Operating systems

Salvatore Andaloro

July 4, 2024

Contents

Chapter 1

Introduction

An operating system is a program that acts as an intermediary between the user and the hardware. The main goals of an operating system are:

- Execute user programs and make problem solving easier
- Make the computer easy to use
- Use hardware in an efficient manner

The operating system is just a single component in a computer system; the others are the hardware, the application programs and the users and other machines. The operating system is therefore tasked with managing resources and conflicts and control the execution of programs to prevent and manage errors and improper use of resources.

An operating system is composed of a kernel and other system programs. The **kernel** is the core of the operating system, has complete control over everything that is happening in the system and runs for the whole time the system is turned on. System programs are other programs that are shipped with the operating system.

1.1 Computer startup

The first program that runs on startup is the *bootstrap program*. This program is stored in the ROM or EEPROM and is generally known as **firmware**. It initializes registers, memories and device controllers and loads the kernel into the main memory. The kernel starts **daemons**, i.e. background processes that provide various services to the user. Examples of daemons in Linux systems are `systemd` (daemon that starts other daemons), `syslogd` (logging daemon) and `sshd` (serves SSH connections).

1.2 Interrupts

A general computer architecture is composed of one or more CPUs and device controllers that are connected through a common bus with a shared memory.

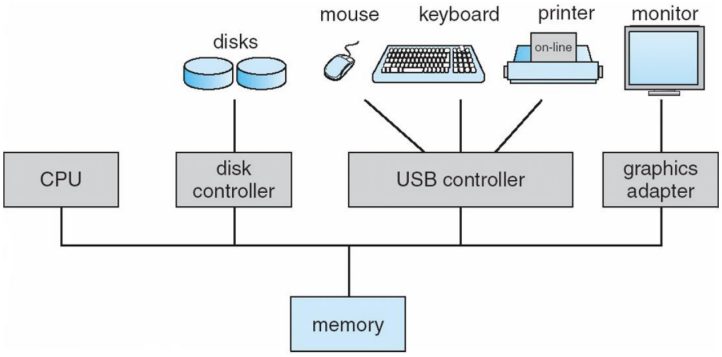


Figure 1.1: General computer architecture

The CPU and IO devices work independently, but they need to communicate with each other. They can achieve this using interrupts. For example, when a device controller has finished some operation (such as loading data into a register), it can inform the CPU that the data is ready to be read by generating an interrupt.

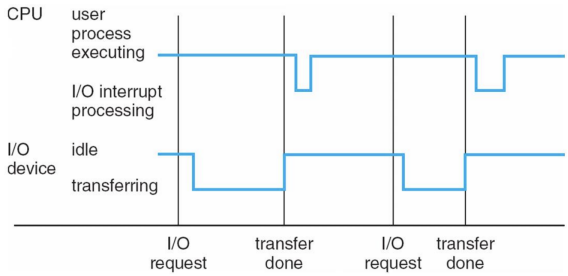


Figure 1.2: Interrupt timing diagram

When an interrupt happens, the OS saves the current program counter (PC) and jumps to the routine that is responsible of handling the interrupt. The list of routines and its addresses are stored in the interrupt table or interrupt vector. The table is initialized at startup and is stored in RAM for fast access.

A trap or exception is a software-generated interrupt caused by an error or an user request.

Depending on the importance of the interrupt, some interrupts must be handled immediately, while others can wait. The first ones are called non-maskable, while the latter are maskable.

While transferring data, the CPU receives an interrupt when every chunk of data has been successfully received. When a lot of data is transferred at once, a lot of interrupts are generated. To reduce the overhead, a feature called DMA (Direct Memory Access) has been introduced. Using this technique, the CPU receives an interrupt only after all the data has been transferred successfully.

1.3 Storage

Storage systems are categorized by speed, cost and volatility. Each storage system has therefore its advantages and disadvantages, therefore there is no "best" storage device.

Therefore a computer has multiple types of storage.

The primary memory is DRAM (dynamic random access memory, based on charged capacitors) or SRAM (static random access memory, based on inverters), which is usually volatile. On the contrary secondary storage is non-volatile, has much bigger capacity but is slower (ex. hard disks, solid-state drives).

Each storage system has a device controller and a device driver. The driver provides an uniform interface between the controller and the kernel.

1.3.1 Caching

Caching is a very common technique for speeding up access to commonly used data. Information is temporarily copied from the slower storage to cache and then every time that information is needed the OS will check cache first. Due to the high cost of cache, it is much smaller than other types of storage, therefore cache management must be properly optimized.

1.4 Modern system architectures

Currently most systems are multiprocessor or multi-core. In these systems, tasks can be allocated in two ways:

- asymmetric processing: each processor/core is assigned a specific task
- symmetric processing: each processor/core performs all tasks

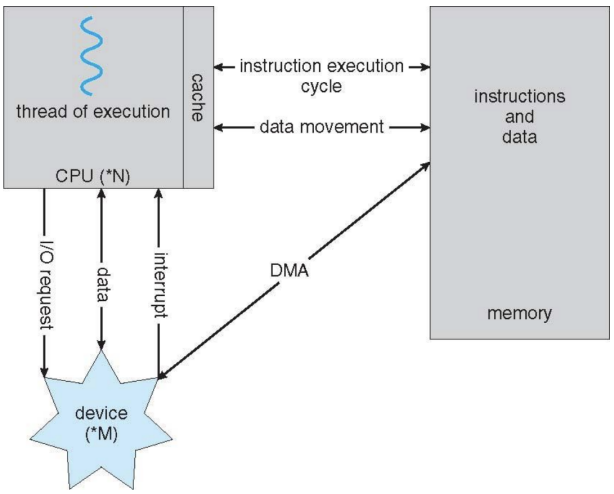


Figure 1.3: A Von-Neumann architecture

1.4.1 Difference between multiprocessor and multi-core

Multiprocessor systems have multiple processors with a single CPU and share the same system bus and sometimes the clock. Multi-core systems have a single processor that contains multiple CPUs. Multi-core systems are more widespread than multiprocessor

systems because they usually consume less power and have on-chip buses, which are faster than standard buses.

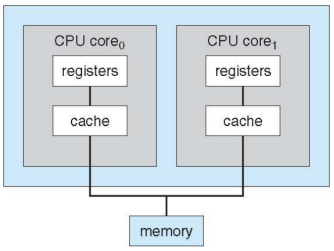


Figure 1.4: A multi-core processor

1.4.2 Clustered systems

Clustered systems are systems composed of multiple machines that usually share the same storage via a storage-area network (SAN). These systems provide a high-availability service that can survive failures of single machines.

- Symmetric clustering: all machines can run tasks and they monitor each other. If a machine fails the others can take over.
- Asymmetric clustering: each machine is assigned to a specific set of tasks. If a machine fails, another machine that was turned on and in "hot-standby mode" takes over.

1.4.3 Multi-programmed systems

The OS can run multiple tasks on the same CPU by using a technique called multiprogramming (batch system): the OS organizes jobs so that the CPU has always one ready to execute. When a job has to wait (for example for I/O) the OS switches to another job. This is called job scheduling. Timesharing (multitasking) is an extension of this technique where the OS switches so frequently among different tasks that the user doesn't notice and can interact with all applications at the same time. This is needed for "window" based systems, where the user can see multiple things are the same time.

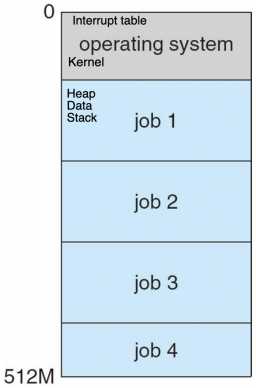


Figure 1.5: Memory layout for multiprogrammed systems

The OS and users share the same hardware, devices and software resources. To protect the system a dual-mode system is established. In such system, jobs can be run in user mode or kernel mode. Some instructions are allowed only for kernel mode systems. To switch between user mode and kernel mode, applications need to perform a system call. Intel processors have four modes of operation, where 0 is fully privileged and 3 is fully restricted.

1.4.4 Process management

A process is a program in execution. The *program* is a passive entity, while the *process* is an active entity. The life of the process is generally managed by the operating system. Single-threaded processes have a program counter specifying the location of the next instruction to execute. Instructions are executed sequentially, until the end of the program is reached. Multi-threaded processes have one program counter per thread.

If a system has more cores, each core has its own program counter.

1.4.5 Memory management

To execute a program, the instructions must be in memory. Memory management is handled by the operating system and has the following goals:

- Keeping track of which parts of memory are currently being used and by whom
- Deciding which processes (or parts thereof) and data to move into and out of memory
- Allocating and deallocating memory space as needed

1.4.6 Storage management

The OS provides a logical view of the storage and abstracts the physical properties in **files**. Files are organized in directories and there is usually an access control system. The OS deals with free-space management, storage allocation and disk scheduling.

The memory is therefore organized in a hierarchy, where each level offers different access speeds. While transferring data from a level to another, the OS must ensure that the data stays consistent. Moreover, multiprocessor environment must provide cache coherency in hardware such that all CPUs have the most recent value in their cache.

1.4.7 I/O management

The OS hides the peculiarities of hardware devices from the user using I/O subsystems. These subsystems are responsible for the device-driver interfaces and memory management of I/O including buffering (storing data temporarily while it is being transferred), caching (storing parts of data in faster storage for performance), spooling (the overlapping of output of one job with input of other jobs).

1.4.8 OS protection

OS must provide mechanisms to defend the system against external attacks. An attack is anything posing a threat to confidentiality, availability or integrity. For example OS distinguish among users, where each has a specific set of privileges. Privilege escalation is an attack where a user can gain privileges of a more privileged user.

1.4.9 Computing environments

There exist many computing environments, such as:

- Stand-alone general purpose machines
- Network computers (thin clients)
- Mobile computers
- Real-time embedded systems: operating system that runs processes with very important time constraints
- Cloud computing
- Distributed computing: many systems connected together over a network (client-server or peer-to-peer)
- Virtualization: guest OS emulates another OS or hardware and runs software on it. The program that manages this is called VMM (Virtual machine manager).

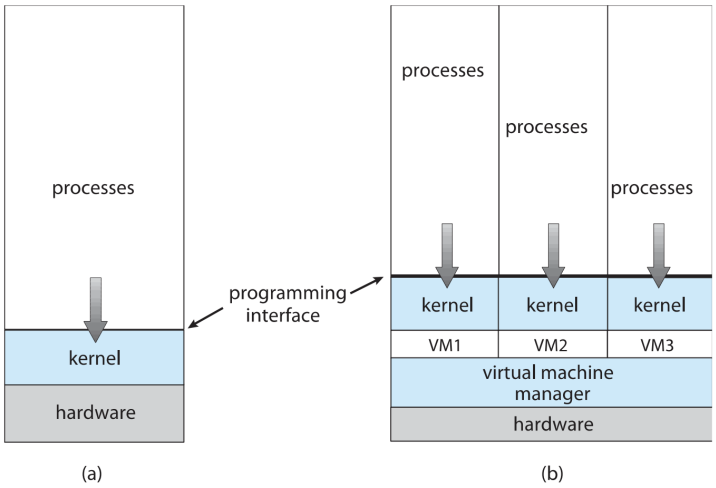


Figure 1.6: Virtualization

1.5 Services provided by operating systems

Operating systems provide the following services:

- User interface: can be command-line (shell - specific flavour of CLI, can be implemented in kernel or as a system program), Graphics User Interface (GUI)
- Program execution - The system must be able to load a program into memory and to run that program
- I/O operations
- File-system manipulation

- Communication between processes
- Error detection: errors may occur in CPU and memory hardware, in I/O devices, in user program
- Resource allocation: when multiple users or multiple jobs running concurrently, resources must be allocated to each of them
- Accounting: to keep track of which users use how much and what kinds of computer resources
- Protection (i.e. control access to system resources)
- Security (guarantees protection against threats)

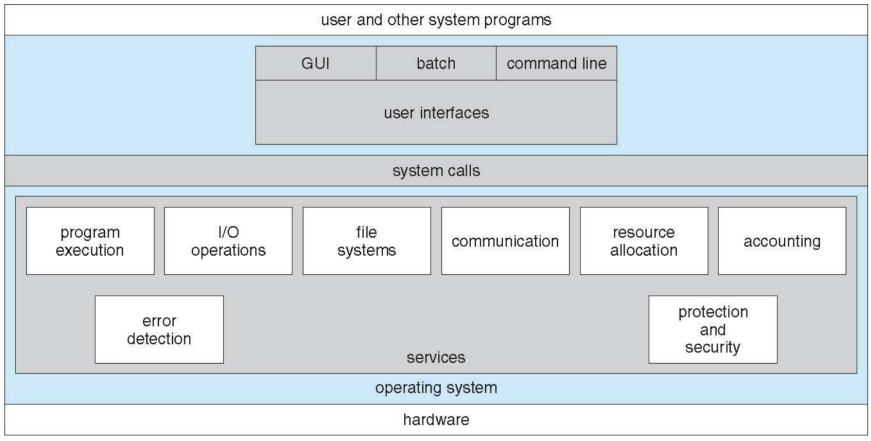


Figure 1.7: Services provided by an operating system

1.6 System calls

System calls are an interface provided by the operating system to interact with it. They are mostly accessed by using a high-level API provided by a language such as C, C++ etc. APIs are the preferred method to interact with the system because they are simpler and easier to use that direct system calls and provide portability to all systems that support that API. Examples of APIs are the POSIX API for UNIX systems, the Win32 API for Windows systems and the Java API for machines running the JVM. API calls are translated into system calls by the system call interface. Usually the translation is done at runtime. This is because if the translation were to be done at compile time the compiled program would be much larger in size and much more platform dependant.

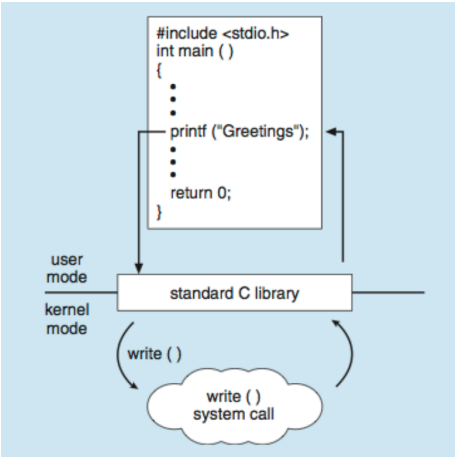


Figure 1.8: The `printf()` function in C uses the `write()` system call to print to the screen

1.6.1 Parameter passing

A system call usually requires some parameters, for ex. the `open_file()` system call needs to know the name of the file. Parameters can be passed to the system call by using registers. If there are not enough registers they can be stored in the stack or inside a table in memory. Then just the pointer to the data is passed to the system call.

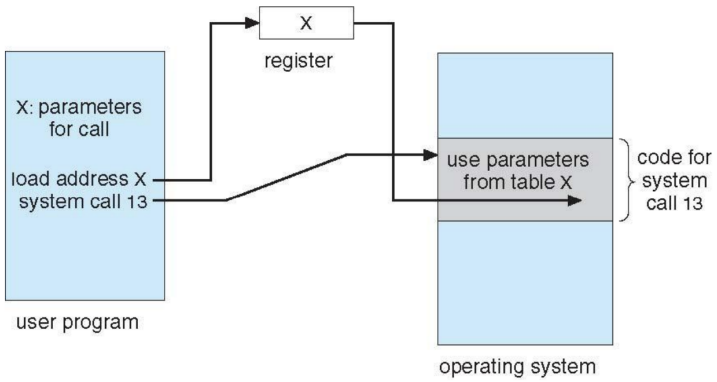


Figure 1.9: Parameter passing

Examples of system calls:

- Process management: create, terminate, load, execute, get process attributes, set process attributes, wait time, wait event, signal event, dump memory on error, single step executing for debugging, locks for managing shared data
- File management: create, open, delete, read, write
- Device management: request device, release device, read, write, get device attributes, set device attributes
- Information maintenance: get time or date, set time or date, get system data, set system data
- Communications: send/receive messages, open/close connection, gain access to shared memory
- Protection: control access to resources, get and set permissions, allow/deny user access

1.7 System programs

All operating systems come bundled with convenient programs, which ease program execution and development. System programs can be part of the following categories:

- File manipulation/management
- Status information: date, time, available resources, logging, debugging information, performance statistics, registry (database for storing configurations for programs)
- Program language support: compilers, assemblers, debuggers, interpreters
- Program loading and execution: linkers, loaders
- Communications: provide mechanisms for communications among processes, users and computer systems
- Background services: usually start at boot and may run fow whole time system is on, known as services or deamons, perform disk checking, process scheduling, error loading, printing

1.8 OS design and implementation

The first problem in designing a system is to define goals and specifications. The goals can be divided into two basic groups: user goals and system goals. User goals could be that the operating system should be convenient to use, easy to learn, reliable, safe and fast. System goals could be that the operating system should be easy to design, implement and maintain, flexible, reliable, error-free and efficient.

One important principle is OS design is the separation of policy from mechanism. Policies define what needs to be done, while mechanisms define how it should be done. This separation is useful because policies are likely to change, but most of the times the mechanism doesn't need to be changed substantially. For example, a mechanism could be process scheduling, and possible policies are to schedule processes that use disk first or processes from a given user first or daemons first etc.

1.9 OS structure

OSs may be structured in different ways or may be designed according to different architectures.

1.9.1 Simple structure - MS-DOS

MS-DOS has a very simple structure: a shell starts a program and when the process ends the shell is rebooted into a new program. There is at most one process running. When a new program starts, it overwrites all memory except the part that contains the kernel.

1.9.2 Monolithic kernel - UNIX

Originally UNIX had a monolithic structure. The kernel manages everything below the system call interface and above the hardware. For example, it manages the file system, CPU scheduling, memory management. Basically, the kernel managed The advantages of using a monolithic kernel are that it is fast and energy-efficient, but it is not modular and even small changes require refactoring of the code and recompilation of the whole OS.

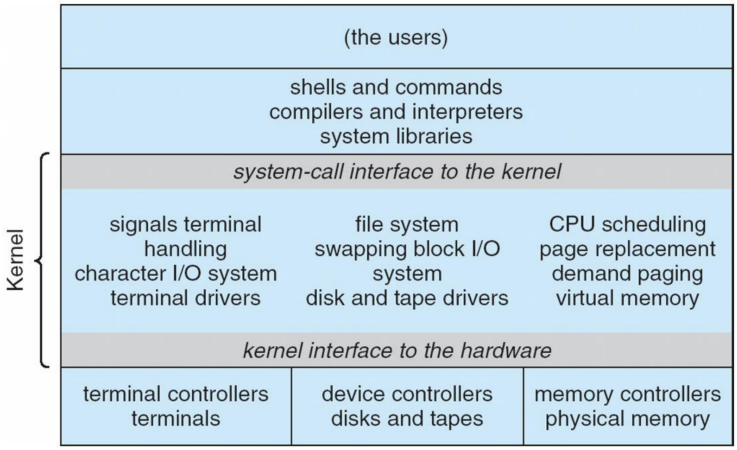


Figure 1.10: UNIX structure

1.9.3 Layered approach

The operating system is divided into multiple layers, where each layer is built on top of the lower layers (similar to ISO/ISO reference model and TCP/IP stack). This allows for more modularity and a change of one layer doesn't always imply a recompilation of the whole operating system. An example of a possible layered structure is the following: hardware -> drivers -> file system -> error detection and protection -> user programs.

1.9.4 Microkernel

The microkernel approach moves processes as much as possible outside the kernel into the user space. Communication between user modules is achieved using message passing though the kernel. The advantages of this approach are full modularity and extendability, security (a malicious process can't damage others) and reliability (less code is running in kernel mode). Communicating through the kernel introduces additional overhead, thus has a negative performance impact.

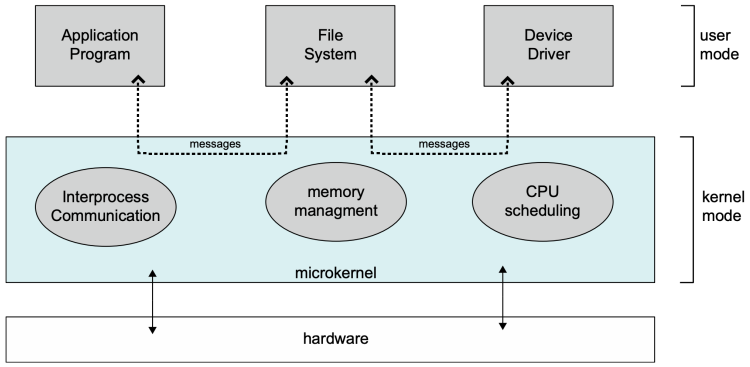


Figure 1.11: Microkernel structure

1.9.5 Modular OS

Modern OSes implement loadable kernel modules, which can be loaded on-demand at runtime. This structure follows the object-oriented approach. The base kernel contains only the core functions such as message passing and process management. Modules talk to each other over known interfaces. Examples of modules are device drivers, system calls, network drivers and interpreters.

Chapter 2

Process management

A process is a program in execution. Processes are identified by a **process identifier** (pid). The OS allocates some space in RAM for the process in the following way:

- Text section: the program code
- Data section: contains global variables (initialized and uninitialized)
- Heap: memory dynamically allocated during runtime
- Stack: contains temporarily variables, such as function parameters, return addresses, local variables

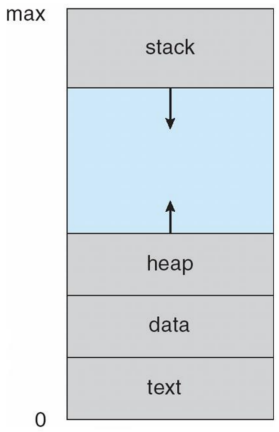


Figure 2.1: Memory layout for a process

A process during execution cycles through the following states:

- new: the process is created
- ready: The process is in a queue and is waiting to be assigned to a processor
- running: Instructions are being executed
- waiting: The process is in a queue and is waiting for some event to occur (ex. a memory transfer, an I/O)
- terminated: The process has finished execution

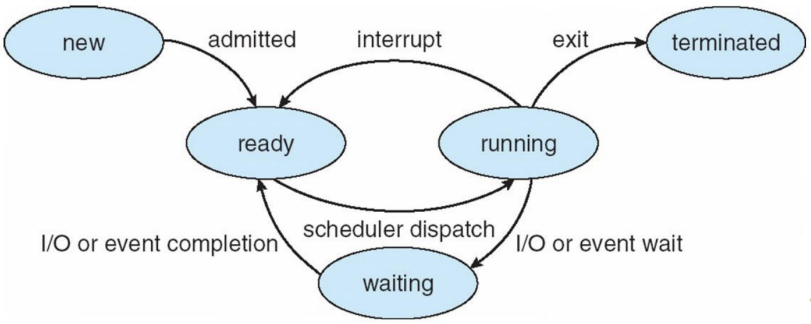


Figure 2.2: Process state machine

The OS keeps track of the state of the process in a data structure stored in RAM called process control block (PCB). The PCB contains the following information:

- Process state: running, waiting, etc.
- Program counter: location of next instruction
- CPU registers: contents of registers used by the process
- CPU scheduling information: priorities, scheduling queue pointers
- Memory-management information: memory allocated to the process
- Accounting/Debug information: CPU used, clock time elapsed since start, time limits
- I/O status information: I/O devices allocated to process, list of open files

In Linux the PCB for every process is stored as a file in the /proc folder: `less /proc/<pid::self>/status`.

A **context switch** is the process of storing the state of the currently executing process in the PCB (so that its execution can be resumed at a later point) and loading the PCB of the next process. Performance-wise, context switches are pure overhead, since they do not perform actual work. Context switches can be categorized in:

- voluntary context switch: the process stops itself because needs to wait for a resource
- nonvoluntary context switch: the processor decides to switch process

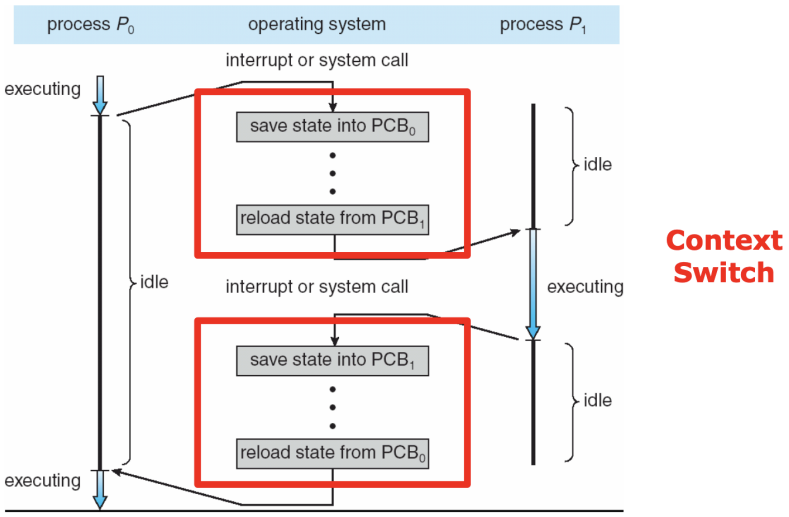


Figure 2.3: Context switch

Modern OSes have extended the process model to include threads. By using threads programs can execute multiple tasks at the same time. OSes that support threads store their information in the PCB of the process that started them.

2.1 Scheduling

The CPU has a process scheduler, which decides which process to execute. The job of the CPU scheduler is to maximize CPU utilization, i.e. having a process running at all times and switching quickly among processes to achieve effective time sharing. The scheduler stores the processes in various queues:

- Job queue: set of all processes in the system
- Ready queue: set of all processes residing in main memory, ready and waiting to execute
- Device queues: set of processes waiting for an I/O device

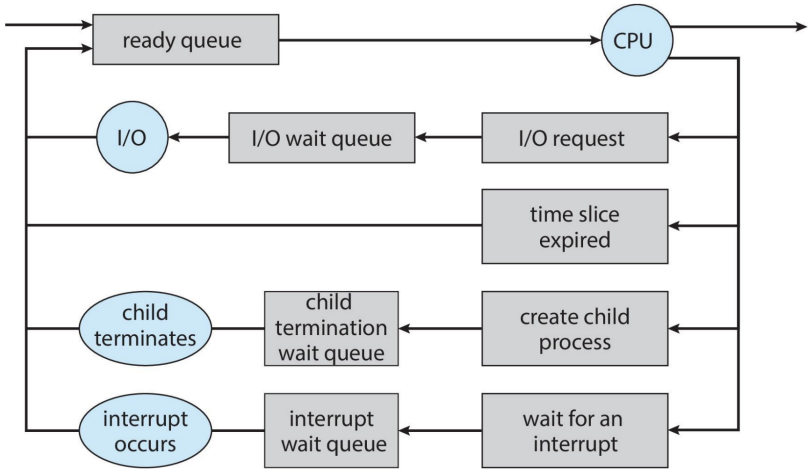


Figure 2.4: Process queues

2.2 Process creation

Processes are arranged in a tree structure: process can create other *child* processes, which in turn can have other children. In Linux the process tree can be printed using `pstree`.

Resources among the parent and children can be shared in different ways:

- Parent and children share all resources
- Children share subset of parent's resources
- Parent and child share no resources

Moreover they have different options for execution:

- Parent and children execute concurrently
- Parent waits until children terminate

If there is no parent waiting for the child process, then the child process is called zombie process. If the parent is instead terminated, the child process is called orphan process.

Some OSes don't allow processes without a parent. In such OSes, if the parent process terminates, then also all child processes get killed.

In a Linux system the root process that spawns all other processes is called `systemd`.

In UNIX processes are managed using the following system calls:

- `fork()`: creates a new process by copying all data structures
- `clone()`: creates a new thread which uses the same data structures as the parent process
- `exec()`: replaces the parent's memory with the children's one (machine code, data, heap, and stack)
- `wait()`: called by parent to wait for the end of the child's execution
- `abort()`: called by parent to kill the child process

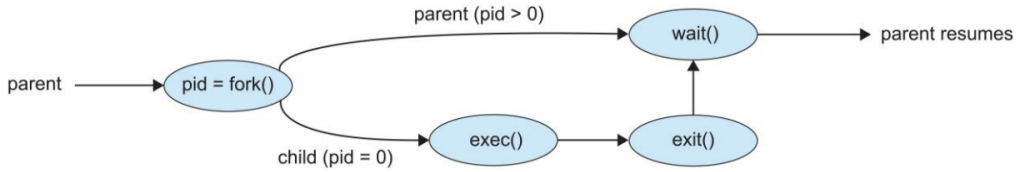


Figure 2.5: Creation of children processes

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

int main() {
    pid_t pid;

    // Returns 0 if called from the child process
    // Returns the PID of the child process of -1 on error
    // if called from the parent process
    pid = fork();

    if (pid < 0) {
        fprintf(stderr, "Fork failed\n");
        return 1;
    } else if (pid == 0) {
        printf("Child print\n");
    } else {
        wait(NULL); // Waits for the child process to finish executing
        printf("Parent print after child\n");
    }
}
```

Listing 1: A process that spawns a child process and waits for its termination

2.3 Communication between processes

Processes can communicate using:

- shared memory: processes that wish to communicate create a shared area of memory, that is managed directly by the processes
- message passing

A common paradigm for inter-process communication is the producer-consumer model. A producer process produces information that is consumed by a consumer process. The information passes through a buffer, which can be of bounded or unbounded size. If it is bounded and the buffer is full, then the producer must wait, while if it is unbounded the producer does never have to wait.

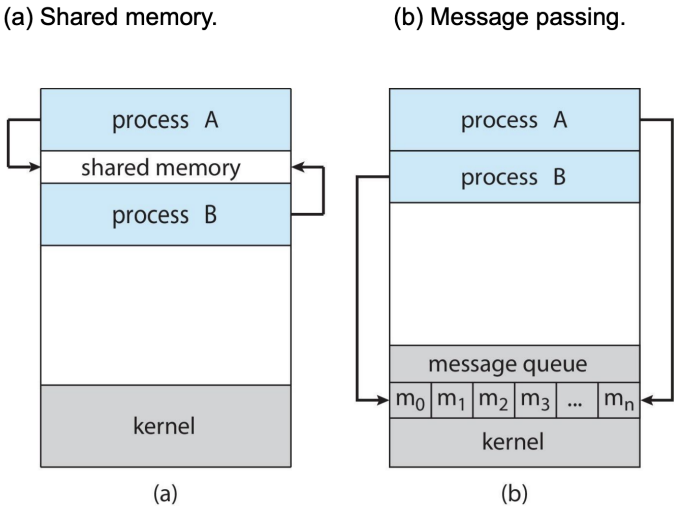


Figure 2.6: Models of communication between processes

2.3.1 Shared memory

Processes can communicate using shared memory. Processes can allocate a part in RAM as shared memory. Then they can access it by mapping it to their address space[1]. The size of the shared memory is controlled by the processes, not by the OS. In UNIX memory mapping is done by the `mmap()` system call.

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

[1] array of addresses that the process is allowed to use

```c
int main()
{
  const int SIZE = 16;
  /* name of the shared memory */
  const char *name = "OS";
  const char *message0 = "Hello world";
  const char *message1 = "I'm a shared message";

  /* shared memory file descriptor */
  int shm_fd;
  /* pointer to a shared memory object */
  void *ptr;

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

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

  /* map the shared memory segment in the address space of the process
   ↪  */
  ptr = mmap(0,SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
  if (ptr == MAP_FAILED) {
    printf("Map failed\n");
    return -1;
  }

  /**
   * write to the shared memory region. Note we must increment the
    ↪  value of ptr after each write.
   */
  sprintf(ptr,"%s",message0);
  ptr += strlen(message0);
  sprintf(ptr,"%s",message1);
  ptr += strlen(message1);
  return 0;
}
```

Listing 2: A process that creates a shared memory area and writes to it

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

```c
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>

int main()
{
  const char *name = "OS";
  const int SIZE = 16;

  int shm_fd;
  void *ptr;
  int i;

  /* open the shared memory segment */
  shm_fd = shm_open(name, O_RDONLY, 0666);
  if (shm_fd == -1) {
    printf("shared memory failed\n");
    exit(-1);
  }

  /* map the shared memory segment in the address space of the process
  ↪ */
  ptr = mmap(0,SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);
  if (ptr == MAP_FAILED) {
    printf("Map failed\n");
    exit(-1);
  }

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

  /* remove the shared memory segment */
  if (shm_unlink(name) == -1) {
    printf("Error removing %s\n",name);
    exit(-1);
  }

  return 0;
}
```

Listing 3: A process that opens a shared memory area and reads from it

2.3.2 Message passing

Alternatively processes can communicate using message passing. Message passing provides two operations: `send(message)` and `receive(message)`. Depending on the implementation, the message size can be fixed or variable. Message passing can be implemented using the following channels:

- Shared memory (already seen in the previous section)
- Shared hardware bus
- Network

We can distinguish the channel on a logical level as follows:

- Direct or indirect
  In direct communication, processes must name each other explicitly and a link is associated with exactly one pair of communicating processes. In indirect communication, message is sent in a queue of a "mailbox". Each mailbox is identified by an id and multiple processes can send and receive messages from it. Multiple processes reading from a mailbox at the same time should be avoided in the following ways: by allowing a link to be associated with at most two processes, by allowing only one process at a time to execute a receive operation or by letting the system select arbitrarily the receiver.
- Synchronous (blocking) or asynchronous (non-blocking)
  - Blocking send - the sender is blocked until the message is received
  - Blocking receive - the receiver is blocked until a message is available
  - Non-blocking send - the sender sends the message and continues
  - Non-blocking receive - the receiver receives either a valid message or null
- Automatic or explicit buffering
  1. Zero capacity - no messages are queued on a link (sender must wait for receiver)
  2. Bounded capacity (explicit buffering) - finite length of n messages (sender must wait only if link full)
  3. Unbounded capacity (automatic buffering) - infinite length (sender never waits)

Pipes

Pipes provide a way for to processes to communicate with each other. OSes implement two types of pipes: ordinary pipes and named pipes.

Ordinary (or unnamed pipes or anonymous pipes in Windows) cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created. Ordinary pipes are unidirectional, which means that the parent process can only write to it and the child process can only read from it.

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

```c
#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;
}
```

Listing 4: A process that spawn a child and communicates to it using an ordinary pipe

Named pipes can be accessed outside a parent-child relationship. They are bidirectional and multiple processes can read and write to it. When no process holds a reference to the file descriptor the pipe is destroyed by the system.

```
#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);
}
```

Listing 5: A process that creates a named pipe and writes into it the content from the standard input

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

Listing 6: A process that reads from pipe and writes its content to the standard input

2.4 Threads

A process can execute multiple instructions at once by using multiple threads. The OS keeps track of threads in the PCB. Threads share the same data and text (code) section and OS resources of the parent process. On the contrary each thread has its own thread id, register data, stack and program counter.

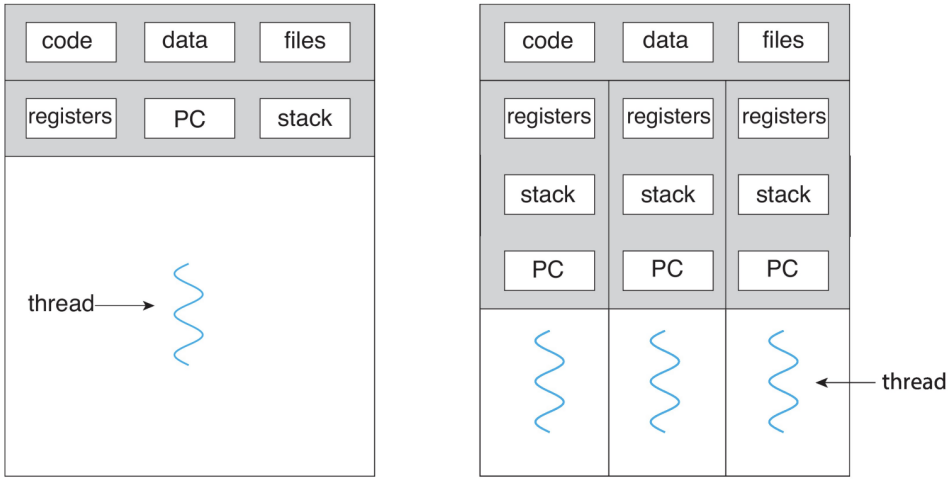


Figure 2.7: Single threaded process (left) and multithreaded process (right)

2.4.1 Difference between child processes and threads

When a process is forked all information is duplicated (data, code, files), while threads of the same process share the same information. When a thread is created, the parent process has to specify which part of the code the thread has to execute. Therefore threads are more lightweight: resources are shared by default (meanwhile processes have to communicate using shared memory or message passing), they are quicker to create, use less memory, context switching is faster and can scale easily (can take advantage of multi-core architectures).

2.4.2 Applications of threads

Threads are used in client-server programs such as web server. Every time a request is received, the program creates a thread which fulfills the request. Another example is running background tasks, such as the spellchecker in a Word program.

2.4.3 Linux threads

Threads are created using the `clone()` call. Threads shares the address space of the parent task. Flags can be passed to the call to control the behavior of the calls.

The Linux scheduler doesn't make distinctions between threads and processes, but sees all of them as schedulable tasks.

2.4.4 Multi-core programming

The advantage of having multi-core systems is that tasks can be parallelized. There are two types of parallelism:

- Data parallelism: data is distributed across multiple cores, each thread performs same operation (ex. quicksort)
- Task parallelism: tasks are distributed across multiple cores, each thread performs a different operation (ex. pipelining)

The challenges with maintaining multi-core or multiprocessor systems are: dividing activities, load balancing, data splitting and data dependency, testing and debugging.

Concurrency means supporting more than one task by allowing all tasks to make progress. Is can be implemented even in single-core/single processor system by using time sharing.

Parallelism means that the system can perform more than one task simultaneously. Therefore a system can support concurrency without parallelism.

2.4.5 Amdahl's Law

Amdahl's Law describes the theoretical performance gain by using parallel code.

$$\text{speedup} \leq \left(S + \frac{1-S}{N} \right)^{-1}$$

Where S is the percentage of code that has to be serial, N is the number of cores, and speedup $= 1$ means that there is no speedup.

2.4.6 Kernel threads

User threads need to be mapped to kernel threads to be executed. A kernel thread is a kernel entity (an entity that can be handled by the system scheduler), like processes and interrupt handlers. User threads can be mapped to kernel threads in three ways: one-to-one (preferred by Linux and Windows), many-to-one, many-to-many. In the many-to-one mapping one thread causes all threads to block, therefore cannot take advantage of multithreading. The one-to-one mapping takes full advantage of multithreading (one blocking call doesn't cause all other threads to be blocked), but many threads may be created, causing a lot of overhead. In the many-to-many mapping, OS can choose how many threads it needs to manage. This mapping is more complicated to implement and manage.

2.5 Thread libraries

Thread libraries provide the programmer an API for creating and managing threads. They can be implemented entirely in user-space or the OS bay provide them at the kernel level. For example, pthreads is an API defined by the POSIX standard. Since it is just a specification, the implementation is up to the developer of the library.

2.6 Implicit threading

Implicit threading is a strategy which transfers the burden of the creation and management of threads done to compilers and run-time libraries rather than programmers. It is a technique which is growing in popularity since as the numbers of threads increase, guaranteeing program correctness becomes more and more difficult with explicit threads. Programmers can take advantage of implicit threading in using different tools:

- Thread pools
  Thread pools is a design pattern where a fixed amount of threads is created, which all wait for work. It is justified because it is usually slightly faster to service a request with an existing thread than creating a new thread, the total number of threads is always bounded and threads can be scheduled to run programmatically
- Fork join parallelism
  In this strategy, multiple threads are forked and then joined
- OpenMP
  In OpenMP the programmer defines parallel regions, which then get automatically executed in parallel

Chapter 3

CPU Scheduling

The system executes tasks in cycles made of CPU bursts and I/O bursts. During a CPU burst a process or program demands and actively utilizes the CPU for computation, while during an I/O burst a process or program waits for data to be read from or written to external storage devices.

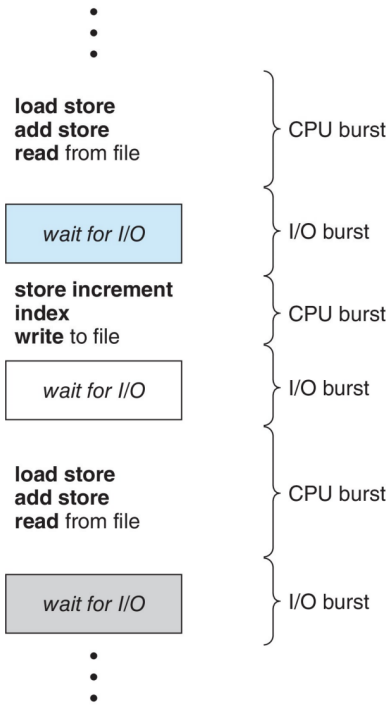


Figure 3.1: CPU bursts and IO bursts

Generally CPU bursts are short.

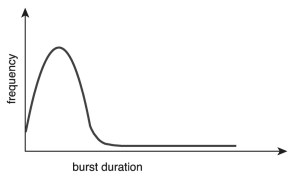


Figure 3.2: CPU burst duration frequency

The job of the CPU scheduler is to select among the processes in the ready queue and allocate the CPU to them for some time.

3.1 Preemptive scheduling

Preemption is the act of temporarily interrupting an executing task, with the intention of resuming it at a later time. Preemptive systems are therefore able to suspend tasks that take a long time, put them back in the ready queue and resume them at a later time.

Preemptive scheduling can result in race conditions when data is shared among several processes. For example if a thread is interrupted while is writing some data, the information can be left in an inconsistent state, which can become a problem if other threads are reading it.

The unit that gives control of the CPU to a new task is called the dispatcher. The dispatcher handles the context switch, switches to user mode and jumps to the proper location in the program to start execution. The dispatcher latency should be as low as possible.

3.2 Scheduling metrics

The following are the most important metrics to evaluate the performance of a scheduler:

- 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 (completion time - arrival time)
- 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

3.3 Scheduling algorithms

There exist multiple scheduling algorithms, each one prioritizing certain metrics at the expense of others.

3.3.1 First-come, first-served scheduling (FCFS)

FCFS executes tasks in the order they arrive in the queue. FCFS scheduling is non-preemptive: once a process is started, it will execute until the next I/O burst or the end of the process. The average waiting time is highly dependent on the order in which tasks arrive. For example, if a long task arrives before some short tasks, the latter will have a high waiting time (convoy effect).

3.3.2 Shortest-job-first scheduling (SJF)

In SJF tasks are executed in order of their burst time: the task with the shortest burst time is executed first. SJF is non-preemptive and theoretically minimizes the average waiting time. The problem of this strategy is that estimating CPU bursts is unfeasible, because it would require estimating how much time a task requires to be executed. If we assume that the individual CPU burst times of a task are correlated, we could compute an average of the previous burst times of a task and make an estimate for the next burst time (for example using an exponential average of the previous samples).

Shortest time remaining first scheduling (SRT) is the preemptive version of SJF: every time a new task arrives, the scheduler will stop execution of the current task and will execute the task with the current shortest burst time first. In general SRT is better than the non-preemptive version.

3.3.3 Round robin scheduling

In round robin scheduling each process gets a small unit of CPU time (called time quantum q), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue. q must be large with respect to context switch (otherwise the overhead is too high), but q not too high (otherwise performance will be similar to FCFS). If there are n processes in the ready queue, no process waits more than $q(n-1)$ time units. Typically, this scheduling has a higher average turnaround time than SJF, but better response time.

3.3.4 Priority scheduling

In priority scheduling a priority number is associated with each process. The CPU is allocated to the process with the highest priority. The problem of this strategy is that low priority processes may never execute (starvation). This can be solved by increasing the priority of the process as time progresses (aging).

The processes that have the same priority will be executed using round-robin.

3.3.5 Multilevel queues

In this scheduling strategy there are multiple queues, where each one has a different scheduling strategy and a propriety number. The CPU will execute the first task in the highest propriety queue. Tasks can switch queues if their priority changes.

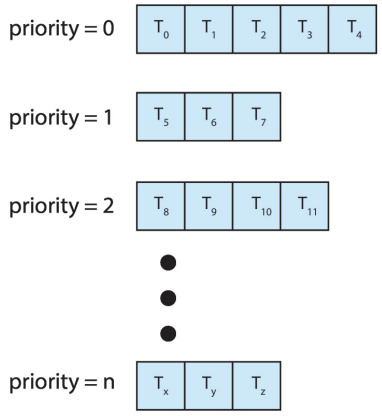


Figure 3.3: Multilevels queues

3.3.6 Multiple-processor scheduling

CPU scheduling is more complex when multiple CPUs are available. In multiprocessor systems each processor generally independently decides which thread to execute next (symmetric multiprocessing). All threads may be in a common ready queue or each processor may have its own queue.

In modern multicore processors a single core can assign itself multiple threads and switch among them when a memory stall happens (in Intel CPUs this feature is called multithreading). The physical core exposes itself to the OS as multiple virtual cores.

When a thread has been running on one processor, the cache contents of that processor stores the memory information accessed by that thread. This is referred to as a thread having affinity for a processor ("processor affinity"). If then that thread gets executed on another processor, it will lose all the cached content. To address this inefficiency, the OS can try to keep a thread running on the same processor (soft affinity) or force it to run on the same processor (hard affinity).

3.3.7 Real-time scheduling

Real-time operating systems must be able to meet deadlines for certain tasks deemed as critical. Priority-based scheduling don't provide this guarantee. Systems where there is no guarantee as to when tasks will be scheduled are called soft real-time systems, while system that can guarantee that tasks meet the deadline are called hard real-time systems. Real-time systems also have periodic tasks: tasks that require the CPU at constant intervals.

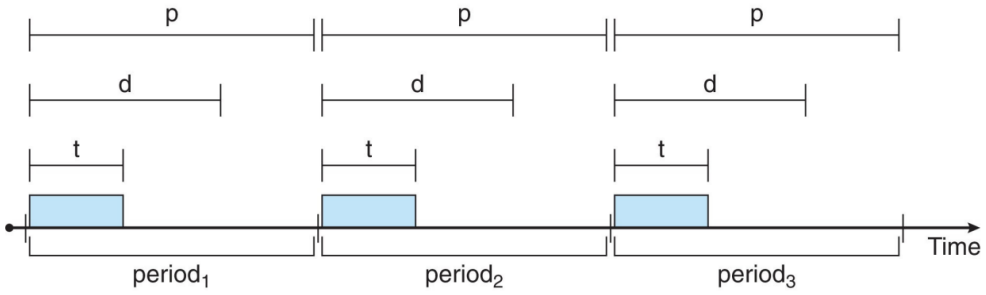


Figure 3.4: Periodic task with processing time t, deadline d and period p

Rate monotonic scheduling

Each process gets assigned priority based on the inverse of the period of the task. Therefore tasks with shorter periods have the highest priority. This scheduler is not ideal, because tasks with longer periods will be constantly interrupted by tasks with shorter periods, making them miss the deadline.

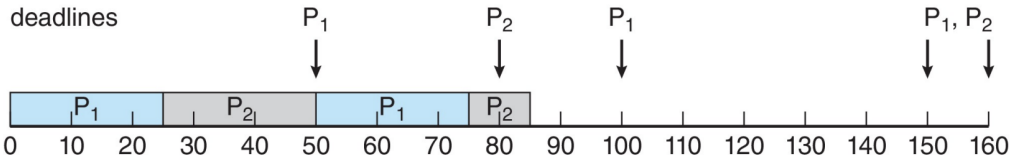


Figure 3.5: Task P2 misses the deadline because task P1 always has the highest priority

Earliest Deadline First Scheduling

Priorities are assigned according to the time missing to the next deadline: the earlier the deadline, the higher the priority.

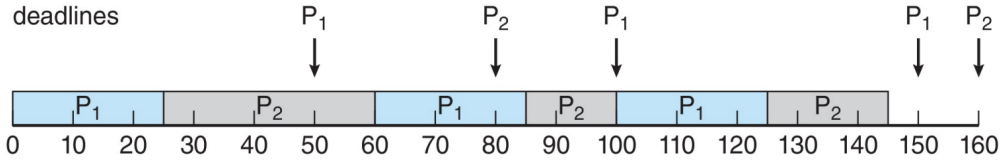


Figure 3.6: Earliest Deadline First Scheduling

3.4 Scheduling evaluation

There are various ways in which scheduling algorithms can be evaluated and compared.

3.4.1 Deterministic modeling

In deterministic modeling the scheduling algorithms are evaluated on a particular workload. The metrics are then collected and compared. This approach is not always ideal because it considers the behavior of the algorithms in one scenario only, but is simple to perform and fast.

3.4.2 Queueing models

A scheduling algorithm can be modeled mathematically: the arrival of processes, and CPU and I/O bursts is described probabilistically. Then the various metrics are collected and compared.

Little's formula is a general formula that states that in steady state, processes leaving queue must equal processes arriving.

$$n = \lambda \cdot W$$

n is the average queue length, W is the average waiting time in queue and λ is the average arrival rate into queue. This formula is valid for any scheduling algorithm and arrival distribution.

3.4.3 Simulations

The algorithms can be evaluated by looking at their metrics by analyzing their behavior on multiple workloads.

Chapter 4

Process synchronization

In modern operating systems processes can run concurrently. Although concurrency allows to achieve very high performance, it introduces some problems that need to be managed.

A situation when several processes access and manipulate the same data concurrently and the outcome of execution depends on the order in which the access takes place, is called a race condition. Because such situations occur frequently (especially in multithreaded systems), processes need to be synchronized and coordinated.

4.1 Critical section problem

The critical section problem happens when multiple processes are writing and reading from some shared data at the same time. The part of the code that modifies some shared data (such as updating common variables, writing to a file, updating tables...) is called **critical section**.

A solution to the critical section problem must satisfy the following requirements:

1. Mutual Exclusion: if process Pi is executing its critical section, then no other processes can be executing in their critical sections
2. Progress: if no process is executing its critical section, then a process that want to execute its critical section should be able to do so
3. 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 (note that this does not imply anything on the time a process stays in its critical section)

4.1.1 Peterson's solution

The Peterson's solution applies to single-core processor and with two processes. The processes share two variables: `int turn` and `boolean flag[n]`. The `turn` variable indicates whose turn it is to enter the critical section. The `flag` array is used to indicate if a process is ready to enter the critical section.

```
while (true) {
    flag[i] = true; // Mark that i wants to execute the CS
    turn = j; // Give process j the possibility to execute its CS
    while (flag[j] && turn == j); // Wait for process j to finish
    /* Critical section */
    flag[i] = false; // Mark that i finished executing its CS
}
```

Listing 7: Implementation of Peterson's solution for process i with two processes i and j

This solution satisfies the three CS requirements:

1. Mutual exclusion is preserved (process i enters CS only if: either flag[j] = false or turn = i)
2. Progress requirement is satisfied
3. Bounded-waiting requirement is met (because of alternating turns)

Although Peterson's process is theoretically perfect, modern compilers and architectures perform various optimizations on the code. One of them is to change the order in which instructions are executed if they detect that it does not change the logic of the individual process. Therefore the `flag[i] = false` statement is not guaranteed to be executed exactly at the end of the critical section, thus breaking the synchronization mechanism.

This can be fixed by using memory barriers. When a memory barrier instruction is performed, the system ensures that all loads and stores of all processes are completed before any subsequent load or store operations are performed. Therefore we can add a memory barrier before setting the flag to false in the while loop.

4.2 Hardware solutions

For single processor systems a simple solution to implement critical sections would be to disable interrupts when executing them. In this way the process cannot be preempted and can execute until the end of the critical section. This solution si very inefficient performance-wise. Many systems therefore provide hardware instructions for implementing the critical section code. Two such instructions are:

- test-and-set instruction
- compare-and swap instruction

These instructions must run atomically to work.

4.2.1 Test and set instruction

To access a critical section processes have to first call the `test_and_set()` instruction, which must be executed atomically.

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

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

Listing 8: Implementation of the `test_and_set()` function

This solution satisfies mutual exclusion and progress, but it does not satisfy the bounded waiting requirement, because there is no mechanism that ensures that a process that requested access will enter the critical section.

4.2.2 Compare and swap instruction

The `compare_and_swap()` function must be executed atomically. When used with the lock it ensures that nobody else is reading or modifying the lock when it is executing.

To access a critical section processes have to first call the `compare_and_swap()` instruction on the lock and check if it is their turn. Then they will give the lock to the next process in line or release the lock.

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

// Example of usage
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;
```

```
    // Find next process in line that requested access
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        // Release the lock for everybody
        lock = 0;
    else
        // Give lock to next process in line that requested access
        waiting[j] = false;
    /* End of critical section */
}
```

Listing 9: Implementation of the `compare_and_swap()` function

This solution satisfies all requirements for the critical section.

4.2.3 Atomic variables

The `lock` variable from the previous example is an example of an **atomic variable**, i.e. a variable whose value changes atomically. Typically, instructions such as compare-and-swap are used as building blocks for other synchronization tools. An atomic variable is a variable that provides atomic operations on basic data types such as integers and booleans.

4.3 Software solutions

Previous solutions are complicated and generally inaccessible to application programmers. In this section some software solutions will be shown, under the assumption that the programmer has access to some atomic functions.

4.3.1 Mutex locks

A simple solution for solving the critical section problem in software is using a mutex lock. A mutex lock is a boolean variable indicating if lock is available or not. When a process wants to access a critical section, it tries to acquire the lock and waits until it is able to do so using the `acquire()` atomic function. Then, it releases it when it is done executing using the `release()` atomic function. This solution requires busy waiting, which is undesirable. While a process is in its critical section, any other process that tries to enter its critical section must wait in a loop (busy waiting). This lock is therefore called spinlock.

4.3.2 Semaphores

Semaphores are similar to mutex locks, but can be generalized to more than one process. Let `S` be an integer variable. When a process wants to enter its critical section, it calls

the `wait(S)` atomic function, which will wait until the `S` variable is one. Then it will decrement it back to zero and let the process execute its critical section. After the process has finished executing its critical section it will call the `signal(S)` atomic function. This function will increment `S`, thus signalling to the next process that it can execute. This solution is also affected by the busy waiting problem. Depending on the domain of the integer variable S, there are two types of semaphores:

- Counting semaphore: integer spans an unrestricted domain
- Binary semaphore: integer can only be 0 or 1

4.3.3 Semaphore without busy waiting

Semaphores can be implemented without suffering from the busy waiting problem. This can be done by storing in the semaphore object the queue of processes that wish to execute their critical section. When the `wait()` function is called, the S variable is decremented, the process is added to the queue and then it is sent to sleep (in this way busy waiting is avoided). When the `signal()` function is called, S is incremented and a check will be done to see if there are processes waiting in the queue (true if $S \leq 0$). If there are processes waiting, the first process gets removed from the list and woken up.

4.3.4 Monitors

Monitors are structures that abstract synchronization and expose to the programmer a set of functions to access the data that the monitor holds. The monitor structure supports condition variables: these variables have a `x.wait()` method, which suspends the code of that process until the `x.signal()` method is called.

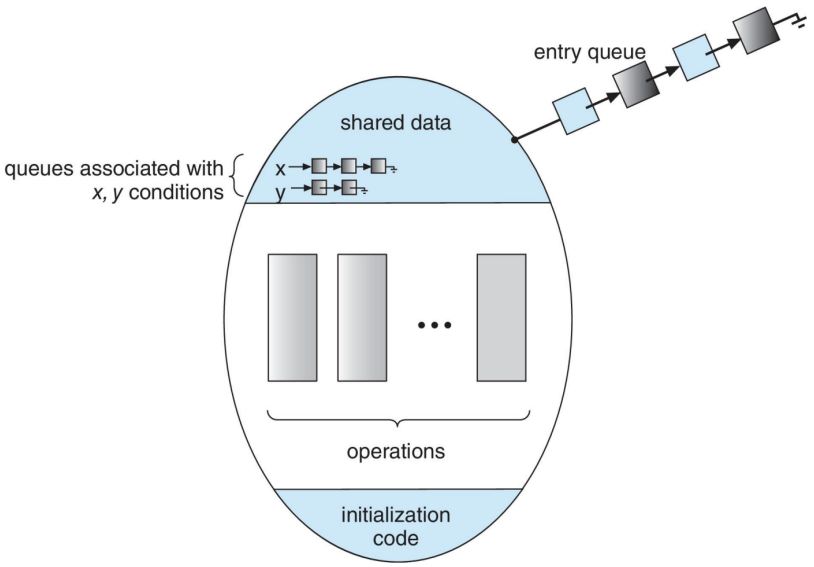


Figure 4.1: Monitor with condition variables

Monitors can be implemented using semaphores and mutex.

Resource allocator

The resource allocator is an example of a monitor. Assume that a single resource has to be shared among multiple processes which will use it for a specific amount of time (given by the priority number). This can be implemented using a monitor structure implementing an `acquire()` and `release()` procedure. The `x.wait(c)` is a *conditional-wait* function, where c is the priority. When `x.signal()` is called, the process with the lowest priority number will be executed.

```
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;
    }
}
```

Listing 10: Monitor for resource allocator

4.4 Liveness and deadlock

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.

Deadlock happens when two or more processes are waiting indefinitely for an event that can be caused only by one of the processes waiting. Deadlock is an example of a liveness failure. Other examples are starvation (when a process waits indefinitely to acquire resources or to be executed) and priority inversion. Priority inversion is a scheduling problem where a lower priority process holds the lock needed by a higher priority process. Priority inversion can be solved by using priority inheritance: 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.

4.5 Well-known synchronization problems

In the literature there are some well known synchronization problems which have been solved already, namely the bounded buffer problem, the readers-writers problem and the dining philosophers problem.

4.5.1 Bounded buffer problem

A bounded buffer lets multiple producers and multiple consumers share a single buffer. Producers write data to the beginning of the buffer, while consumers read data from the end of the buffer. Producers must stop pushing data if the buffer is full, and consumers must stop if the buffer is empty.

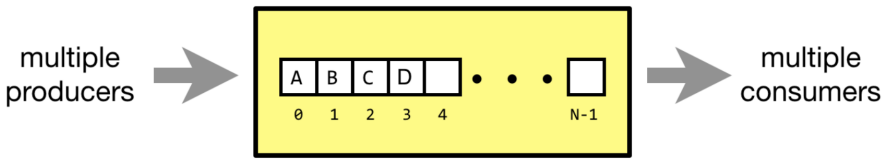


Figure 4.2: Bounded buffer with capacity n

```
// Free - semaphore with value of free slots in buffer
// Empty - semaphore with value of elements present in the buffer
// Mutex - binary semaphore that locks access to buffer to one process
 ↪  only

free <- number of slots in buffer
elements <- 0
mutex <- 1

// Producer
while (true) {
    wait(free); // If free = 0 wait, otherwise decrement free by 1 and
     ↪  continue
    wait(mutex); // If someone is reading/writing wait, otherwise
     ↪  continue
    produce();
    signal(mutex); // Release access of buffer
    signal(elements); // Increment elements by 1
}

// Consumer
while (true) {
    wait(elements); // If elements = 0 wait, otherwise decrement
     ↪  elements by 1 and continue
```

```
    wait(mutex); // If someone is reading/writing wait, otherwise
    ↪ continue
    consume();
    signal(mutex); // Release access of buffer
    signal(free); // Increment free by 1
}
```

Listing 11: Solution for bounded buffer problem

4.5.2 Readers-writers problem

A data set is shared among a number of concurrent processes: some processes are only readers, others can read and write. The problem to be solved is to allow multiple readers to read at the same time, but allowing only one writer to write at the same time.

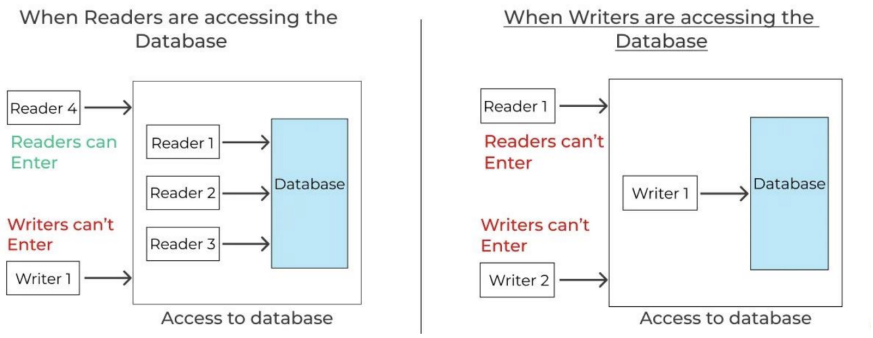


Figure 4.3: Readers-writers problem

```
// rw_mutex - lock to modify the data
// mutex - lock required to modify counter of readers

rw_mutex <- 1
mutex <- 1
read_count <- 0

// Writer
while (true) {
    wait(rw_mutex);
    write();
    signal(rw_mutex);
}

// Reader
while (true) {
    wait(mutex);
    read_count++;
```

```
    // Check if I'm the first reader and lock r/w access
    // To do so I have to be sure to be the only one, thus the need
    ↪  for the mutex lock
    if (read_count == 1) {
        wait(rw_mutex);
        signal(mutex);
    }
    signal(mutex);
    read();
    wait(mutex);
    read_count--;
    // Check if I'm the last reader and unlock r/w access
    if (read_count == 0) {
        signal(rw_mutex);
    }
    signal(mutex);
}
```

Listing 12: Solution for readers-writers problem

4.5.3 Dining philosophers

N philosophers sit at a round table. Each has one chopstick on the left and one on the right and a bowl of rice. Each philosopher can only alternately think and eat. Occasionally they try to pick up two chopsticks (one at a time) to eat from the bowl (need both to eat, then release both when done).

The problem is how to design a concurrent algorithm such that any philosopher will not starve; i.e., each can forever continue to alternate between eating and thinking.

Note that if each philosopher takes the right chopstick at the same time, then they will wait indefinitely for the other one and no one will eat, thus creating a deadlock.

There are several possible solutions that avoid deadlock: allowing a philosopher to pick both chopsticks at the same time (implemented in code using a critical section) or by introducing asymmetries (for ex. some philosophers take the right chopstick, while others the left one).

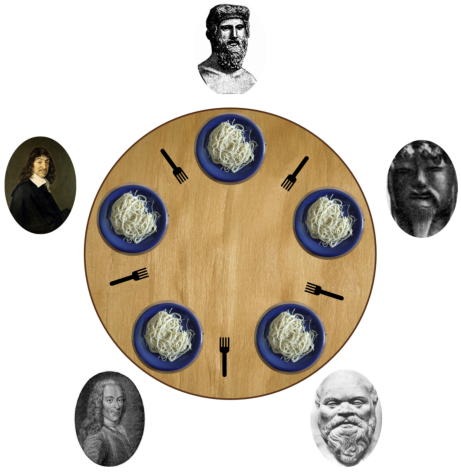


Figure 4.4: Philosophers' problem

4.6 Deadlock

Deadlock is a situation where two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.

A deadlock can happen if the following conditions hold simultaneously:

- Mutual exclusion: only one thread at a time can use the resource
- Hold and wait: a thread holding one resource is waiting to acquire additional resources held by other threads
- No preemption: a resource can be freed only voluntarily by the thread holding it
- Circular wait: in a set of threads $\{T_0, T_1, T_2, \ldots, T_n\}$, thread T_0 is waiting for T_1, T_1 is waiting for T_2, T_{n-1} is waiting for T_n and T_n is waiting for T_0

4.6.1 Resource allocation graph

Deadlocks can be represented using oriented graphs, called resource allocation graphs. Let the set of vertices V be of two types:

- $\{T_1, T_2, \ldots, T_n\}$ the set of threads
- $\{R_1, R_2, \ldots, R_n\}$ the set of resources

The request edges are denoted by $T_i \rightarrow R_j$ and the assignment edges are denoted by $R_j \rightarrow T_i$. Resources can have multiple instances, represented by dots on the graph.

A deadlock happens if there is a cycle in the graph and all resources involved have ony one instance. If there is a cycle and resources have several instances, then there might be a deadlock. If there are no cycles, then there is no deadlock.

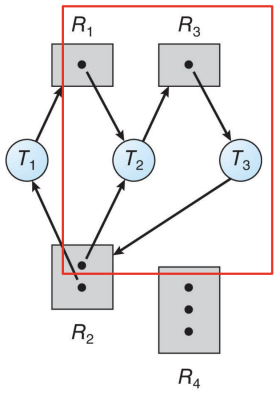


Figure 4.5: Resource allocation graph with deadlock

4.6.2 Deadlock prevention

Deadlock can be prevented by avoiding one of the necessary conditions for deadlock.

- Mutual exclusion: not feasible to work without it, must hold if there are non-sharable resources (such as when writing to files)
- Hold and wait: must guarantee that whenever a thread requests a resource, it does not hold any other resources. The cons of this solution are low resource utilization and possibility of starvation
- No preemption: if process cannot obtain all the resources it needs, it will free them all, go to sleep and be waken up where all the resources are available
- Circular wait (solutions implemented in OSs): to avoid circular wait an order can be given to the resources. If a process wants to access two or more resources, it need to first obtain access to the resource with the lower index

4.6.3 Deadlock avoidance

Deadlock can be avoided if the system knows which resources threads want to access. Then the system can execute an algorithm that detects possible deadlocks and prevents them before they happen. This technique is feasible in real-time operating systems, where tasks have to be allocated in advance for schedulability evaluation.

4.7 Thread-safe states

A state is safe if the system can allocate resources to the various processes in some particular order and avoid deadlock in doing so. More formally a state is safe if there exists a sequence of all threads in the system $\langle T_1, \ldots, T_n \rangle$, such that for each T_i, the resources that can be requested by the thread can be satisfied by the currently available resources or by the resources held by the threads T_j, where $j \leq i$. If a system is safe, no deadlocks can happen. An unsafe state is a state that may lead to deadlock. Not all unsafe states are actual deadlock.

There are two algorithms to check of safe states:

- Resource-allocation graph (only for single instance resources)

A claim edge $T_i \rightarrow R_j$ is defined on the resource-allocation graph (represented using a dashed arrow), which means that process T_i may request the resource. The claim edge is converted into a request edge when a thread requests the resource. The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph.

- Banker's algorithm (multiple resource instances)

Chapter 5

Memory management

The CPU can access only information located inside registers and main memory (RAM). Therefore processes to be run need to be copied disk into main memory. When information needs to be retrieved from main memory, many cycles are needed, causing a stall. To reduce the time wasted, a faster memory storage called cache is used. Cache is larger in size than CPU registers, but also slower.

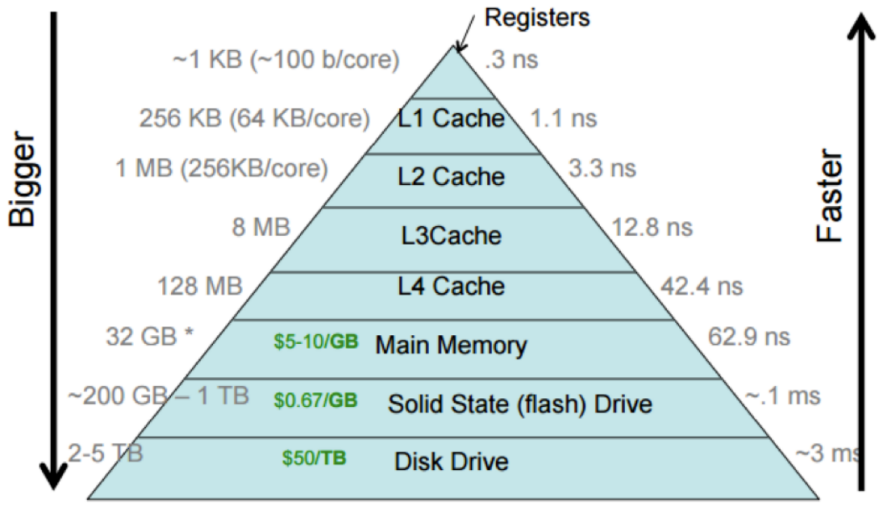


Figure 5.1: Memory hierarchy

5.1 Hardware access protection

The OS has to ensure that a process can only access addresses in its address space. This protection can be implemented using a pair of base and limit registers, which define the address space of the process. Any attempt in user mode to access memory out of bounds results in a trap. Changing relocation or limit registers are privileged instructions.

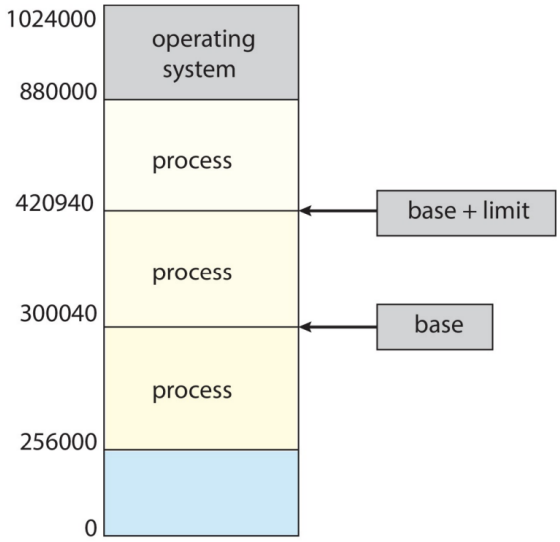


Figure 5.2: Base and limit registers

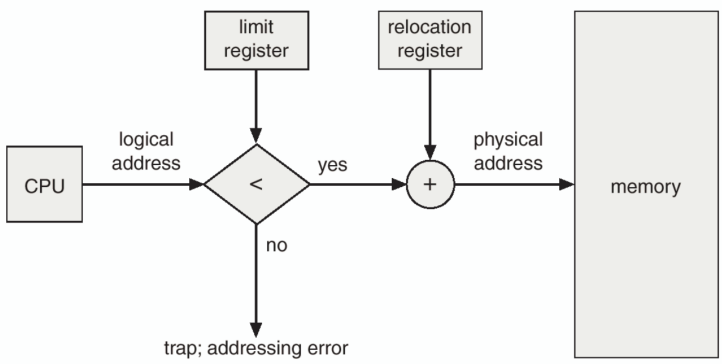


Figure 5.3: Hardware access protection

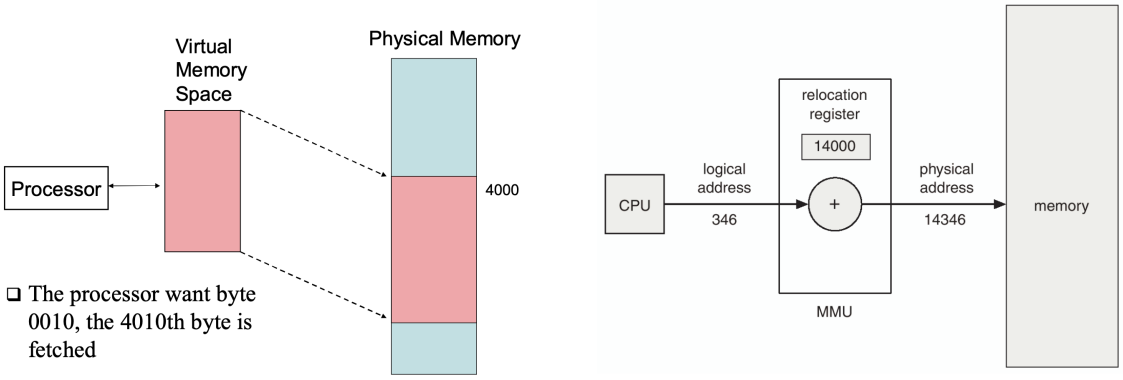
5.2 Address binding

Addresses are represented in different ways at different stages of a program's life. Addresses in source code are usually symbolic. A compiler typically binds these symbolic addresses to relocatable addresses (ex. "14 bytes from the beginning of this module"). The linker or loader will then bind relocatable addresses to absolute addresses. Address binding of instructions and data to memory addresses can happen at compile time, load time or execution time.

5.3 Logical and physical address space

The addresses used by the physical memory unit are called physical addresses. In primitive computing devices, the addresses used by programmers were the actual physical address, but modern memories are more complicated: for example, they are composed of multiple banks of memory chips. Therefore numbering and accessing a

specific position in memory is not straightforward. In advanced computer architectures, the processor uses a separate address space. Addresses belonging to this space are called logical or virtual addresses. The memory management unit (MMU) is used to map logical addresses to physical addresses.



(a) Memory view (b) Relocation register inside MMU

Figure 5.4: A simple MMU which offsets all virtual addresses by a constant value defined in the relocation register

5.4 Dynamic loading

Programs don't have to be fully copied to memory to be executed. With dynamic loading, routines (parts of a program) are not loaded until they are called. Routines are stored in memory in a relocatable format. No special support from the OS is required, dynamic loading is implemented by the programmer (although OS can help by providing libraries).

5.5 Static and dynamic linking

Programs can use external libraries in two ways: static or dynamic linking. In static linking, system libraries and program code is combined by the loader into the binary program image at compile-time. In dynamic linking, the linking phase of routines is postponed until execution time. Instead of the routine, a small piece of code, stub, is placed. When run, the stub replaces itself with the address of the routine at runtime. The operating system checks if the routine is in processes' address space and if not, it adds it. Dynamic linking is particularly useful for linking shared system libraries.

5.6 Contiguous Allocation

Main memory is usually divided into two partitions:

- Resident operating system, usually with low memory address
- User processes, with high memory addresses

When programs are loaded and unloaded from memory they create "holes". When a new request for allocation arrives, the OS has to decide where to place the new information. Three such strategies are:

- First fit: allocate in the first hole that is big enough
- Best-fit: allocate in the smallest hole that is big enough
- Worst-fit: allocate in the largest hole

The last two strategies require searching over whole memory, unless holes are ordered by size.

5.7 Fragmentation

This phenomenon of having non-contiguous data is called fragmentation. There are two types of fragmentation:

- External fragmentation: memory space is not contiguous
- Internal fragmentation: allocated memory is be slightly larger than requested memory

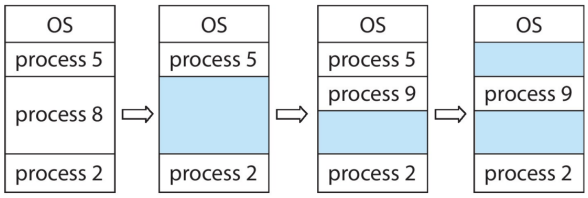


Figure 5.5: External fragmentation

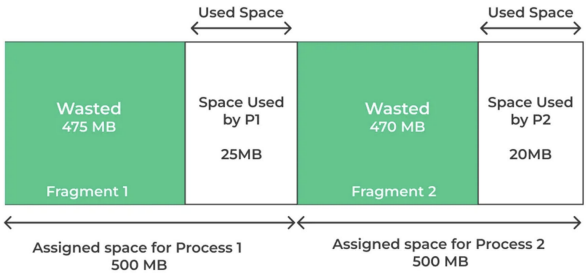


Figure 5.6: Internal fragmentation

External fragmentation can be reduced by compaction. Compaction is done by a tool that periodically shuffles the location of programs in memory and places them contiguously. This process reduces wasted space, but it is slow and very IO intensive.

5.8 Paging and TLB

The fragmentation problem can be solved by allowing programs to be split into multiple segments (this implies that the physical address space is not contiguous anymore). The program's logical memory is divided into blocks of same size called pages. Also physical

memory is divided into same sized blocks called frames. Then a table called *page table* is maintained to translate logical addresses to physical addresses.

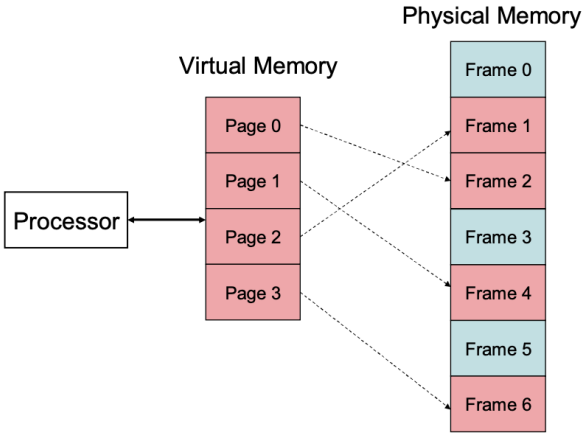


Figure 5.7: Paging

Each 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

The page size must not be too large, otherwise there will be a lot of space wasted due to internal fragmentation. Also, the page size must not be too small, otherwise there will be a lot of overhead due to the large number of pages. Typically, a page size of 4KB is used.

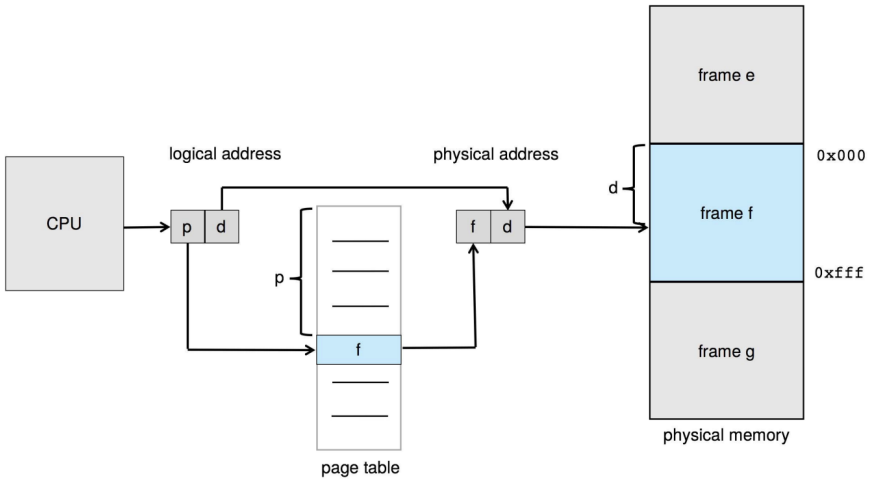


Figure 5.8: Paging hardware

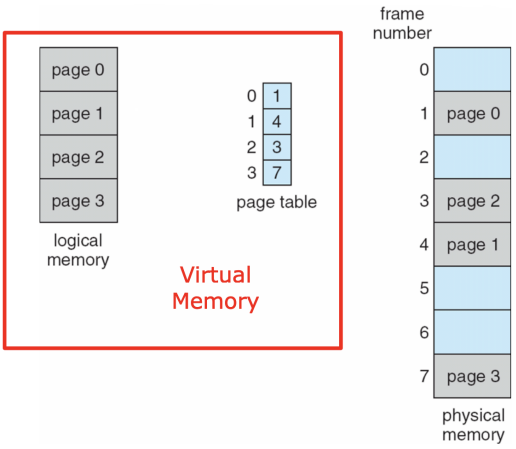


Figure 5.9: Paging example

The page table is kept in main memory. Two pointers are maintained:

- Page-table base register (PTBR) points to the page table
- Page-table length register (PRLR) indicates size of the page table

In this scheme every data/instruction access requires two memory accesses: one to the page table and one to the actual page. The lookup time can be improved by using a cache, which contains a table called translation look-aside buffer (TLB). Usage of cache is justified because it exploits temporal locality of programs (the tendency of programs to use data items over and again during the course of their execution). Each TLB entry stores also address-space identifiers (ASIDs), which identifies which process owns that entry for address-space protection. In this way the TLB doesn't have to be flushed at every context switch. On a TLB miss, the entry is loaded into the TLB for faster access next time. This means that appropriate replacement algorithms need to be considered.

The hit ratio is percentage of times that a page number is found in the TLB. The effective access time (EAT) for the TLB is given by

$$EAT = \text{hit ratio} \cdot \text{memory access time} + (1 - \text{hit ratio}) \cdot 2 \cdot \text{memory access time}$$

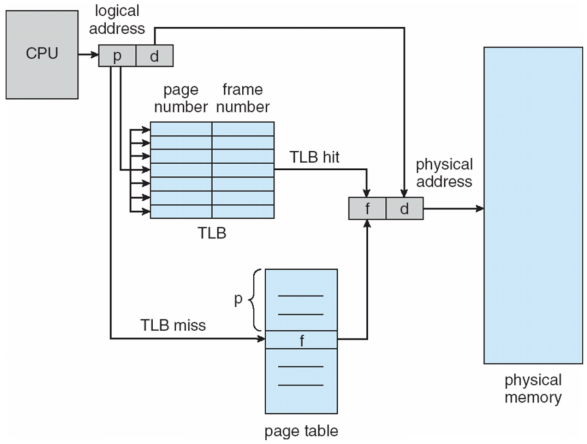


Figure 5.10: Paging with TLB

5.8.1 Memory protection

Memory protection is implemented by associating a protection bit with each frame to indicate if read-only or read-write access is allowed.

A valid/invalid bit attached to each entry in the page table: valid indicated that the associated page is in the process' logical address space, invalid that is not.

5.9 Page faults

Pages in the virtual memory space can be stored as frames in the physical memory or disks. Pages are brought into memory only when they are needed (dynamic loading). A **page fault** happens a page is not in main memory. To mark pages that are in RAM already a valid/invalid bit is used.

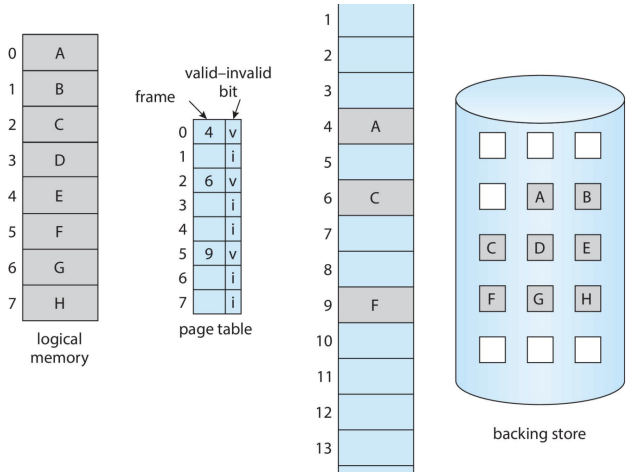


Figure 5.11: Virtual memory

The full procedure for retrieving a page from memory goes as follows:

1. Look up page table to see if valid bit is set. If not, a page fault happens and a trap is fired to the OS
2. If the reference is invalid, terminate the process. If it was valid but we have not yet brought in that page, we now page it in.
3. Find a free frame in memory
4. Swap page into frame via scheduled disk operation
5. Change page table to indicate page now in memory
6. Restart the instruction that caused the page fault

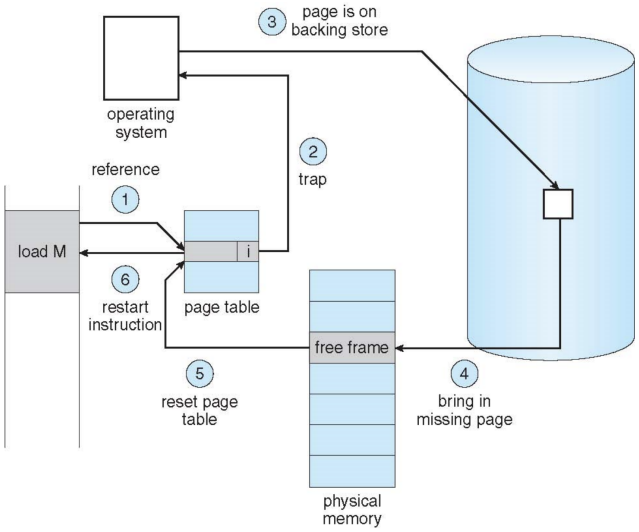


Figure 5.12: Page fault

5.9.1 Cost of a page fault

Let p be the probability of page faults (where $p = 0$ means there are no page faults and $p = 1$ means that every access is a page fault), then the effective access time is

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

Usually memory access time is around $200\mu s$ and page fault overhead is $8ms$.

5.9.2 Page replacement strategies

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. If there are no free frames left, a page will be removed of memory with a procedure called page replacement. The goal of a page replacement algorithm is to have the lowest amount of page faults. Therefore, the algorithm is evaluated by running it on a particular string of memory references (reference string) and computing the number of page faults on that string.

There are various algorithms for page replacements: FIFO, optimal, least recently used (LRU), most recently used (MRU), second chance.

First-in first-out

The first page that was brought into memory will be the one swapped first. This algorithm can be implemented by storing the arrival time of each page. This algorithm is not optimal and may exhibit Belady's Anomaly: more frames are supposed to produce less page faults, but this is not usually true with FIFO. For example, let be given the sequence of frames 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5. A memory that holds four frames will produce more faults than a memory with three frames.

Optimal replacement

If we know the sequence of the next pages that will be used, the page that won't be used for the longest time will be removed first. Of course it is impossible to know the future, so this algorithm is impossible to implement. Nonetheless it can be used as a benchmark for other algorithms, since it is the optimal solution.

Least recently used (LRU) algorithm

This algorithm exploits the principle of temporal locality. The operating system will replace the page that has not been used for the longest amount of time. This algorithm can be implemented by storing the current timestamp for each page when it gets accessed. The page with the lowest timestamp will be the one removed. This algorithm can be implemented also using a doubly linked list: every time a page is accessed it will be moved to the top of the list. The page at the bottom of the list will be the one removed.

LRU approximations

LRU is a slow algorithm because it requires searching, therefore there have been found some approximated LRU algorithms which are quicker.

Reference bit: each page has a reference bit, initially set to zero. When the page is referenced, the bit is set to one. When a page needs to be replaced, the pages with the bit set to zero will be chosen first.

Second chance algorithm: it is a FIFO algorithm, but each page has also the reference bit. If the page that has to be replaced according to FIFO has its reference bit is 1 it will be just set to zero and the algorithm will try to replace the next page in the FIFO queue (thus giving it a second chance).

Counting algorithms (LFU and MFU)

In counting algorithms the OS keeps a counter of the number of references that have been made to each page. Then, depending on the algorithm chosen (least frequently used or most frequently used), the page with the smallest count or the largest count will be removed.

5.10 Allocation of frames to processes

Each process needs a minimum number of frames to be able to work without too many page faults. There are various algorithms to allocate frames to processes, the simplest ones are equal allocation and proportional allocation. In equal allocation each process gets the same amount of frames. In proportional allocation, each process gets a number of frames proportional to its size. Both of these strategies don't account for high and low priority tasks: assigning more frames to one process is likely to make it go faster and finish earlier.

With multiple processes competing for frames, we can classify page-replacement algorithms into two broad categories: global replacement and local replacement. Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process. Local replacement requires that each process select from only its own set of allocated frames. Global replacement is more common because it gives greater throughput, but the single process execution time is more variable. Local replacement gives more consistent results, but memory may be underutilized, thus leading to smaller throughput.

A global replacement algorithm can be one that starts to remove pages when the number of free memory goes below a certain threshold (instead of waiting for the free frame list to go to zero). The kernel process that manages it is called reaper. This strategy attempts to ensure there is always sufficient free memory to satisfy new requests, i.e. that the free-frame list is never empty.

5.11 Thrashing

If a process does not have enough frames to perform to perform its operations, the page-fault rate becomes very high because the same pages get swapped over and over. This high paging activity is called thrashing and leads to low CPU utilization and might induce the OS to spawn more processes, which worsen the situation. The problem can be solved by monitoring the page-fault frequency for every process. If the frequency is low, the process will lose a frame, while if it is too high the process will gain one frame.

5.12 Swapping

A whole process can be swapped temporarily out of memory to a backing store, and then brought back into memory when execution is resumed.

Swapping can increase dramatically the time for the context switch. For example, if a 100MB process is swapped to hard disk with transfer rate of 50MB/sec, the swap out time of 2 seconds and additionally the swap in time for the new process is 2 more seconds. This huge time penalty is why swapping is not generally used in modern operating systems. OSes decide to swap only when free memory is extremely low. Pages of data allocated to a process that are not in physical memory are stored in the paging file.

5.13 Management of the page table

The page table can be very big for large memories. For example, if each page table addresses 4KB of memory and the memory is 4GB big ($\approx 2^32$), then the page table would contain one million entries. Given that each process may use any number of pages, if each entry is 4 bytes, then the page table for each process is 4 MB big. Since it is very unlikely that a single process would use the whole memory, the page table can be stored in more efficient data structures, at the tradeoff of speed.

5.13.1 Hierarchical page tables

The page table can be broken into multiple pieces, and a quick lookup table can be maintained to access the various pieces of the page table. The time complexity for accessing a page is $O(logn)$ (the time complexity of a simple page table is $O(1)$, because we know the exact address).

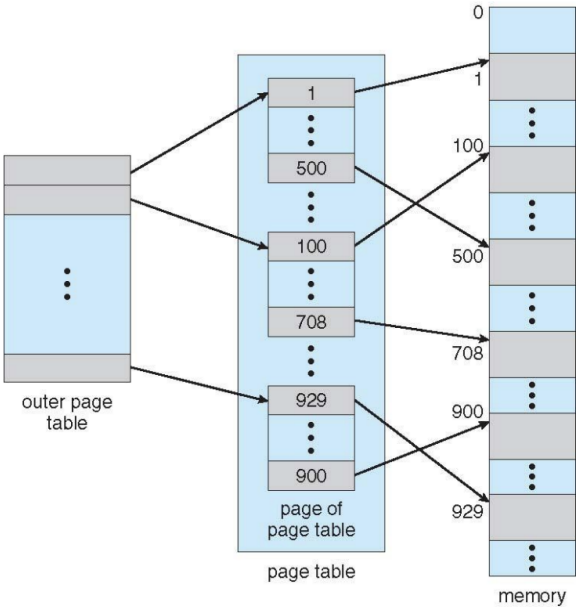


Figure 5.13: Hierarchical page table

5.13.2 Hashed page tables

Another possible data structure is the hash table. In case of collisions a linked list is maintained. In case of no collisions, the time complexity of this data structure is again $O(1)$.

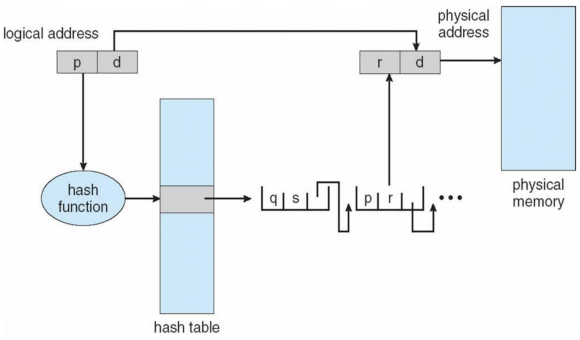


Figure 5.14: Hashed page table

5.13.3 Inverted page table

Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages in a single table. 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.

5.14 Segmentation

Using pages of fixed sizes leads to internal fragmentation. By using segmentation each process gets split in multiple segments, but the total size is equal to the size of the process. The segment table stores the information about all segments (base address and limit).

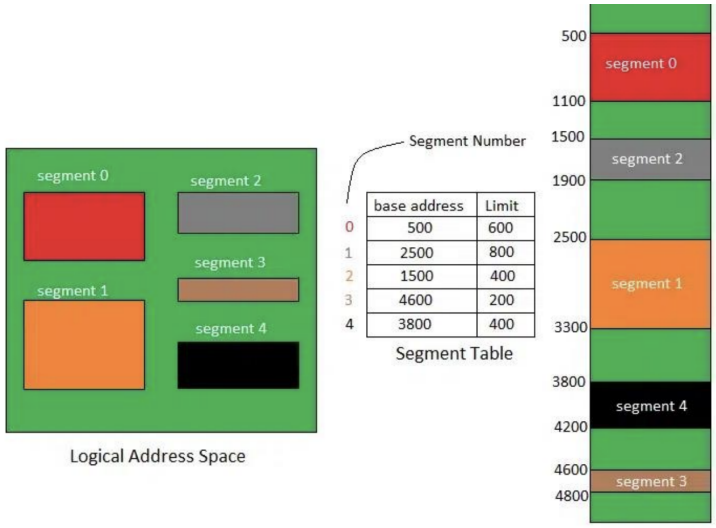


Figure 5.15: Segmentation

5.15 Mass storage systems

The most used devices for secondary storage in modern computers are hard disk drives (HDDs) and nonvolatile memory (NVM) devices.

HDDs are made of spin platters of magnetically-coated material under moving read-write heads. Data is organized in sectors located in concentric tracks called cylinders. The access time to access a random sector (or positioning time) is given by time to move the disk arm to the desired cylinder (seek time) and the time for desired sector to rotate under the disk head (rotational latency). A head crush happens if the disk head makes contact with the disk surface.

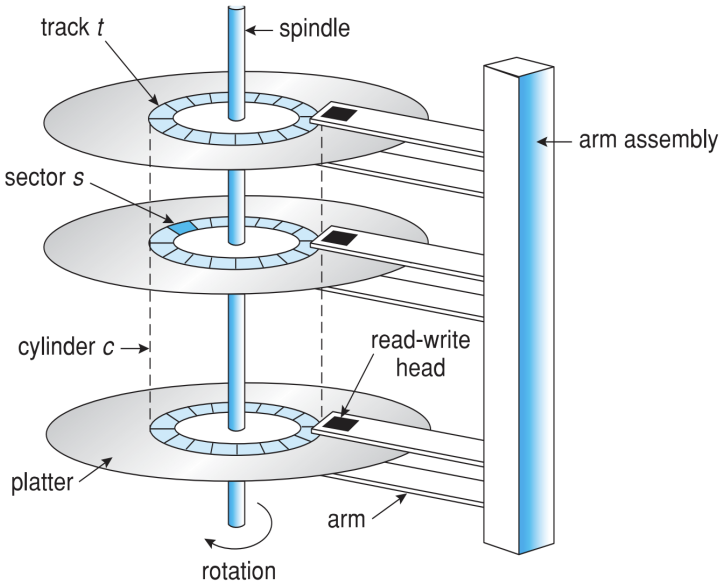


Figure 5.16: Hard drive mechanism

The alternative to hard disks are nonvolatile memory devices, such as solid-state disks (SSD) or USB drives. They do not have mechanical parts, therefore they are more reliable to vibrations and impacts and faster. They are more expensive and are more complicated to manage. For example, the information stored can be erased only in chucks. There is also a limited amount of erasures, before the drive warns out. This complexity is managed by a NAND flash controller, which is attached to the memory and regulates access to the memory.

5.15.1 Scheduling

The latency time and the seek time of HDDs can be reduced by implementing smart algorithms that reorder the read and write requests to minimize the movements of the arm and the rotations of the platters. Optimization can be done only when a queue of read and write operations exists. The basic algorithm, which does no optimization on the queue of requests, is the first-come first-served algorithm.

The SCAN algorithm (or elevator algorithm) moves the arm from one end of the disk to the other and back. While doing that, it services requests in the queue. The SCAN-C algorithm (C for circular) is an optimized version of SCAN, where instead of servicing requests also on the trip back it jumps immediately to the start again. In this way the average waiting time is uniform for all positions. SCAN and C-SCAN perform better when there is a lot of information needed to be read and written.

5.15.2 Storage device management

A disk is divided into different sectors. Each sector can hold header information, plus data, plus error correction code. The sector size can be changed by low-level formatting.

A disk can be partitioned into one or more groups of cylinders, each treated as a logical disk. For each partition the operating system can then manage however it wants,

usually using storing data in a data structure called filesystem. The operating system itself is stored in a boot partition, which is loaded into RAM at startup.

5.15.3 Network attached storage and cloud storage

Network-attached storage (NAS) is storage made available over a network rather than over a local channel. Usually it is implemented via remote procedure calls (RPCs) between host and storage over typically TCP or UDP on IP network. The user sees this storage as if it was a physical disk attached to the machine.

Cloud storage also provides access to storage across a network, but unlike NAS, data is retrieved using API calls at a software level.

5.15.4 RAID

RAID (redundant array of inexpensive disks) is a technique to offer redundancy of data and increase the mean time to failure and data loss. To achieve this it creates copies of the same data across multiple disk drives. It can also improve the read and write speed by using a technique called striping, where data is written and read in parallel from a group of disks. RAID is arranged into six levels:

- Mirroring or shadowing (RAID 1): each disk has a duplicate
- Striped mirrors (RAID 1+0) or mirrored stripes (RAID 0+1): provides high performance and high reliability
- Block interleaved parity (RAID 4, 5, 6): provide 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.

5.16 Error correction and detection

Error detection and correction is a fundamental aspect for storage devices and is often implemented in the storage device controller itself. Error detection determines if a problem has occurred. Two such algorithms are parity bit and cyclic redundancy check (CRC). Error correcting codes (ECC) not only detect, but can also correct some errors.

Chapter 6

Files and filesystem

A file system is a structure used by an operating system to organize and manage files on a storage device.

6.1 Files

Files can be of many types: text files, source files, executable files etc. Depending on the implementation of the operating system, each file has some attributes assigned to it, namely:

- Name: human-readable name
- Identifier: unique ID inside the filesystem
- Type: type of the file
- Location: pointer to the file location in memory
- Size: size of the file
- Protection: defines who can write/read/execute the file
- Other fields: time, date, user identification...

The file control block (FCB) contains all information related to a file (ownership, permissions etc).

Operations on files

A filesystem should support the following operations: create, write, read, seek, delete, truncate, open, close.

When a file is opened, the OS updates an "open file table", which tracks the files that are currently opened. The OS may offer the ability to lock a file, so that a process (exclusive lock) or a group of processes (shared lock) can earn the exclusive right to write to a file.

A file can be read sequentially or by direct access. More advanced methods involve the creating of an index of the file for a faster lookup. Sequential operations supports only the read/write next and reset operations, while direct access supports reading and writing to a specific position in the file.

6.1.1 Memory-mapped files

Similar to processes, also files can be loaded in smaller chunks into memory. To achieve this, the files are mapped to an address range within a process's virtual address space, and then paged in as needed using the ordinary demand paging system. File writes are done to the memory page frames, and are not immediately written out to disk.

6.1.2 Directories

Directories are a collection of nodes which point to files. Directories can be structured in different ways:

- Single-level: single directory for all users
- Two-level: separate directory for each user
- Tree-structured: directories and files organized in a tree structure; general structure, scalable, easy to search, no file sharing support, possible file duplication
- Acyclic-graph: tree structure with files and folders that point to other files (aliases), no information duplication

In acyclic graph structures, when a file is deleted, all other aliases need to be deleted too, therefore the filesystem maintains a list backpointer to the aliases. Another problem that has to be avoided is creating cycles (ex. alias file that points to another alias file that points to it). This check can be done when a link is created, by using a cycle detection algorithm.

6.1.3 Protection

Filesystems should implement a protection mechanism, so that the file owner/creator should be able to control what can be done by other users. For example, the owner may restrict the following modes of access: read, write, execute, append, delete, list. In Unix there are three modes of access: read, write, execute. Moreover, there are three classes of users: owner, group, others.

6.2 Filesystem

A filesystem should manage files, access control, synchronization and protect, provide a high level interface for programs to use and interface with the physical hardware.

Filesystems can be divided into general-purpose filesystems and specialized filesystems. General-purpose filesystems are the ones generally used to hold user data, such as files, directories, programs etc. General purpose filesystems are used by the operating system to help it implement special tasks, such as IO management and daemon management.

6.3 Volumes

The entity containing file system is known as a volume. Each volume contains a volume control block (also known as superblock or master file table), which contains volume details, such as: total number of blocks, number of free blocks, block size and free block pointers or array.

6.4 File system implementation

A filesystem provides the user an interface to the physical storage, creating a mapping between the logical layer to the physical hardware. A filesystems is usually implemented in layers.

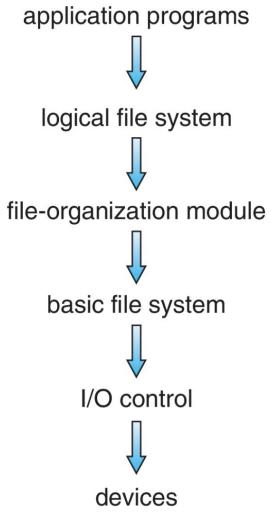


Figure 6.1: Layered filesystem

6.4.1 Filesystem layers

The device driver controls the devices by using commands. An example command is read drive1, cylinder 72, track 2, sector 10, into memory location 1060. It outputs low-level hardware specific commands to hardware controller.

The basic file system translates command like "retrieve block 123" to a command that can be interpreted by the device driver. It also manages memory buffers and caches (allocation, freeing, replacement). The file organization module translates logical block numbers to physical block numbers and manages free space and disk allocation.

The logical file system manages metadata information about files (stored in file control block), directory management and protection. The FCB is called inode (index node) in Linux.

The Linux inode contains the following information:

- file type (regular file/directory/symbolic link/block special file/character special file/etc)
- permissions

- owner/group id
- size
- last accessed/modified time
- number of hard links
- memory position of the blocks composing the file

In Linux the items of the table are called file descriptors.

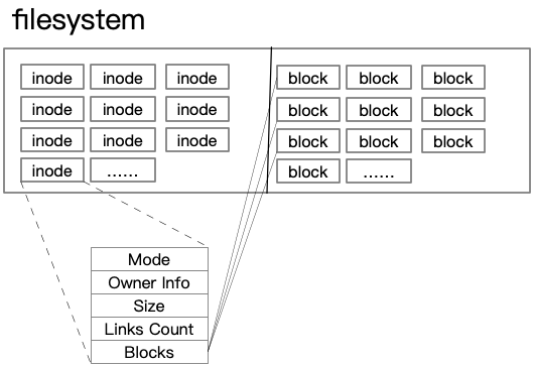


Figure 6.2: Linux inode

This is the layer to which programs talk to, using systems calls such as `open()`.

6.4.2 Directory implementation

A simple implementation of directories is using a linked list. This implementation is simple to program, but it is slow, since it has a linear search time. The lookup can be sped up by using a hash table to quickly find the position in memory.

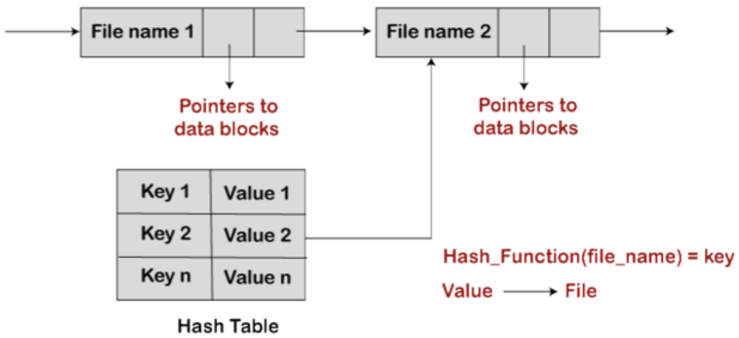


Figure 6.3: Directory implementation using list and hash table

6.4.3 File allocation method

The allocation method refers to how disk blocks are allocated for files: contiguous allocation, linked allocation, allocation using a File Allocation Table (FAT), indexed allocation method.

Contiguous allocation method

In contiguous allocation each file occupies a set of contiguous blocks. This method is simple (requires knowing only the size of the file) and has the best performance in most cases. The downside is that is prone to external fragmentation, thus requires frequent compaction (defragmentation).

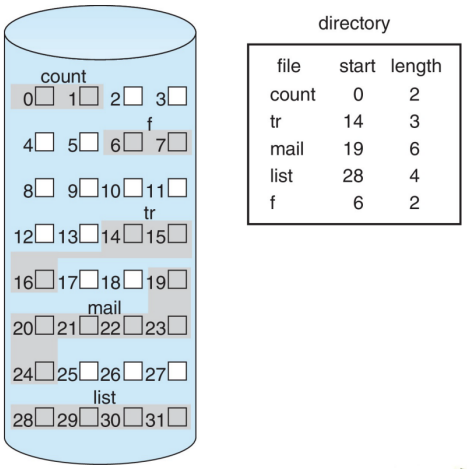


Figure 6.4: Contiguous allocation

Linked allocation

In linked allocation each file is a linked list of blocks. The file ends at NULL pointer. This method does not have the problem of external fragmentation. The big problem of this allocation method is that locating a block takes many I/Os and disk seeks.

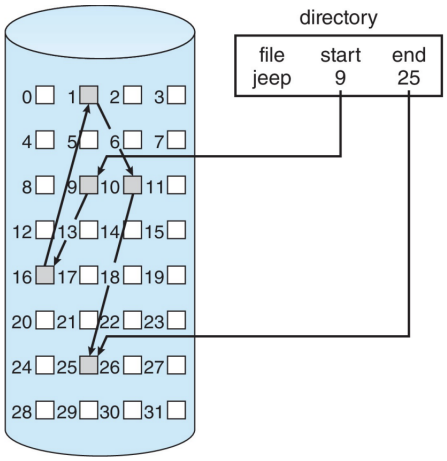


Figure 6.5: Linked allocation

FAT allocation method

In FAT allocation volumes contain a file allocation table, which holds the position in memory of each block, instead of storing it in the blocks as in liked allocation. This greatly increases the speed of searching files. This allocation method still uses linked allocation for files, thus retaining also its advantages.

Indexed allocation method

In this allocation method each file has its own table of pointers to its data blocks.

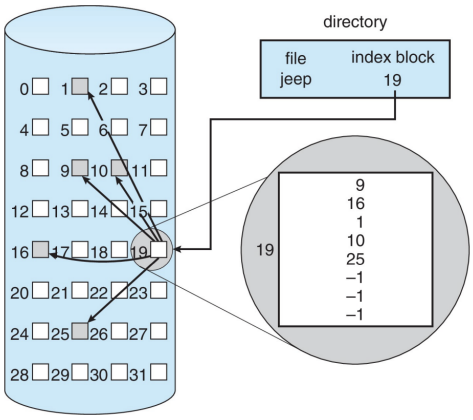


Figure 6.6: Indexed allocation method

Performance

The best file allocation method depends on how the files are being accessed. Linked allocation is good for sequential, but not for random access. Indexed allocation is more complex, because it requires an index block read then data block read.

6.4.4 Free space management

Free space management is needed, so the OS can quickly find a free space when it has to write something to disk. The easiest method is that the filesystem maintains a list of available blocks (bit vector). The downside is that this table consumes disk space. An alternative would be to store in each free memory location the pointer to the next free position (linked list). The upside is that this method does not waste any space, but obtaining a contiguous space may require a long search. This can be improved by modifying the linked list to store the address of the next $n - 1$ free blocks in first free block, plus a pointer to next block that contains free-block pointers (grouping). Another method is to keep a table with each first free block and the count of the following free blocks (counting).

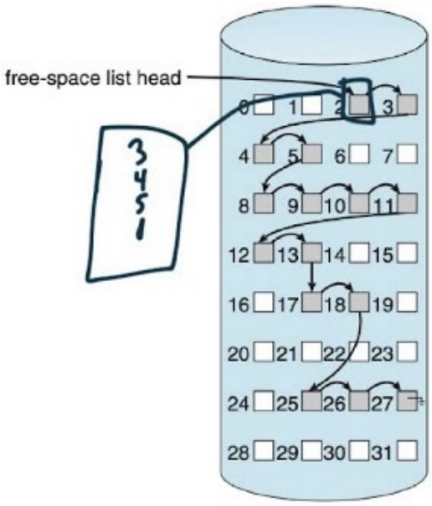


Figure 6.7: Improved free space linked list

6.5 Remote file systems

Distributed file systems allow the users to use the filesystem as normal, but the files and folders are actually stored on a remote machine. They are normally implemented based on the client-server model: the client asks the server to perform the various operations on files (read, write, delete, update et).

6.6 Virtual file systems

A virtual file system is an abstract layer on top of a more concrete file system. A VFS specifies the interface (API) between the kernel and a concrete file system. In this way the kernel can use different filesystems (such as remote filesystems) in the same way. The virtual equivalent of the Linux inode is the vnode.

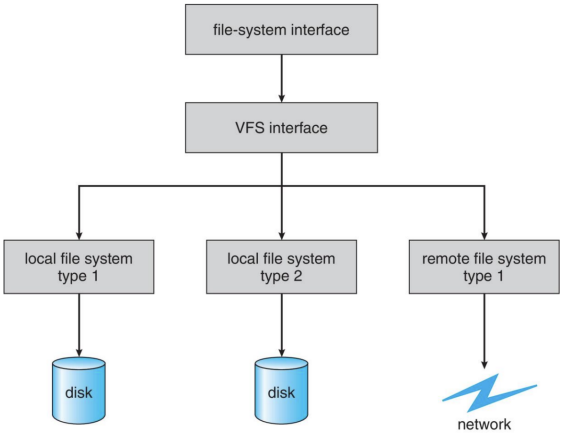


Figure 6.8: Virtual filesystem

Chapter 7

Other topics

7.1 Virtualization

The idea of virtualization is to put a layer between the implementation of something ad its interface. An application of virtualization are virtual file systems. The same idea can be applied to virtualize access to hardware: the software in charge of this is called virtual machine manager (VMM) or hypervisor. The OS that interfaces directly to the hardware is called host, while the ones running through a VMM are called guests. There are different types of VMMs:

- Type 0 VMM: the VMM is implemented directly in hardware
- Type 1 VMM: the VMM is an OS-like program which allows execution of virtual machines and it runs at boot
- Type 2 VMM: the VMM is run as a normal user process on a real operating system. This solution is the one that gives the less freedom, since the host OS regulates the access to resources, the processor used, the amount of CPU time etc.

There are other variations of VMMs, namely:

- Paravirtualization: the guest operating system is optimized to work well with the VMM
- Emulators: allow applications written for one hardware environment to run on a very different hardware environment
- Application containment: not actual virtualization, provides virtualization-like features by segregating applications from the operating system, making them more secure, manageable

Running Mode

The VMM is run in user mode on the host OS, but the guest OS need s to run some software in kernel mode. This means that the VMM needs to virtualize the kernel mode, which is known as trap-and-emulate.

7.1.1 Containers

Containers are a form of operating system virtualization. Inside a container are all the necessary executables, binary code, libraries, and configuration files to execute a software. The main difference with virtual machines is that they do not contain an OS. They still require a translation layer to interface with the host operating system, such as the Docker engine.

7.1.2 Other applications of virtualization

Another application of the virtualization idea is the programming environment virtualization. In this case the programming language is designed to run within a custom-built virtualized environment. For example Java has many features that depend on running in Java Virtual Machine (JVM).

7.2 Security

A system is said to be secure if resources are accesses and used as intended under all circumstances. **Intruders** attempt to breach security. A **threat** is a potential security violation. An **attack** is an attempt to brach security, which can be accidental or malicious.

There are different types of security violations: 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).

There are different types of attacks: ransomware, replay attack, MITM attack, session hijacking, privilege escalation, trojan, backdoor, malware, spyware.

Security measures against attacks can be implemented at all layers of the system to be effective (physical, application, OS or network layer). Security is as weak as the weakest link in the chain.

Cryptography can be used as a tool to improve security (ex. authentication and communications). It can be implemented at various layers of the ISO reference model (network layer - IPSec, transport layer - SSL aka TLS).

Firewalls are software or hardware solutions that limit network access between two security domains.

7.2.1 Principles of protection

A very important principle of protection is the principle of least privilege. This principle states that 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 or gets abused. Privileges can be set statically or dynamically. Rough-grained privilege management is simpler, but less effective. Fine-grained management is more complex and adds more overhead, but is more protective. Additionally, an audit log can be maintained to record all sensitive activities.

7.2.2 Protection methods

Protection rings can be established: components are ordered by amount of privilege and are protected from each other. For example, the kernel is in one ring and user applications in another. This privilege separation requires hardware support.

Other protection methods are sandboxing (running processes in a limited environment) and code signing (verifying that the software is genuine by checking its signature).

7.3 I/O systems

I/O communication

The communication with I/O devices can be done in two ways:

- Polling: the device sets a busy bit when it is communicating, the host periodically checks it waits until the device is not busy (host is busy waiting) and sets a command-ready bit when it wants to receive data.
- Interrupts: I/O device can trigger an interrupt line, which is connected directly to the CPU. The line is checked by the processor after every instruction and each interrupt is managed by the interrupt handler. The interrupt handler can ignore (maskable interrupts) or give priority to some interrupts. The interrupt vector to dispatch the interrupt to correct handler. The interrupt mechanism is also used for software exceptions and page faults (traps).

7.3.1 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). 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.