FUSE

12.4 From API to implementation

12.4.1 File-System Operations

We have system calls at the API level: Open, close, read, write, seek...but how do we implement their functions? On-disk and in-memory structures.

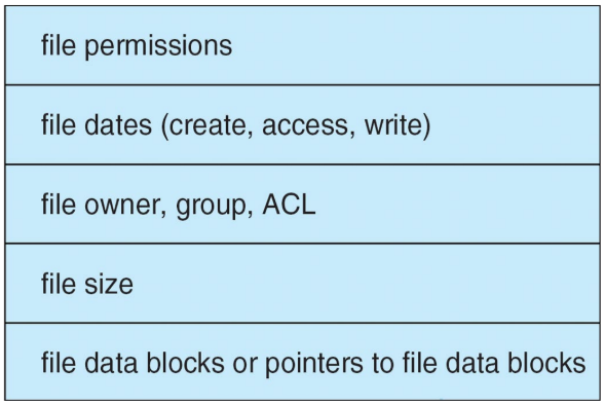
Boot control block (or **MBR**) contains info needed by system to boot OS from that volume. Needed if volume contains OS, usually first block of volume.

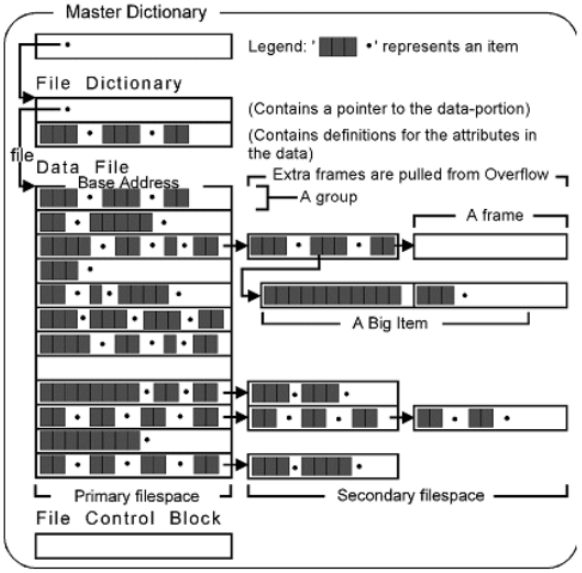
Volume control block (superblock, master file table) contains volume details: Total # of blocks, # of free blocks, block size, free block pointers or array.

Directory structure organizes the files: Names and inode numbers, master file table.

12.4.2 File Control Block - FCB

OS maintains FCB per file, which contains many details about the file: Typically, inode number (if linux), permissions, size, dates.





12.4.3 The Linux iNode

An inode (aka index node) is a data structure used by Unix/Linux in order to describe an object within a filesystem. Such an object could be a file or a directory.

Every inode stores pointers to the disk block's locations of the object's data and metadata.

Overall, the metadata contained in an inode is:

- file type (regular file/directory/symbolic link/block special file/character special file/etc),
- permissions,
- owner/group id,
- size,
- last accessed/modified time,
- change time
- number of hard links

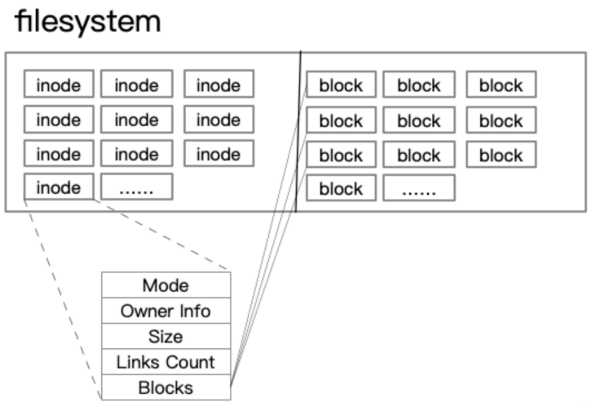


Figure 12.1: The Linux iNode

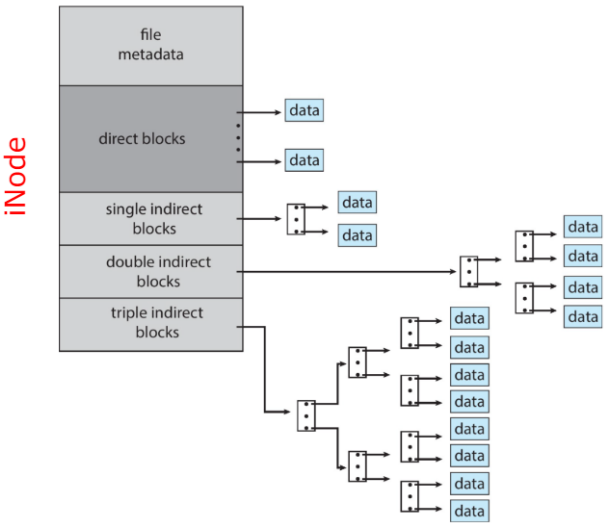


Figure 12.2: UNIX UFS - 4K bytes per block, 32-bit addresses

12.4.4 In-Memory File System Structures

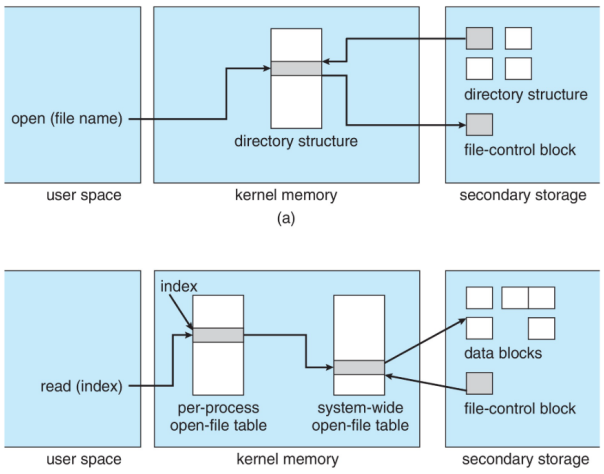
Mount table - storing file system mounts, mount points, file system types.

System-wide open-file table - contains a copy of the FCB of each file and other info.

Per-process open-file table - contains pointers to appropriate entries in systemwide open-file table as well as other info.

Items of the table are called:

- fd file descriptors in UNIX/Linux
- fh file handler in Windows
- same naming as C functions

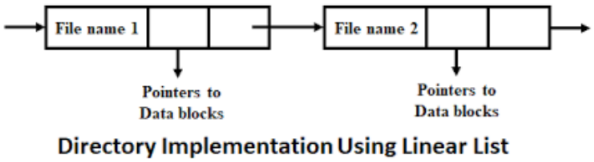


12.5 Directory Implementation

12.5.1 Linear

Linear list of file names with pointer to the data blocks.

- Simple to program
- Time-consuming to execute, Linear search time, could keep ordered alphabetically via linked list or use B+ tree

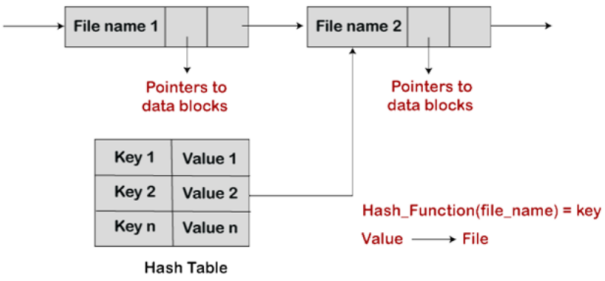


Directory Implementation Using Linear List

12.5.2 HashTable

Hash Table – linear list with hash data structure:

- Decreases directory search time
- Collisions – situations where two file names hash to the same location
- Only good if entries are fixed size, or use chained-overflow method



12.6 Allocation Method

An allocation method refers to how disk blocks are allocated for files:

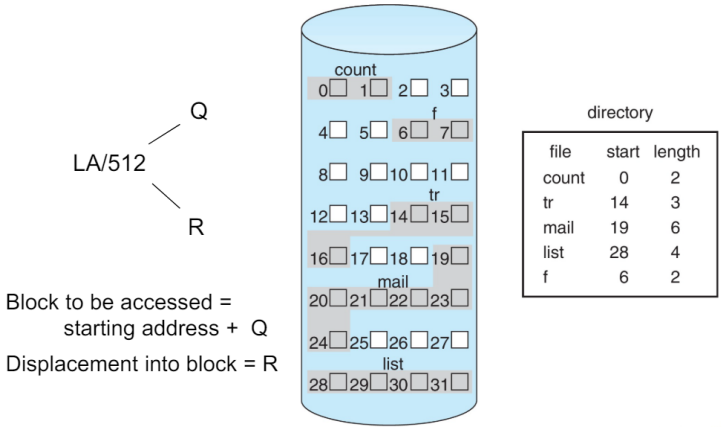
- Contiguous
- Linked
- File Allocation Table (FAT)

12.6.1 Contiguous Allocation Method

An allocation method refers to how disk blocks are allocated for files: each file occupies set of contiguous blocks.

- Best performance in most cases
- Simple – only starting location (block #) and length (number of blocks) are required
- Problems include:
  - Finding space on the disk for a file,
  - Knowing file size,
  - External fragmentation in the HD, need for compaction off-line (downtime) or on-line. This is also called defrag.

Mapping from logical to physical (block size = 512 bytes).



12.6.2 Linked Allocation

Each file is a linked list of blocks

File ends at nil pointer

No external fragmentation

Each block contains pointer to next block

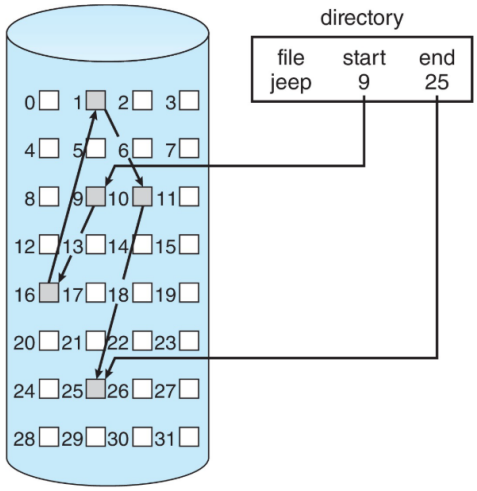
No compaction, external fragmentation

Free space management system called when new block needed

Improve efficiency by clustering blocks into groups but increases internal fragmentation

Big problem: Locating a block takes many I/Os and disk seeks.

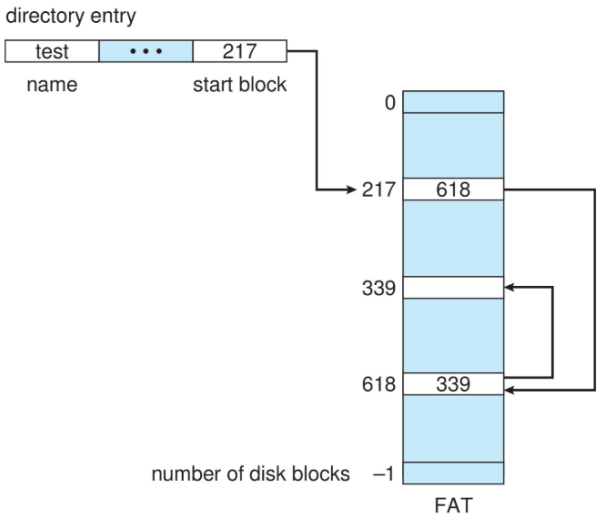
Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk



Block to be accessed is the Q^{th} block in the linked chain of blocks representing the file. Displacement into block = R + 1.

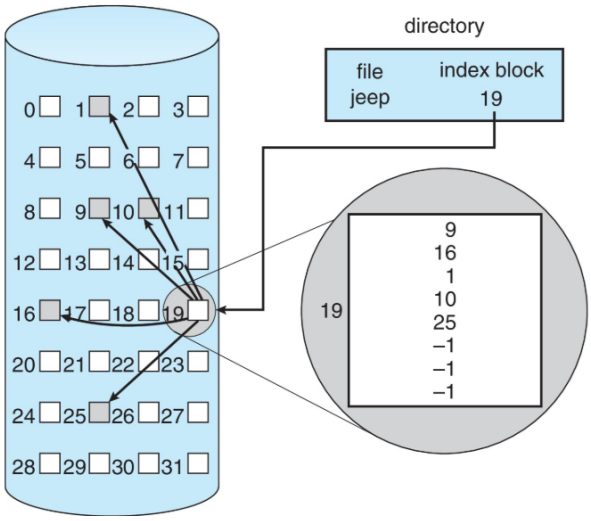
12.6.3 FAT Allocation Method

Beginning of volume has table, indexed by block number. Much like a linked list, but faster on disk and cacheable. New block allocation simple.



12.6.4 Indexed Allocation Method

Each file has its own index block(s) of pointers to its data blocks.



12.6.5 Performance

Best method depends on file access type:

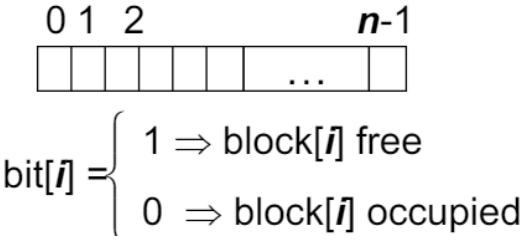
- Contiguous great for sequential and random access
- Linked good for sequential, not random

Declare access type at creation: select either contiguous or linked.
Indexed more complex: single block access could requires index block read then data block read.

12.7 Free-Space Management

File system maintains **free-space list** to track available blocks. Similar to the free pages list for vm management.

Bit vector or **bit map** (n blocks)



Block number calculation:

(number of bits per word) * (number of 0-value words) + offset of first 1 bit

Easy, but Bit map requires extra space.

Example:

$$\text{block size} = 4\text{KB} = 2^{12} \text{ bytes}$$

$$\text{disk size} = 2^{40} \text{ bytes (1 TB)}$$

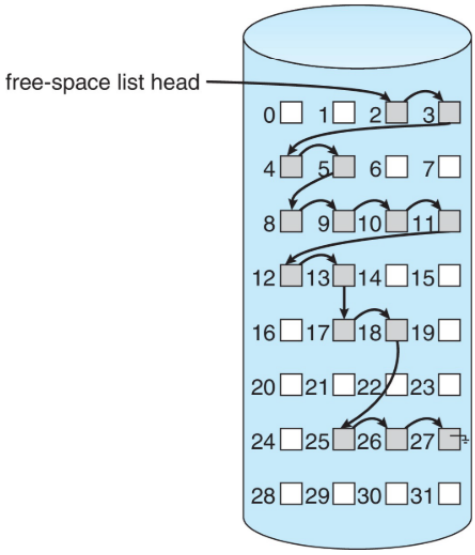
$$\text{n} = 2^{40}/2^{12} = 2^{28} \text{ bits (or 32MB)}$$

Advantage: Easy to get space for contiguous files by checking if enough adjacent blocks are free

12.7.1 Linked Free Space List on Disk

Linked list (free list):

- **BAD**: Cannot get contiguous space easily
- **GOOD**: No waste. Linked Free Space List on Disk of space



No need to traverse the entire list (if # free blocks recorded)

12.7.2 Linked Free Space List

Grouping:

- Modify linked list to store address of next n-1 free blocks in first free block, plus a pointer to next block that contains free-blockpointers (like this one)
- First n-1 blocks will be free
- The n-th will contain pointers to the next n-1 free blocks

Counting

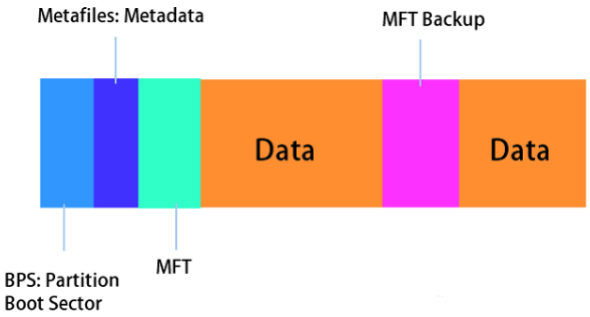
- Because space is frequently contiguously used and freed, with contiguous-allocation allocation, extents, or clustering. Keep address of first free block and count of following free blocks, free space list then has entries containing addresses and counts
- First block of the free list will contain address of next free block + the number of free blocks after this one
- Others will be effectively free blocks

12.8 FS in other OS

12.8.1 Window's File Systems

NTFS, also called New Technology File System, is a proprietary journaling file system developed by Microsoft.

Starting with Windows NT 3.1, it is the default file system of the Windows NT family. It superseded File Allocation Table (FAT) as the preferred file system on Windows and is also supported in Linux and BSD.



FAT, also known as File Allocation Table, is a file system developed for personal computers.

Originally developed in 1977 for use on floppy disks, it was adapted for use on hard disks and other devices. It is often supported for compatibility reasons by current operating systems for personal computers and many mobile devices and embedded systems, allowing the interchange of data between disparate systems.

FAT32, a successor of FAT16, was designed by Microsoft as a new file system version. FAT32 supports an increased number of possible clusters and reuses most of the existing codes.

Figure 12.3: FAT32 File system structure

12.8.2 The Apple File System

In 2017, Apple, Inc., released a new file system to replace its 30-year-old HFS+ file system. HFS+ had been stretched to add many new features, but as usual, this process added complexity, along with lines of code, and made adding more features more difficult. Starting from scratch on a blank page allows a design to start with current technologies and methodologies and provide the exact set of features needed.

Apple File System (APFS) is a good example of such a design. Its goal is to run on all current Apple devices, from the Apple Watch through the iPhone to the Mac computers. Creating a file system that works in watchOS, I/Os, tvOS, and macOS is certainly a challenge. APFS is feature-rich, including 64-bit pointers, clones for files and directories, snapshots, space sharing, fast directory sizing, atomic safe-save primitives, copy-on-write design, encryp- tion (single- and multi-key), and I/O coalescing. It understands NVM as well as HDD storage.

Most of these features we've discussed, but there are a few new concepts worth exploring. **Space sharing** is a ZFS-like feature in which storage is avail- able as one or more large free spaces (**containers**) from which file systems can draw allocations (allowing APFS-formatted volumes to grow and shrink).

Fast directory sizing provides quick used-space calculation and updating. **Atomic safe-save** is a primitive (available via API, not via file-system com- mands) that performs renames of files, bundles of files, and directories as single atomic operations. I/O coalescing is an optimization for NVM devices in which several small writes are gathered together into a large write to optimize write performance.

Apple chose not to implement RAID as part of the new APFS, instead depending on the existing Apple RAID volume mechanism for software RAID. APFS is also compatible with HFS+, allowing easy conversion for existing deployments.

Chapter 13

Other File Systems

13.1 File Sharing

Allows multiple users / systems access to the same files.
Permissions / protection must be implemented and accurate

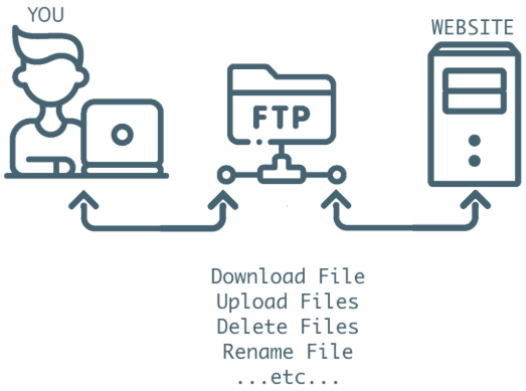
- Most systems provide concepts of owner, group member
- Must have a way to apply these between systems

Owner can change attributes and grant access to other users/groups
Group defines a subset of users who can share access to the file.

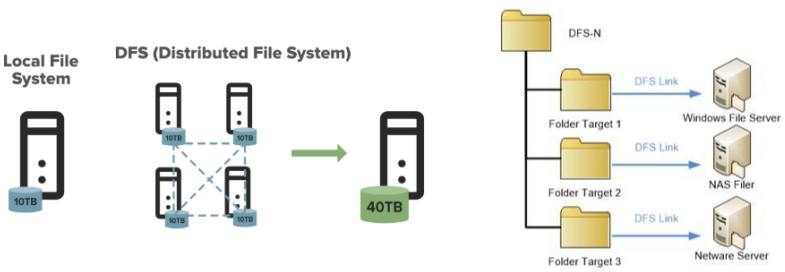
Example for creating a shared folder between users

13.2 Remote File Systems

Sharing files across a network. First method involved manually sharing each file – programs like **ftp**.



Second method uses a **distributed file system** (**DFS**). Remote directories visible from local machine.



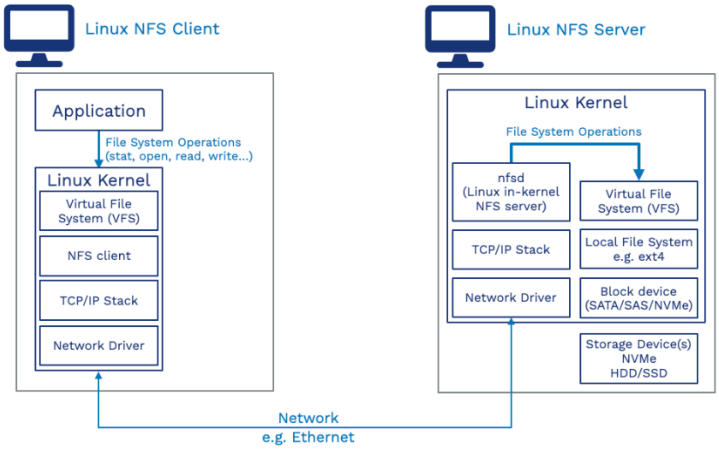
13.3 Client-Server Model

Sharing between:

- a server (providing access to a file system via a network protocol) and
- a client (using the protocol to access the remote file system)

NFS an example:

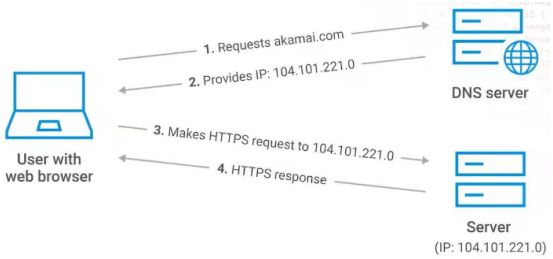
- User auth info on clients and servers must match (UserIDs for example)
- Remote file system mounted, file operations sent on behalf of user across network to server
- Server checks permissions, file handle returned
- Handle used for reads and writes until file closed



13.4 Distributed Information Systems - DIS

Aka **distributed naming services**, provide unified access to info needed for remote computing.

Domain name system (**DNS**) provides host-name-to-network-address translations for the Internet

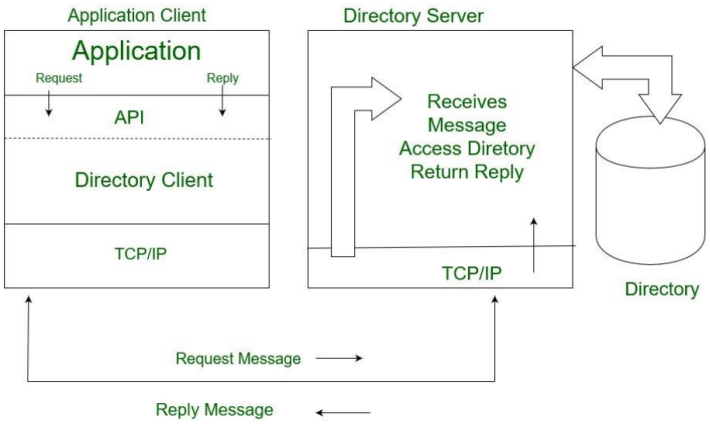


Others like network information service (NIS) provide user-name, password, userID, group information.

Microsoft's common Internet file system (CIFS) network info used with user auth to create network logins that server uses to allow to deny access:

- Active directory distributed naming service
- Kerberos-derived network authentication protocol

Industry moving toward lightweight directory-access protocol (LDAP) as secure distributed naming mechanism



13.5 Consistency Semantics

Important criteria for evaluating file sharing-file systems. Specify how multiple users are to access shared file simultaneously.

Remember reader-writer?

When modifications of data will be observed by other users

Directly related to process synchronization algorithms, but atomicity across a network has high overhead (see Andrew File System)

The series of accesses between file open and closed called **file session**.
UNIX semantics:

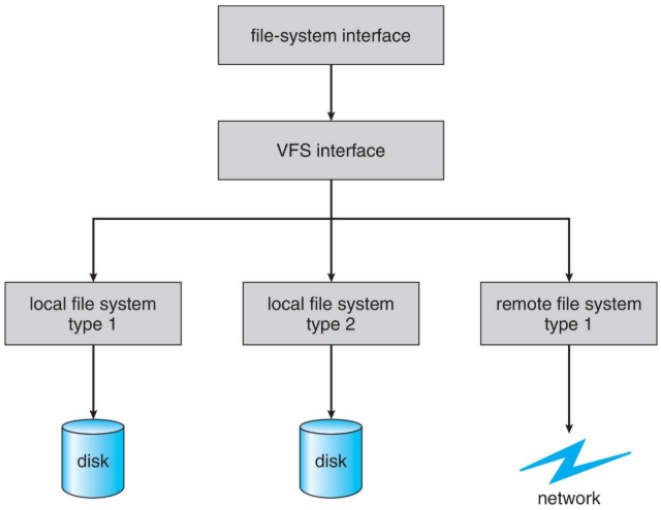
- Writes to open file immediately visible to others with file open
- One mode of sharing allows users to share pointer to current I/O location in file
- Single physical image, accessed exclusively, contention causes process delays

13.6 Virtual File Systems

Virtual File Systems (**VFS**) on Unix provide an object-oriented way of implementing file systems.
VFS allows the same system call interface (the API) to be used for different types of file systems:

- Separates file-system generic operations from implementation details
- Implementation can be one of many file systems types, or network file system, implements vnodes which hold inodes or network file details
- Then dispatches operation to appropriate file system implementation routines

The API is to the VFS interface, rather than any specific type of file system



13.6.1 Virtual File System Implementation

For example, Linux has four object types:

$$inode, file, superblock, dentry$$

VFS defines set of operations on the objects that must be implemented. Every object has a pointer to a function table:

Function table has addresses of routines to implement that function on that object.

For example:

- int open(. . .)— Open a file
- int close(. . .)— Close an already-open file
- ssize t read(. . .)— Read from a file
- ssize t write(. . .)— Write to a file
- int mmap(. . .)— Memory-map a file

Chapter 14

Virtualization and Virtual Machines

14.1 Virtualization

Remember what we just said about virtual FSs?

- an abstract layer on top of one or more "real" more concrete FSs.
- allows client applications to access different types of concrete file systems in a uniform way.
- Example: access local and network storage devices transparently without the client application noticing the difference.
- It can be used to bridge the differences in Windows, classic Mac OS/macOS and Unix filesystems, so that applications can access files on local file systems of those types without having to know what type of file system they are accessing.

This is one example of a more general process that is called virtualization.

More in general, Virtualization aims at putting a layer between the implementation of "something" and its interface / client application / service.

For FSs, the user does not care about where a file is or how a filesystem is structured, it just uses the APIs.

This is so nice that it has been extended for "virtualizing" many other things: and particularly relevant for this course, we can virtualize OSs or Hardware or other devices

14.2 Virtual machines

Fundamental idea – abstract hardware of a single computer into several different execution environments.

Similar to layered approach, but layer creates virtual system (virtual machine, or VM) on which operation systems or applications can run.

Several components:

- **Host** – underlying hardware system
- **Virtual machine manager** (**VMM**) or **hypervisor** – creates and runs virtual machines by providing correct interfacing
- **Guest** – process provided with virtual copy of the host. Usually an operating system

Single physical machine can run multiple operating systems concurrently, each in its own virtual machine.

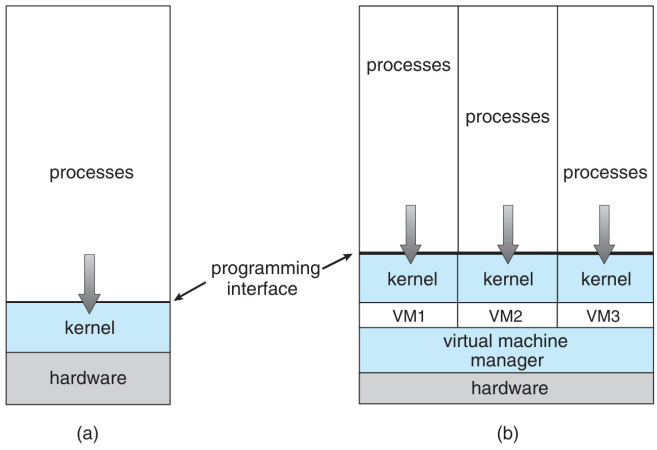
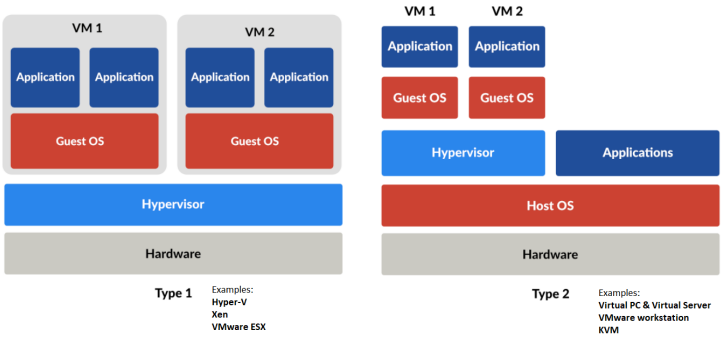


Figure 14.1: (a) Nonvirtual machine. (b) Virtual machine.

14.2.1 Implementation of VMMs

Vary greatly, with options including:

- **Type 0 hypervisors** - Hardware-based solutions that provide support for virtual machine creation and management via firmware. They come with the SoC or motherboard. (IBM LPARs and Oracle LDOMs are examples)
- **Type 1 hypervisors** - Operating-system-like software built to provide virtualization. (ex. VMware ESX, Joyent SmartOS, and Citrix XenServer).
  - Also includes general-purpose operating systems that provide standard functions as well as VMM functions. Including Microsoft Windows Server with HyperV and RedHat Linux with KVM
- **Type 2 hypervisors** - Applications that run on standard operating systems but provide VMM features to guest operating systems. (VMware / Oracle Virtualbox)



14.2.2 Implementation of VMMs

Other variations include:

- Paravirtualization - Technique in which the guest operating system is modified to work in cooperation with the VMM to optimize performance
- Emulators – Allow applications written for one hardware environment to run on a very different hardware environment, such as a different type of CPU. E.g., IDEs for assembly coding
- Application containment - Not virtualization at all but rather provides virtualization-like features by segregating applications from the operating system, making them more secure, manageable. Baseline for sandboxing

14.2.3 Benefits and Features

Host system protected from VMs, VMs protected from each other:

- A virus less likely to spread
- Sharing is provided though via shared file system volume, network communication

Freeze, **suspend**, running VM:

- Then can move or copy somewhere else and resume
- Snapshot of a given state, able to restore back to that state, some VMMs allow multiple snapshots per VM
- Clone by creating copy and running both original and copy

Great for OS **research**, better system development efficiency.

Run **multiple**, different **OSes** on a single machine, **consolidation**, app dev.

Templating – create an OS + application VM, provide it to customers, use it to create multiple instances of that combination.

Live migration – move a running VM from one host to another! No interruption of user access.

14.2.4 Running mode

Dual mode CPU means guest executes in user mode:

Kernel runs in kernel mode

Not safe to let guest kernel run in kernel mode too

So VM needs two modes – virtual user mode and virtual kernel mode. Both of which run in real user mode

Actions in guest that usually cause switch to kernel mode must cause switch to virtual kernel mode

14.2.5 Trap-and-Emulate

How does switch from virtual user mode to virtual kernel mode occur?

- Attempting a privileged instruction in user mode causes an error → trap
- VMM gains control, analyzes error, executes operation as attempted by guest
- Returns control to guest in user mode
- Known as trap-and-emulate
- Most virtualization products use this at least in part

User mode code in guest runs at same speed as if not a guest.

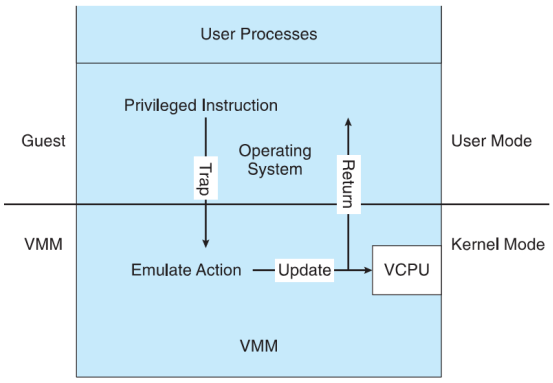


Figure 14.2: Trap-and-Emulate Virtualization Implementation

14.2.6 What about Containers?

The last decade has seen a wide spreading of containers:

- Which are a way of putting software into "boxes"
- makes it very easy to package and ship programs
- And bring your code here and there without worrying too much

So... Containers = Virtual Machines?

Short answer: no, but it is a way to virtualize

Main difference: containers use a shared OS.

Instead of virtualizing hardware as virtual machines, containers rest on top of a single Linux instance.

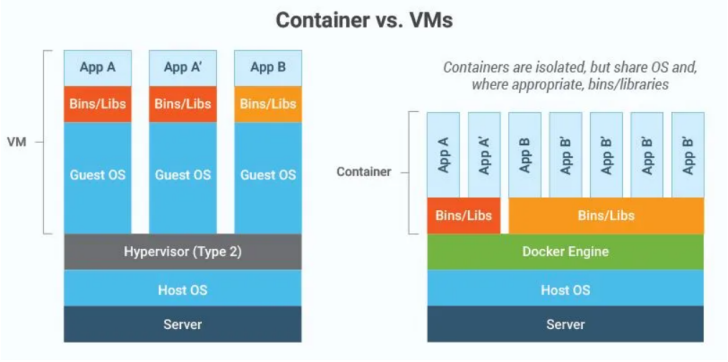


Figure 14.3: Containers vs. VMs

14.3 Types of VMS and implementations

Many variations as well as HW details. Assume VMMs take advantage of HW features, HW features can simplify implementation, improve performance.

Whatever the type, a VM has a lifecycle:

1. Created by VMM
2. Resources assigned to it (number of cores, amount of memory, networking details, storage details). In type 0 hypervisor, resources usually dedicated, other types dedicate or share resources, or a mix.
3. When no longer needed, VM can be deleted, freeing resources

Steps simpler, faster than with a physical machine install. Can lead to **virtual machine sprawl** with lots of VMs, history and state difficult to track.

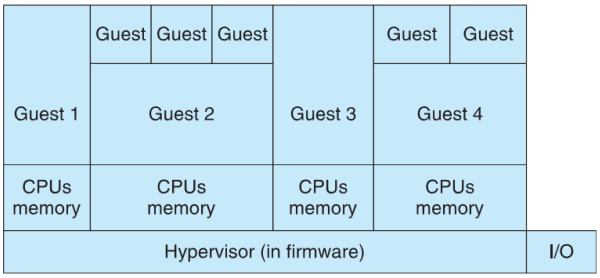
14.3.1 Type 0 Hypervisor

Old idea, under many names by HW manufacturers: "partitions", "domains", A HW feature implemented by firmware, OS need nothing special, VMM is in firmware, Smaller feature set than other types, Each guest has dedicated HW.

I/O a challenge as difficult to have enough devices, controllers to dedicate to each guest.

Sometimes VMM implements a control partition running daemons that other guests communicate with for shared I/O.

Can provide virtualization-within-virtualization (guest itself can be a VMM with guests. Other types have difficulty doing this.



14.3.2 Type 1 Hypervisor

Commonly found in company datacenters. In a sense becoming “datacenter operating systems”:

- Datacenter managers control and manage OSes in new, sophisticated ways by controlling the Type 1 hypervisor
- Move guests between systems to balance performance
- Snapshots and cloning

Another variation is a general purpose OS that also provides VMM functionality:

RedHat Enterprise Linux with KVM, Windows with Hyper-V, Oracle Solaris

Perform normal duties as well as VMM duties

Typically less feature rich than dedicated Type 1 hypervisors

In many ways, treat guests OSes as just another process. Albeit with special handling when guest tries to execute special instructions.

14.3.3 Type 2 Hypervisor

Less interesting from an OS perspective.

- Very little OS involvement in virtualization
- VMM is simply another process, run and managed by host, even the host doesn't know they are a VMM running guests
- Tend to have poorer overall performance because can't take advantage of some HW features
- But also a benefit because require no changes to host OS

Student could have Type 2 hypervisor on native host, run multiple guests, all on standard host OS such as Windows, Linux, MacOS.

That's what we are using for exercises

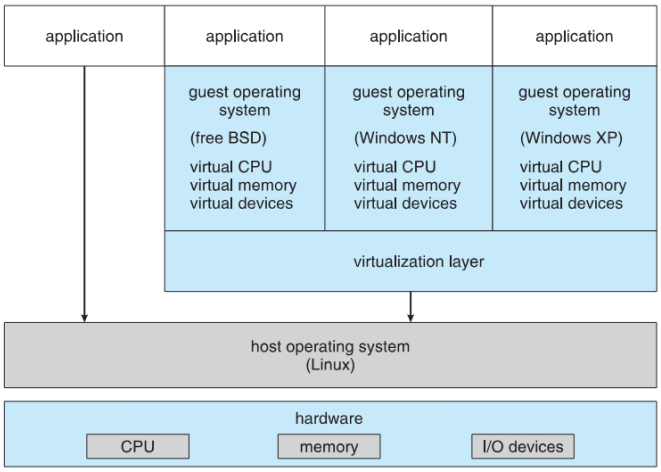


Figure 14.4: VMware Workstation Architecture

14.3.4 Programming Environment Virtualization

Programming language is designed to run within custom-built virtualized environment. For example Oracle Java has many features that depend on running in **Java Virtual Machine** (JVM).

In this case virtualization is defined as providing APIs that define a set of features made available to a language and programs written in that language to provide an improved execution environment.

- JVM compiled to run on many systems (including some smart phones even)
- Programs written in Java run in the JVM no matter the underlying system
- Similar to interpreted languages

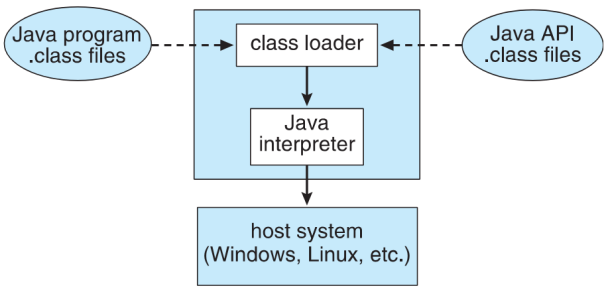


Figure 14.5: The Java Virtual Machine

Oracle containers / zones for example create virtual layer between OS and apps. Only one kernel running – host OS. OS and devices are virtualized, providing resources within zone with impression that they are only processes on system. Each zone has its own applications; networking stack, addresses, and ports; user accounts, etc. CPU and memory resources divided between zones. Zone can have its own scheduler to use those resources.

14.4 Virtualization Issues

Now look at operating system aspects of virtualization. CPU scheduling, memory management, I/O, storage, and unique VM migration feature.

How do VMMs schedule CPU use when guests believe they have dedicated CPUs?

How can memory management work when many guests require large amounts of memory?

14.5 CPU scheduling

Even single-CPU systems act like multiprocessor ones when virtualized, one or more virtual CPUs per guest.

Generally VMM has one or more physical CPUs and number of threads to run on them. Guests configured with certain number of VCPUs.

When enough CPUs for all guests -> VMM can allocate dedicated CPUs, each guest much like native operating system managing its CPUs. Usually not enough CPUs -> CPU overcommitment.

VMM can use standard scheduling algorithms to put threads on CPUs.

Cycle stealing by VMM and oversubscription of CPUs means guests don't get CPU cycles they expect. Some VMMs provide application to run in each guest to fix time-ofday and provide other integration features

If you cant have dedicated CPU(s) for your guest OS, don't run a realtime guest OS!

14.6 I/O

Easier for VMMs to integrate with guests because I/O has lots of variation. Already somewhat segregated / flexible via device drivers, VMM can provide new devices and device drivers.

But overall I/O is complicated for VMMs:

- Many short paths for I/O in standard OSes for improved performance
- Less hypervisor needs to do for I/O for guests, the better
- Possibilities include direct device access, DMA pass-through, direct interrupt delivery. Again, HW support needed for these

Networking also complex as VMM and guests all need network access. VMM can bridge guest to network (allowing direct access), and / or provide network address translation (NAT)[1]

14.7 Storage Management

Both boot disk and general data access need be provided by VMM. Need to support potentially dozens of guests per VMM (so standard disk partitioning not sufficient).

Type 1 – storage guest root disks and config information within file system provided by VMM as a disk image. **Type 2** – store as files in file system provided by host OS. Duplicate file → create new guest. Move file to another system → move guest.

Physical-to-virtual (P-to-V) convert native disk blocks into VMM format.

Virtual-to-physical (V-to-P) convert from virtual format to native or disk format.

VMM also needs to provide access to network attached storage (just networking) and other disk images, disk partitions, disks, etc.

[1] NAT address local to machine on which guest is running, VMM provides address translation to guest to hide its address

Chapter 15

Security & Protection

15.1 What's security? The security problem

System secure if resources used and accessed as intended under all circumstances: **Unachievable**.

- **Intruders** (**crackers**) attempt to breach security
- **Threat** is potential security violation
- **Attack** is attempt to breach security, attack can be accidental or malicious.

Easier to protect against accidental than malicious misuse

15.2 Security Violation Categories

- Breach of **confidentiality**

  Unauthorized access to data

- Breach of **integrity**
  - Unauthorized modification/destruction of data
- Breach of **availability**

  System/service is not ready for users

15.3 Example of attacks

- **Ransomware** (breach integrity) - Encrypts data unless money gets paid to the attacker
- **Replay attack** - Re-send a message as is or with message modification
- **Man-in-the-middle** attack - Intruder sits in data flow, masquerading as sender to receiver and vice versa
- **Session hijacking** - Intercept an already-established session to bypass authentication
- **Privilege escalation** - Common attack type with access beyond what a user or resource is supposed to have
- **Trojan Horse** - Exploits mechanisms for allowing programs written by users to be executed by other users, Spyware, pop-up browser windows
- **Trap Door** - Specific user identifier or password that circumvents normal security procedures, could be included in a compiler Malware - Software designed to exploit, disable, or damage computer
- **Spyware** – Program frequently installed with legitimate software to display adds, capture user data

15.4 Security Measure Levels

Impossible to have absolute security, but make cost to perpetrator sufficiently high to deter most intruders.

Security must occur at four levels to be effective:

- Physical - Data centers, servers, connected terminals
- Application - Benign or malicious apps can cause security problems
- Operating System - Protection mechanisms, debugging
- Network - Intercepted communications, interruption, DOS

Security is as weak as the weakest link in the chain

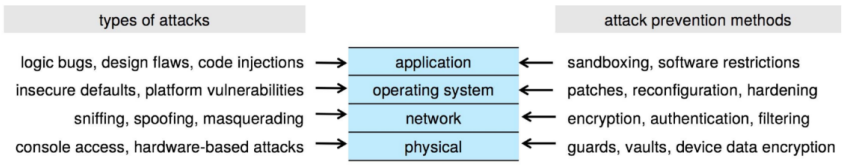


Figure 15.1: Four-layered Model of Security

Below: C Program with Buffer-overflow Condition.

```c
#include <stdio.h>
#define BUFFER SIZE 256

int main(int argc, char *argv[]){
    char buffer[BUFFER SIZE];

    if (argc < 2)
        return -1;
    else {
        strcpy(buffer,argv[1]);
        return 0;
    }
}
```

Code review can help – programmers review each other's code, looking for logic flows, programming flaws.

The arg[1] can overwrite other data.

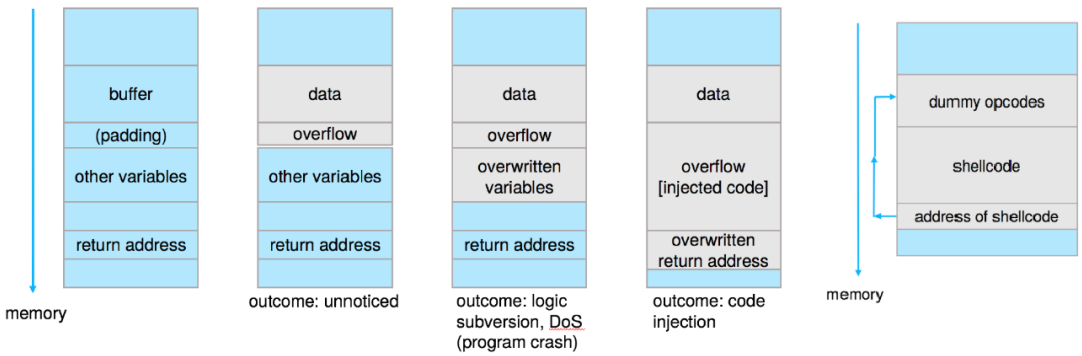


Figure 15.2: Code Injection

The last photo: Frequently use trampoline to code execution to exploit buffer overflow.

15.5 Program Threats

Viruses:

- Code fragment embedded in legitimate program
- Self-replicating, designed to infect other computers
- Very specific to CPU architecture, operating system, applications
- Usually borne via email or as a macro
- Visual Basic Macro to reformat hard drive

Virus dropper inserts virus onto the system. Many categories of viruses, literally many thousands of viruses (File / parasitic, Boot / memory, Macro, Source code, Polymorphic to avoid having a virus signature, Encrypted, Stealth, Tunneling, Multipartite, Armored)

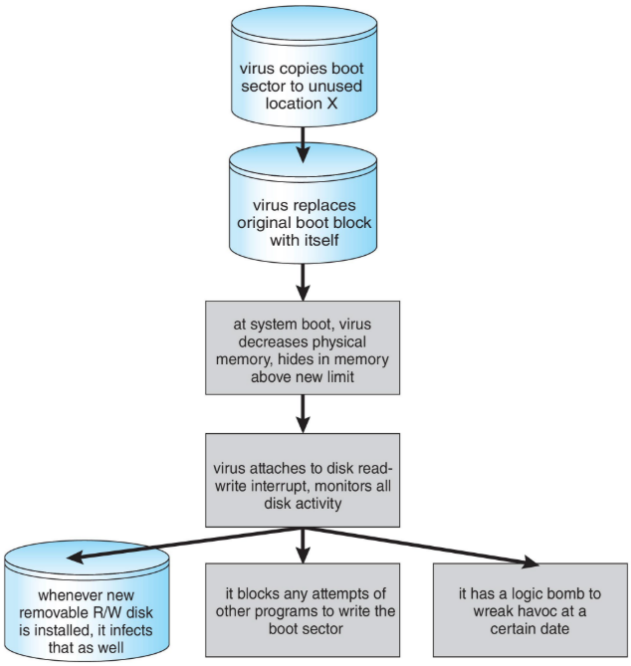


Figure 15.3: A Boot-sector Computer Virus

15.6 The Threat Continues

Attacks still common, still occurring. Attacks moved over time from science experiments to tools of organized crime:

- Targeting specific companies
- Creating botnets to use as tool for spam and DDOS delivery
- Keystroke logger to grab passwords, credit card numbers

Why is Windows the target for most attacks? Most common, Everyone is an administrator, Monoculture considered harmful.

15.7 System and Network Threats

Some systems "open" rather than **secure by default**: Reduce **attack surface**, But harder to use, more knowledge needed to administer.

Network threats harder to detect, prevent:

- Protection systems weaker (not strong)
- More difficult to have a shared secret on which to base access
- No physical limits once system attached to internet, or on network with system attached to internet
- Even determining location of connecting system difficult, IP address is only knowledge

15.7.1 Port scanning

- Automated attempt to connect to a range of ports on one or a range of IP addresses
- Detection of answering service protocol
- Detection of OS and version running on system
- nmap scans all ports in a given IP range for a response
- nessus has a database of protocols and bugs (and exploits) to apply against a system
- Frequently launched from zombie systems, to decrease trace-ability

Automated tool to look for network ports accepting connections, used for good and evil.

15.7.2 Denial of Service

Overload the targeted computer preventing it from doing any useful work. **DoS** can be **DDoS**: **Distributed Denial-of-Service** come from multiple sites at once.

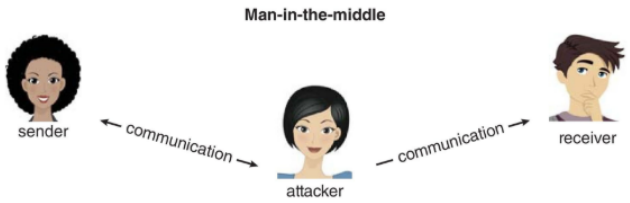
Consider the start of the IP-connection handshake, How many started-connections can the OS handle?

Consider traffic to a web site, How can you tell the difference between being a target and being really popular?

Accidental – CS students writing bad fork() code

Purposeful – extortion, punishment

15.7.3 Man-in-the-middle



15.8 Security mechanisms - Cryptography as a Security Tool

Broadest security tool available, especially against confidentiality threats. Can be used for authentication purposes:

- Internal to a given computer, source and destination of messages can be known and protected: OS creates, manages, protects process IDs, communication ports
- Source and destination of messages on network cannot be trusted without cryptography: Local network – IP address? Consider unauthorized host added. WAN / Internet – how to establish authenticity, Not via IP address.

15.9 Cryptography

Means to constrain potential senders (sources) and / or receivers (destinations) of messages.

Based on secrets (**keys**) and enables:

- Confirmation of source
- Receipt only by certain destination
- Trust relationship between sender and receiver

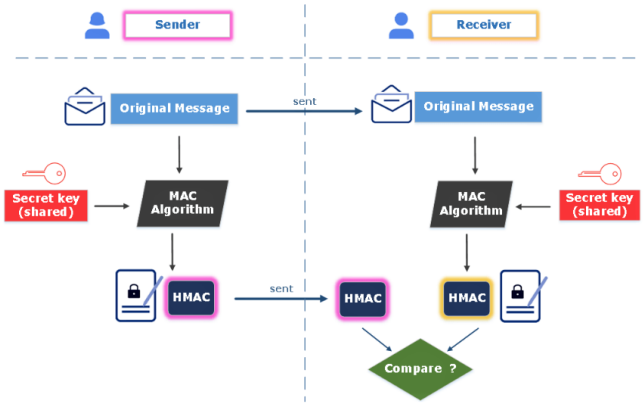
15.9.1 Implementation of Cryptography

Can be done at various layers of ISO Reference Model: SSL at the Transport layer; Network layer is typically IPSec.

Why not just at lowest level? Sometimes need more knowledge than available at low levels, user authentication, e-mail delivery.

15.9.2 Authentication - MAC

Symmetric encryption used in **message-authentication code** (**MAC**) authentication algorithm. Cryptographic checksum generated from message using secret key, can securely authenticate short values .



NOTE: that k is needed to compute both Sk and Vk, so anyone able to compute one can compute the other

15.9.3 Encryption Example - TLS

Insertion of cryptography/authentication at one layer of the ISO network model (the transport layer).

SSL – Secure Socket Layer (also called **TLS**, Transport Layer Security).

Cryptographic protocol that limits two computers to only exchange messages with each other it is very complicated, with many variations. Used between web servers and browsers for secure communication.

The server is verified with a **certificate** assuring client is talking to correct server.

Asymmetric cryptography used to establish a secure **session key** (symmetric encryption) for bulk of communication during session, communication between each computer then uses symmetric key cryptography

15.10 Passwords

Encrypt to avoid having to keep secret. Use algorithm easy to compute but difficult to invert, only encrypted password stored, never decrypted, add "salt" to avoid the same password being encrypted to the same value.

One-time passwords - use a function based on a seed to compute a password, both user and computer. Hardware device / calculator / key fob to generate the password, changes very frequently.

Biometrics - Some physical attribute (fingerprint, hand scan)

Multi-factor authentication - Need two or more factors for authentication: USB, biometric measure, and password.

15.11 Firewalls

A network **firewall** is placed between trusted and untrusted hosts. The firewall limits network access between these two **security domains**.

Can be **tunneled** or **spoofed**.

Tunneling allows disallowed protocol to travel within allowed protocol (i.e., telnet inside of HTTP). Firewall rules typically based on host name or IP address which can be spoofed

Personal firewall is software layer on given host - can monitor / limit traffic to and from the host.

Application proxy firewall understands application protocol and can control them.

System-call firewall monitors all important system calls and apply rules to them.

15.12 Principles of Protection

Guiding principle – **principle of least privilege**.

Programs, users and systems should be given just enough **privileges** to perform their tasks. Properly set **permissions** can limit damage if entity has a bug, gets abused.

Can be static, Or dynamic – domain switching, **privilege escalation**.

Compartmentalization a derivative concept regarding access to data - Process of protecting each individual system component through the use of specific permissions and access restrictions.

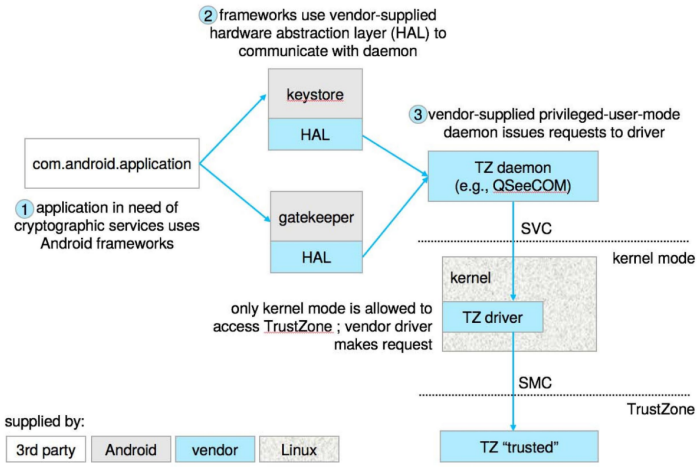
15.12.1 Protection Rings

Components ordered by amount of privilege and protected from each other. For example, the kernel is in one ring and user applications in another.

This privilege separation requires hardware support. Gates used to transfer between levels, for example the syscall Intel instruction also traps and interrupts.

Hypervisors introduced the need for yet another ring.

ARMv7 processors added TrustZone(TZ) ring to protect crypto functions with access via new Secure Monitor Call (SMC) instruction



Only trust app can access the TZ.

15.12.2 Other - Sandboxing

Running process in limited environment. Process then unable to access any resources beyond its allowed set.

Java and .net implement at a virtual machine level, other systems use MAC to implement

15.12.3 Other - System integrity protection SIP

Introduced by Apple in macOS 10.11. Restricts access to system files and resources, even by root. Uses extended file attribs to mark a binary to restrict changes, disable debugging and scrutinizing. Also, only code-signed kernel extensions allowed and configurably only code-signed apps.

System-call filtering: Like a firewall, for system calls Like a firewall, for system calls, also be deeper.

15.12.4 Other - Code signing

Code signing allows a system to trust a program or script by using crypto hash to have the developer sign the executable.

So code as it was compiled by the author, if the code is changed, signature invalid and (some) systems disable execution.

Chapter 16

I/O Hardware

Incredible variety of I/O devices:

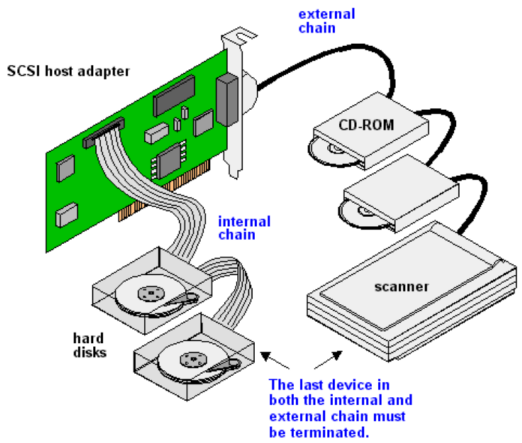
- Storage (disk)
- Transmission (rj45 ethernet)
- Human-interface (mouse, keyboard)

Common concepts – signals from I/O devices interface with computer:

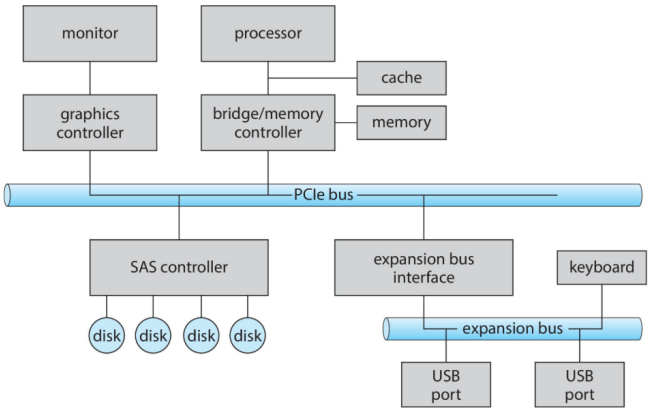
- Port – connection point for device
- Bus - daisy chain or shared direct access, PCI, expansion bus, Small Computer System Interface (SCSI)

16.0.1 SCSI – Daisy Chain

Connected in series, one after the other, transmitted signals go to the first device, then to the second and so on.



Controller (host adapter) – electronics that operate port, bus, device. Sometimes integrated; Sometimes integrated. Contains processor, microcode (drivers), private memory, bus controller, etc.



16.1 I/O strategies

16.1.1 Polling

For each byte of I/O:

1. Read busy bit from status register until 0 (not busy)
2. Host sets read or write bit and if write copies data into data-out register
3. Host sets command-ready bit
4. Controller sets busy bit, executes transfer
5. Controller clears busy bit, error bit, command-ready bit when transfer done

Step 1 is busy-wait cycle to wait for I/O from device. Reasonable if device is fast, inefficient if device slow

16.1.2 Interrupts

Polling can happen in 3 instruction cycles.

Read status, logical-and to extract status bit, branch if not zero. How to be more efficient if zero frequently?

CPU **Interrupt-request line** triggered by I/O device, checked by processor after each instruction.

Interrupt handler receives interrupts: **Maskable** to ignore or delay some interrupts **Interrupt vector** to dispatch interrupt to correct handler: Context switch at start and end, based on priority, some non-maskable (e.g., low battery)

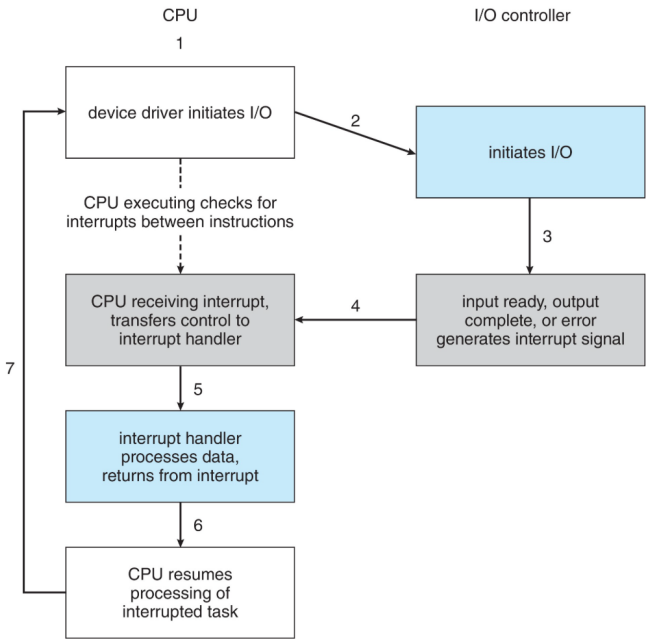
```
while (true)
{
    if(PIND.2 == 0)
        //do something;
}
```

```
main( )
{
    Do your common task
}

whenever PIND.2 is 0 then
   do something
```

Interrupt mechanism also used for:

- exceptions: terminate process, crash system due to hardware error
- Page fault: executes when memory access error

System call executes via trap to trigger kernel to execute request. Multi-CPU systems can process interrupts concurrently, if the OS is designed to handle it



16.1.3 Latency

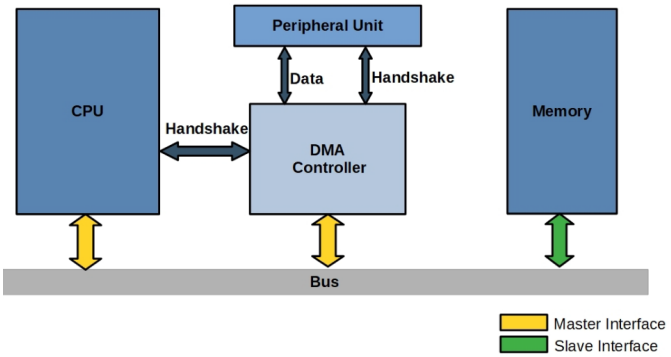
Stressing interrupt management because even single-user systems manage hundreds or interrupts per second and servers hundreds of thousands

16.2 Direct Memory Access

Direct memory access (DMA) is a feature of computer systems that allows certain hardware subsystems to access main memory independently of the central processing unit (CPU).

Used to avoid having the CPU manage interaction with devices and memory, especially for large data movement, requires **DMA** (Direct Memory Access) controller.

Bypasses CPU to transfer data directly between I/O device and memory



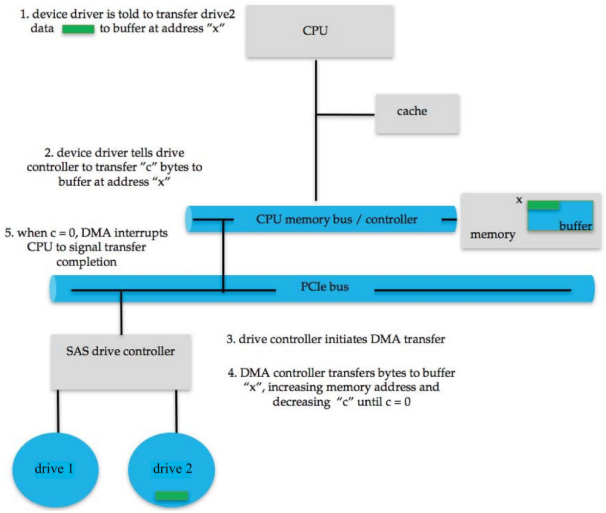
Without DMA, when the CPU is using programmed input/output, it is typically fully occupied for the entire duration of the read or write operation, and is thus unavailable to perform other work.

With DMA, the CPU first initiates the transfer, then it does other operations while the transfer is in progress, and it finally receives an interrupt from the DMA controller when the operation is done.

This feature is useful at any time that the CPU cannot keep up with the rate of data transfer, or when the CPU needs to perform work while waiting for a relatively slow I/O data transfer.

Many hardware systems use DMA, including disk drive controllers, graphics cards, network cards, sound cards.

NOTE: **DMA** can also be used for memory-to-memory operations.



16.3 Application I/O Interface

I/O system calls encapsulate device behaviors in generic APIs

- write() is the same regardless if I am writing on a SSD or HDD
- Not the same if I look at the steps, but common interface

Device-driver layer hides differences among I/O controllers from kernel.
New devices talking already-implemented protocols need no extra work. Each OS has its own I/O subsystem structures and device driver frameworks.
Devices vary in many dimensions:

– Character-stream or block

– Sequential or random-access

– Synchronous or asynchronous (or both)

– Sharable or dedicated

– Speed of operation

– read-write, read only, or write only

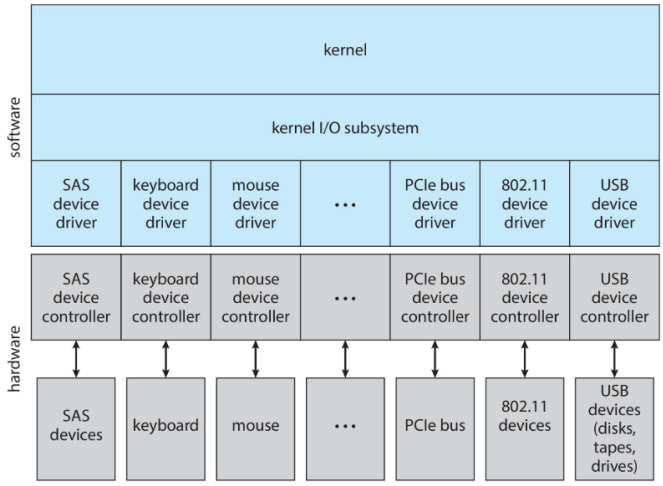


Figure 16.1: Kernel I/O Structure

16.3.1 Characteristics of I/O Devices

I/O devices can be grouped by the OS into:

- Block I/O
- Character I/O (Stream)
- Memory-mapped file access
- Network sockets

Block

Typically disk drives. Commands include read, write, seek. Direct I/O, or file-system access, Memory-mapped file access possible. DMA support.

Character

Include keyboards, mice, serial ports. Commands include get(), put(). Libraries layered on top allow line editing

Network

typically vary a lot and have their own interface. Linux, Unix, Windows and others offer socket interface

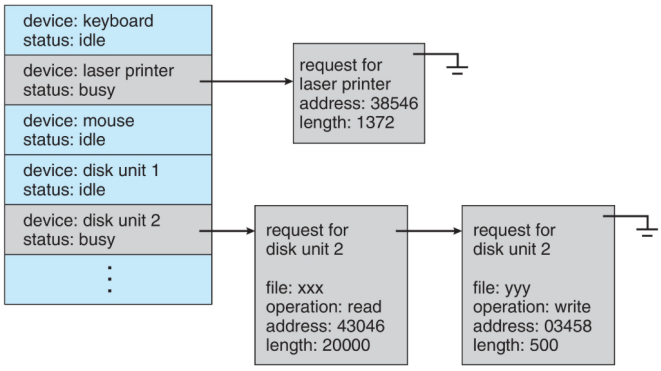


Figure 16.2: Requests and Device-status Table

16.4 I/O strategies

16.4.1 Nonblocking and Asynchronous I/O

- Blocking - process suspended until I/O completed

  Easy to use and understand

  Insufficient for some needs

- Nonblocking - I/O call returns as much as available

  User interface, data copy (buffered I/O)

  Implemented via multi-threading

  select() to find if data ready then read() or write() to transfer

- Asynchronous - process runs while I/O executes

  Difficult to use

  I/O subsystem signals process when I/O completed

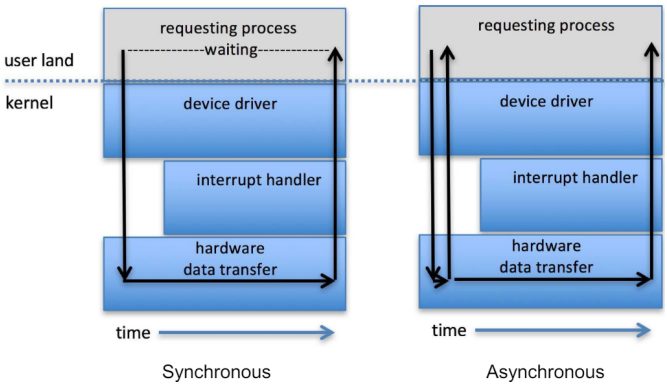


Figure 16.3: Synchronous vs Asynchronous I/O

16.5 Vectored I/O

Vectored I/O allows one system call to perform multiple I/O operations.

For example, Unix readve() accepts a vector of multiple buffers to read into or write from.

Better than multiple individual I/O calls: Decreases context switching and system call overhead, Some versions provide atomicity (Avoid for example worry about multiple threads changing data as reads / writes occurring).

16.6 I/O Protection

User process may accidentally or purposefully attempt to disrupt normal operation via illegal I/O instructions.

Thus, I/O must be performed via system calls. For enhanced control capabilities of the OS.

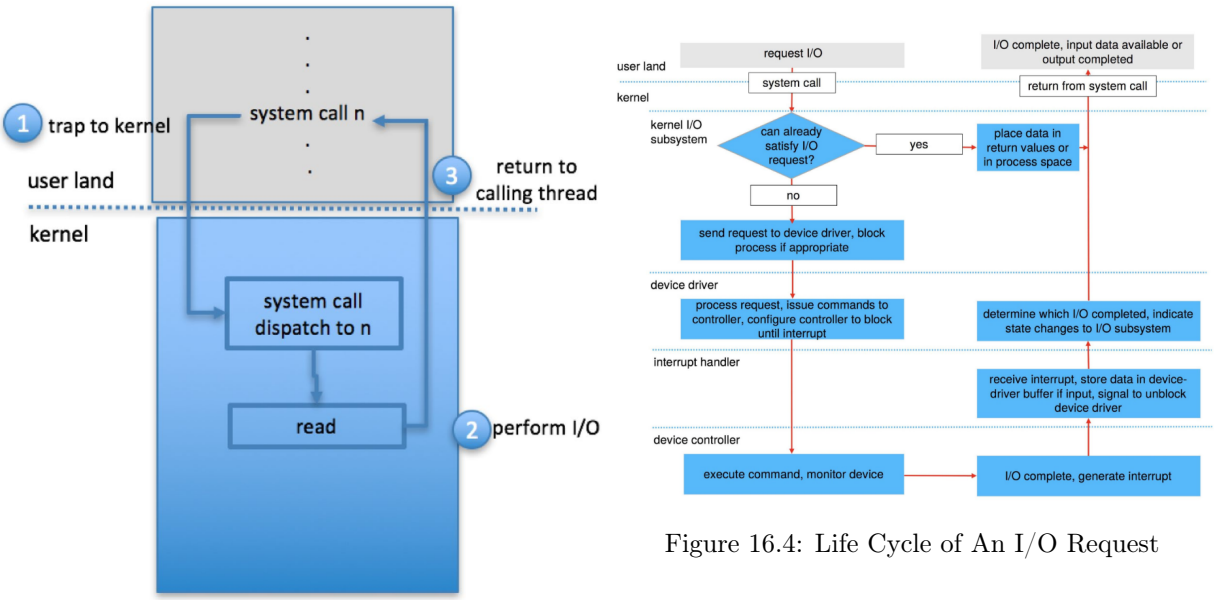


Figure 16.4: Life Cycle of An I/O Request