Operating systems

Salvatore Andaloro

April 3, 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 to be:

- User side: friendly, reliable, safe, fast
- System side: easy to design, modular, error-free, flexible, efficient

An operating system is composed of a kernel, system programs and user 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 usually called **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 the following: there are one or more CPUs and device controllers that are connected through a common bus with a shared memory.

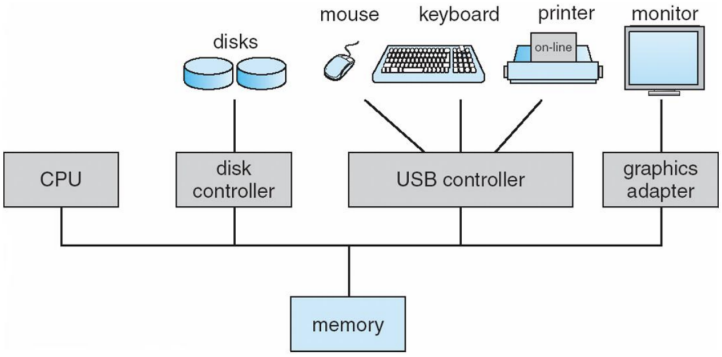

Figure 1.1: General computer architecture

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.

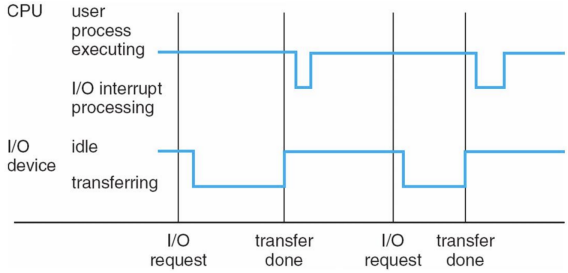

Figure 1.2: Interrupt timing diagram

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. 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 systems 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 and/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

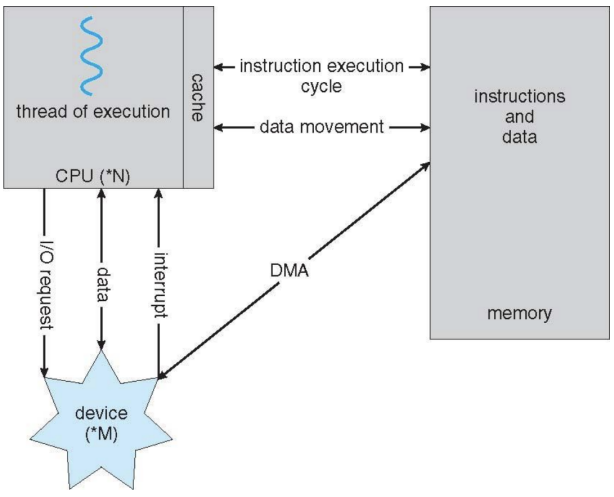

Figure 1.3: A Von-Neumann architecture

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 because they usually consume less power than multiprocessor systems and because on-chip buses are faster.

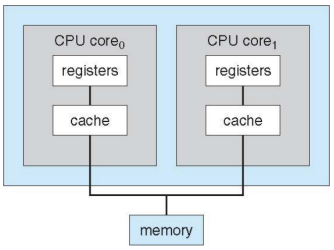

Figure 1.4: A multi-core processor

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 other 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.

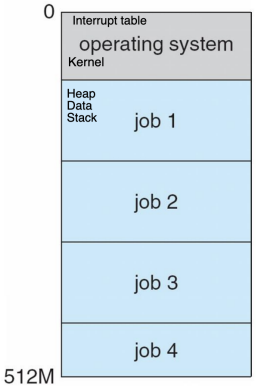

Figure 1.5: Memory layout for multiprogrammed systems

The OS and users share the same hardware, devices and software resources. To protect the system and avoid that different jobs can write in some areas of the memory a privilege system is established. In dual-mode systems jobs can be run in user mode or

kernel mode. Some instructions are allowed only for kernel mode systems. For example Intel processors have for 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 fo the program is reached. Multi-threaded process has 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 usually is 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
- Client-server computing
- Peer-to-peer computing
- Distributed computing: many systems connected together over a network
- Virtualization: guest OS emulates another OS or hardware and runs software on it. The program that manages this is called VMM (Virtual machine manager).

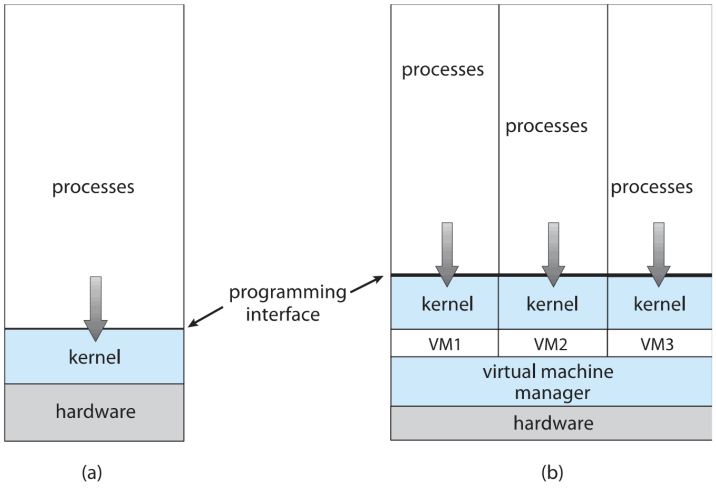

Figure 1.6: Virtualization

1.5 Services provided by operating systems

Operating systems provide the following services:

- User interface: can be command-line (CLI), Graphics User Interface (GUI), Batch
- 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 and security

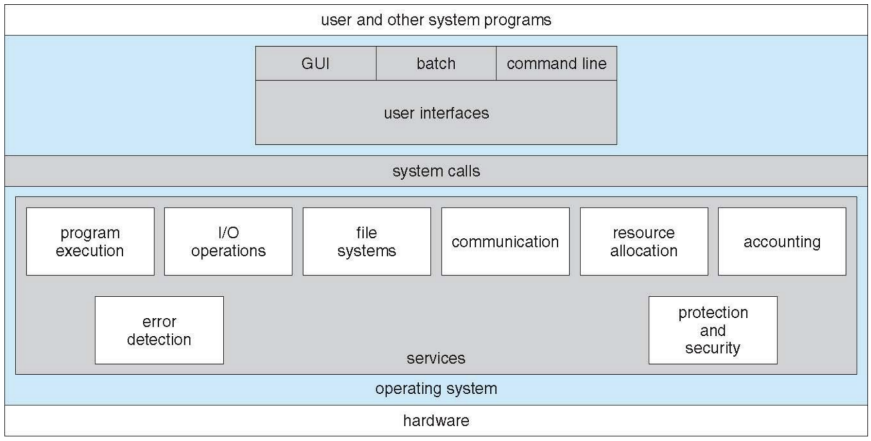

Figure 1.7: Services provided by an operating system

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. In this way developers can use a single API that works on all operating systems and leave the actual system call to the underlying library written for that specific platform. The high-level API can also check for errors before calling the system call.

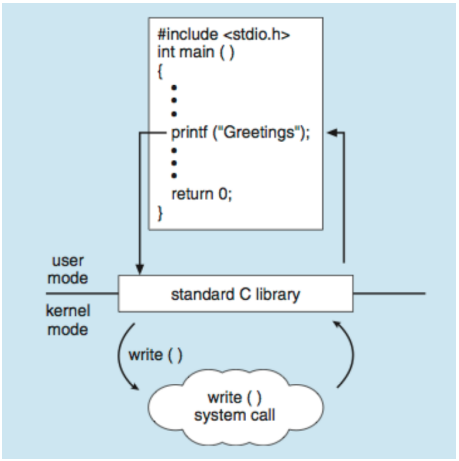

Figure 1.8: The `printf()` function in C uses the `write()` system call to print to the screen

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 using predefined specific

registers. Often there are not enough registers for all required parameters, so parameters can be also stored in memory in a table and just the address of the table is put into the register.

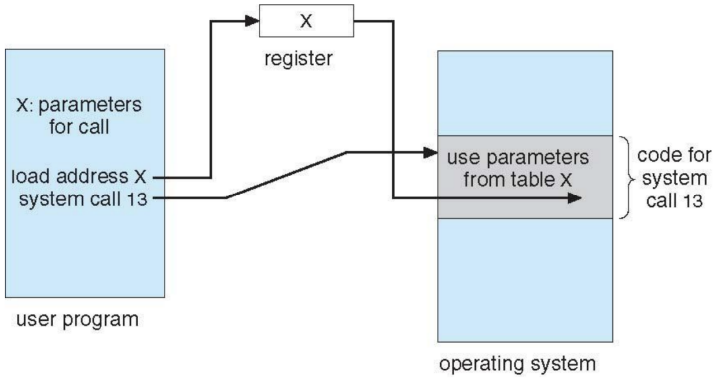

Figure 1.9: Parameter passing

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 OS structure

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

1.7.1 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.

1.7.2 Monolithic kernel - UNIX

Originally UNIX had a monolithic structure. The kernel provided a large number of functions, such as the file system, CPU scheduling, memory management. 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.

1.7.3 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.

1.7.4 Microkernel

The microkernel approach moves processes as much as possible outside the kernel into the user space. Communication between modules is achieved using message passing. 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). Message passing introduces additional overhead, thus has a negative performance impact.

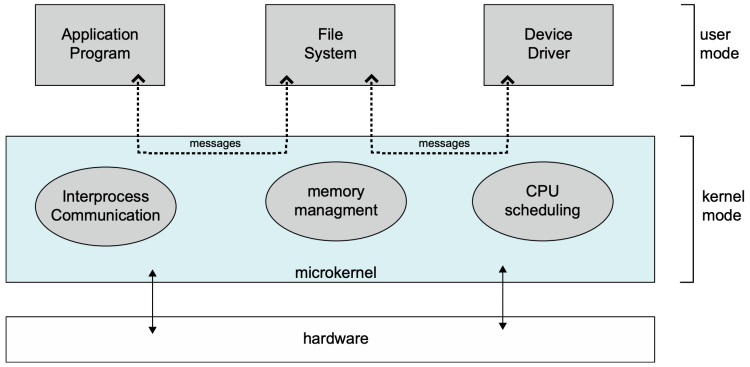

Figure 1.10: Microkernel structure

Chapter 2

Processes

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

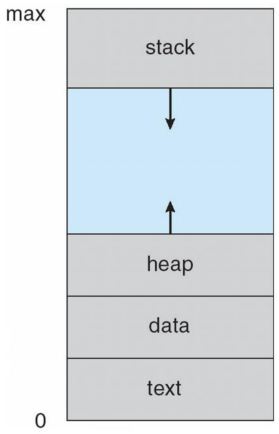

Figure 2.1: Memory layout for a process

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

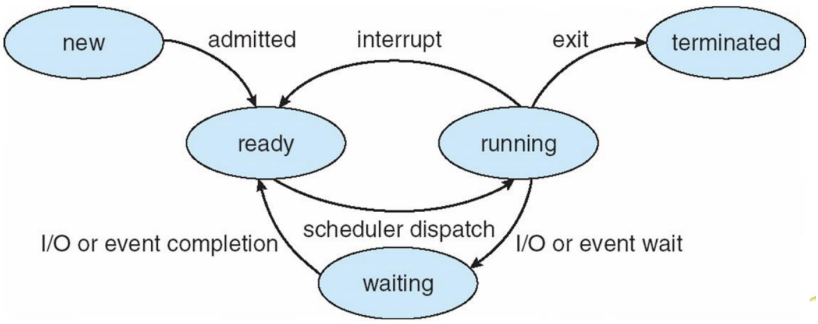

Figure 2.2: Process state machine

The OS also keeps track of the state of the process is stored in the RAM in a data structure 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 a process in the PCB, so that it can be restored and resume execution at a later point. 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

2.1 Scheduling

The CPU has a process scheduler, which decides which process to execute. 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

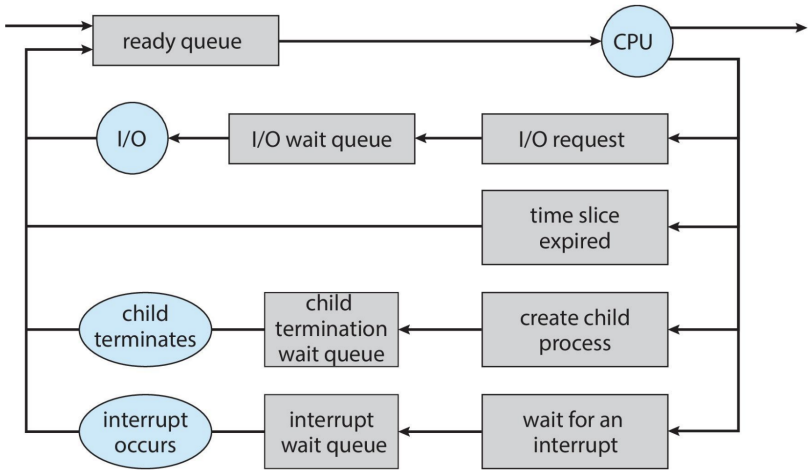

Figure 2.3: Process queues

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

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 process 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

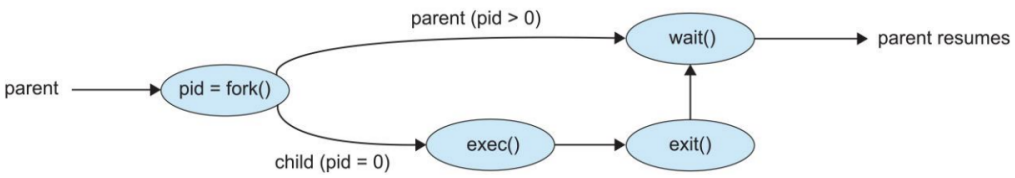

Figure 2.4: Creation of children processes

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

```c
#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

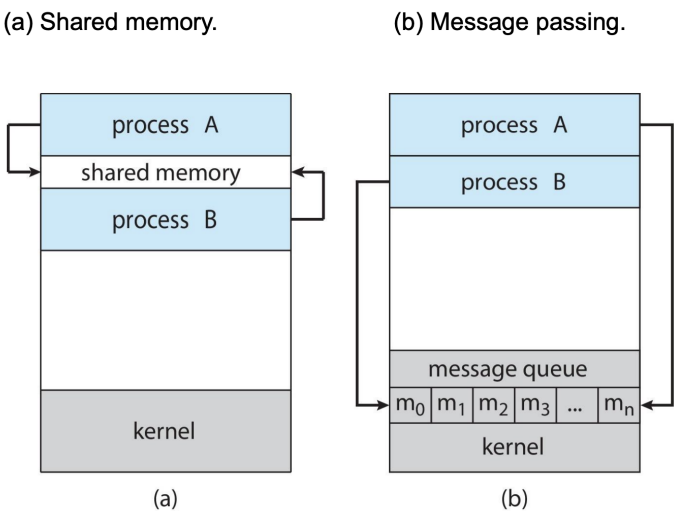

Figure 2.5: Models of communication between processes

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]. 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>

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;
  }
```

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

```
  /**
   * 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

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#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. This can be implemented in the following ways:

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

We can distinguish the channel on a logical level in the following ways:

- Direct or indirect
- Synchronous or asynchronous
- Automatic or explicit buffering

Pipes

Pipes provide a way for to processes to communicate directly with each other. Pipes are accessed using the file descriptor. We can distinguish among two different types of pipes: ordinary pipes and named pipes.

Ordinary pipes 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, meaning that the parent process can only write to it and the child process can only read from it.

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

#define BUFFER_SIZE 25
```

```c
#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

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

```c
#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 registers, stack and program counter.

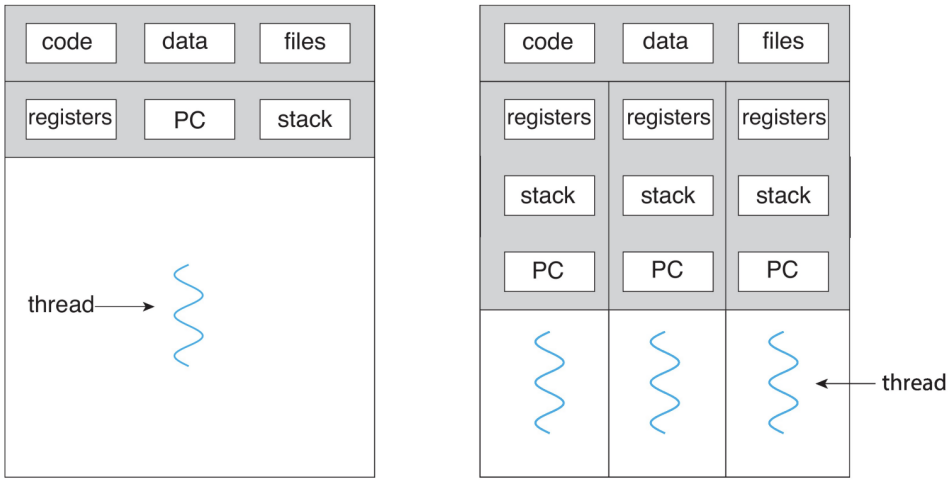

Figure 2.6: Single threaded process (left) and multithreaded process (right)

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: resource sharing is easier than memory/message passing, they are quicker to create 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. 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
- Task parallelism: tasks are distributed across multiple cores, each thread performs a different operation

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 serial code, 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.

Chapter 3

CPU Scheduling

The system executes tasks in cycles made of CPU bursts and IO 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.

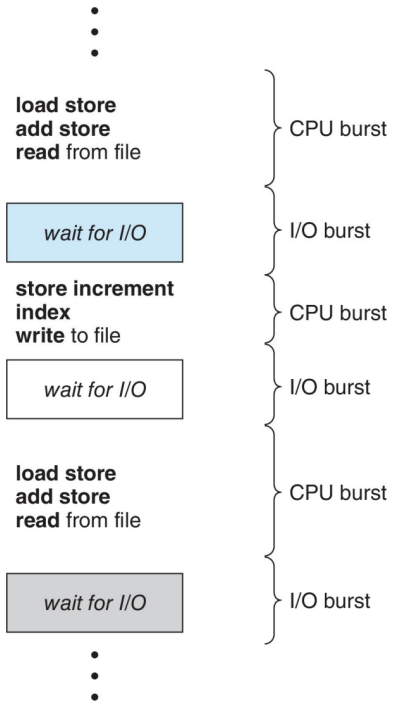

Figure 3.1: CPU bursts and IO bursts

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.

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. Note that q must be large with respect to context switch, otherwise the overhead is too high. 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.

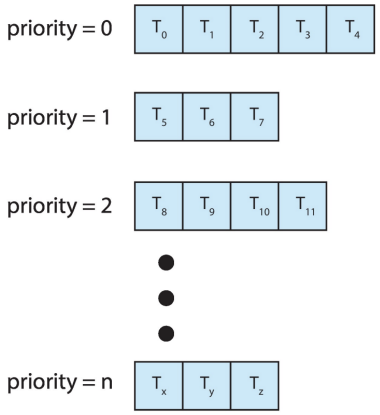

Figure 3.2: Multilevels queues

3.3.6 Multiple-processor scheduling

Schedulers for multiple-processor systems can have a common queue for all processors or a queue for each processor.

When a thread runs on a processor, it uses the cache of a processor (processor affinity). If then that thread gets executed on another processor, it will lose all the cached content.

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 (soft real-time). Also tasks have a new characteristic to consider: periodic tasks require the CPU at constant intervals.

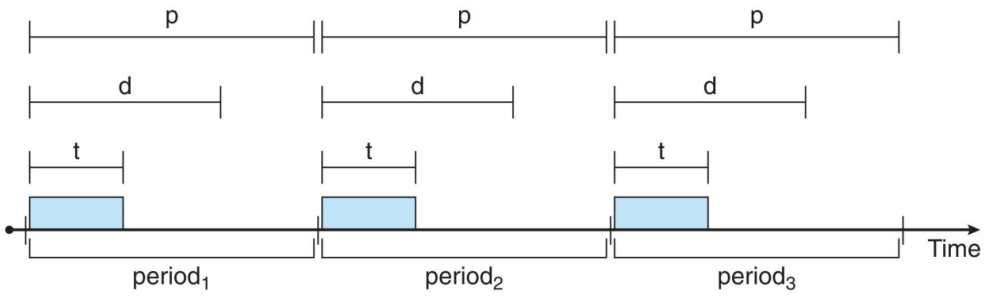

Figure 3.3: Periodic task with processing time t, deadline d and period p

Rate monotonic scheduling

A priority is assigned based on the inverse of its period. 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.

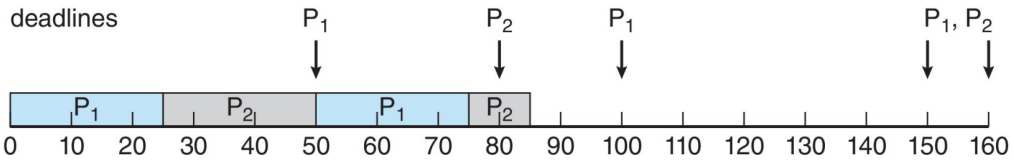

Figure 3.4: Task P2 misses the deadline because task P1 always has the highest priority

Earliest Deadline First Scheduling

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

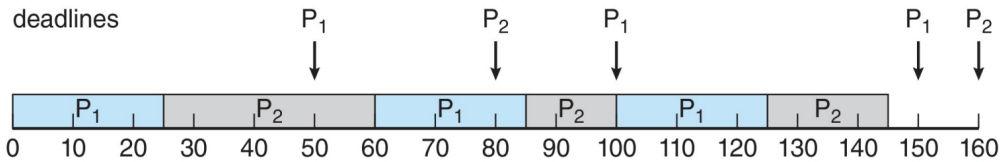

Figure 3.5: Earliest Deadline First Scheduling

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.

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.

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

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.

4.1 Critical section

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 (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 n 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

Many systems provide hardware support for implementing the critical section code. Special hardware instructions that allow us to either test-and-modify the content of a word, or to swap the contents of two words atomically.

4.2.1 Test and set instruction

To access a critical section processes have to first call the `test_and_set()` instruction. This instruction must be non-interruptible (i.e. it has to be executed atomically).

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

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 (for example it must be non-interruptible). 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.

The `lock` variable from the previous example is an example of an **atomic variable**, i.e. a variable whose value changes atomically (non-interruptible).

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 (this lock therefore called a spinlock), which is undesirable.

4.3.2 Semaphore

Semaphores are similar to semaphores, but can be also 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.

4.3.3 Semaphore without busy wait

Semaphores can be used to solve the busy waiting problem. This can be done by making the variable `S` a queue of processes that wish to execute their critical section. Processes are added to the queue using the `block()` atomic function. Now the `wait()` function will just call the `block()` function, thus avoiding busy waiting. The next process in the queue is executed by calling the `wakeup()` function. Therefore the `signal()` function will call the `wakeup()` function.

4.3.4 Monitors

Monitors are structures that abstracts synchronization and exposes 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.

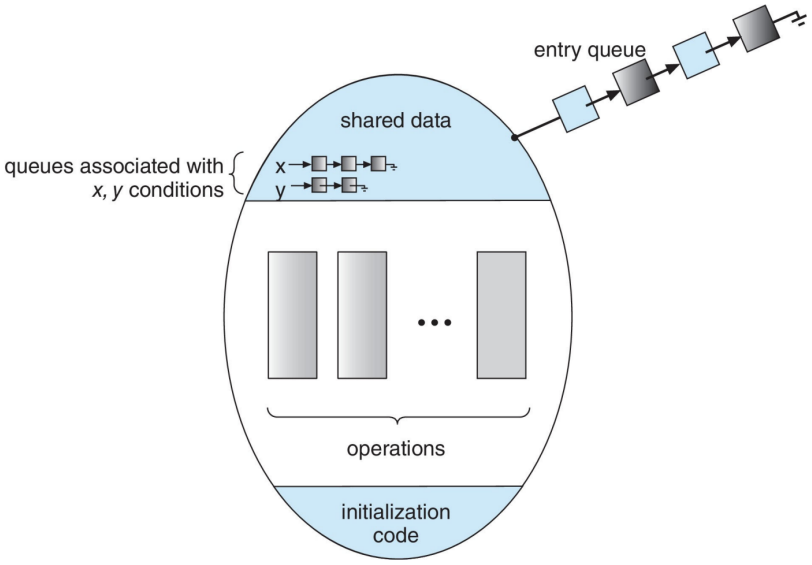

Figure 4.1: Monitor with condition variables

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 with different priorities. 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 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.

Deadlock happens in the following scenarios:

- When two processes are waiting for each other
- Starvation: a process may never be removed from the semaphore queue in which it is suspended
- Priority inversion: scheduling problem when lower-priority process holds a lock needed by higher-priority processes. The solution to this problem is that 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:

- Bounded buffer problem
- Readers-writers 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.

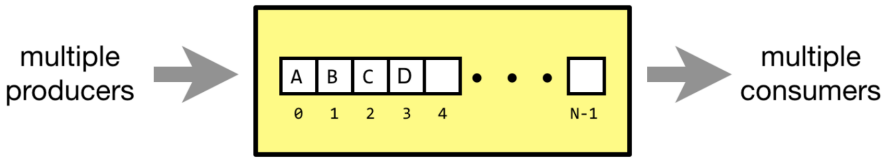

Figure 4.2: Bounded buffer with capacity n

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

```
full <- number of slots in buffer
empty <- 0

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

// Consumer
while (true) {
    wait(empty); // If empty = 0 wait, otherwise decrement empty by 1
    ↪  and continue
    lock(mutex); // If someone is reading/writing wait, otherwise
    ↪  continue
    consume();
    unlock(mutex); // Release access of buffer
    signal(full); // Increment full 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.

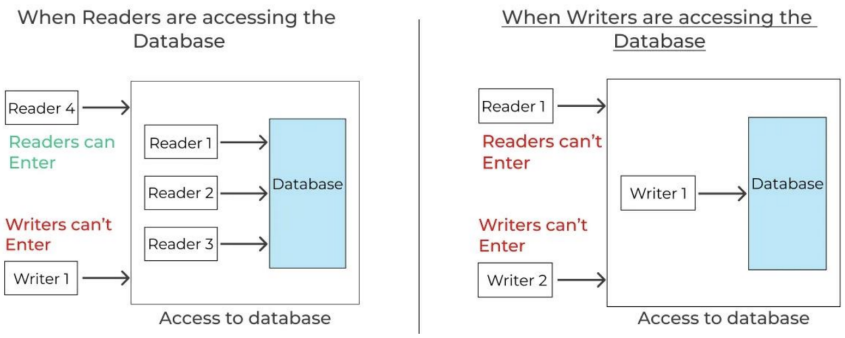

Figure 4.3: Readers-writers problem

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

// Writer
while (true) {
    lock(rw_mutex);
    write();
    unlock(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) {
        lock(rw_mutex);
        unlock(mutex);
    }
    read()
    lock(mutex);
    read_count--;
    // Check if I'm the last reader and unlock r/w access
    if (read_count == 0) {
        unlock(rw_mutex);
        unlock(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 a chopstick at the same time, then they will wait indefinitely for the other one and no one will eat, thus creating a deadlock.

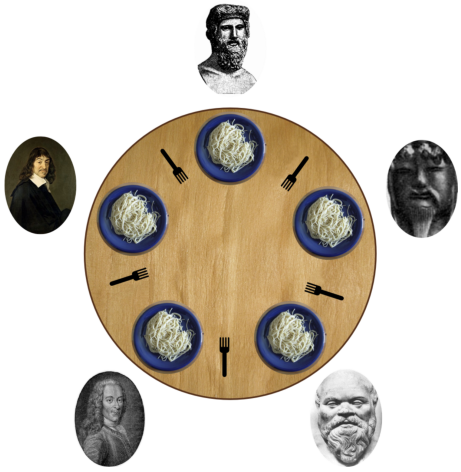

Figure 4.4: Philosophers' problem