Operating systems

Salvatore Andaloro

March 7, 2024

Contents

Chapter 1

Introduction

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

- 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 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.

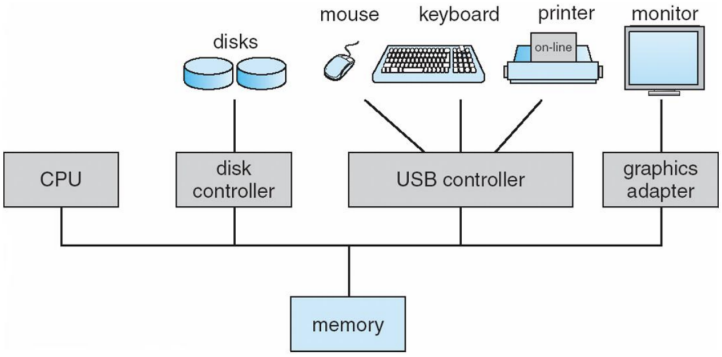

Figure 1.1: General computer architecture

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.

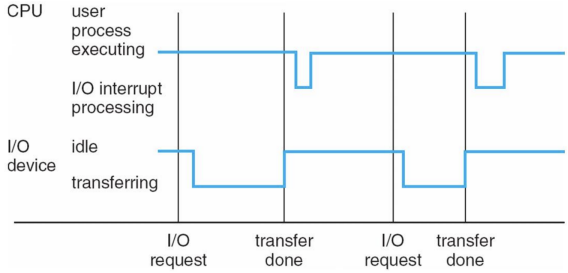

Figure 1.2: Interrupt timing diagram

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

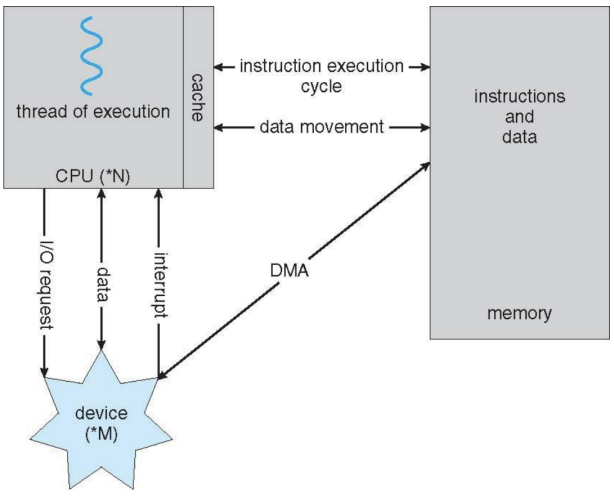

Figure 1.3: A Von-Neumann architecture

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.

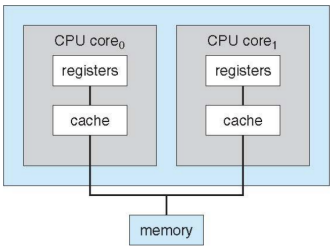

Figure 1.4: A multi-core processor

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.

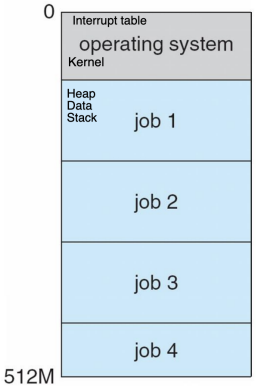

Figure 1.5: Memory layout for multiprogrammed systems

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).

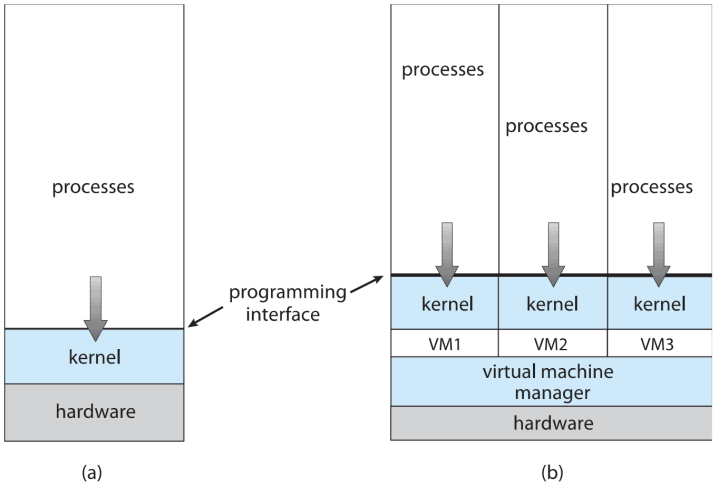

Figure 1.6: Virtualization

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

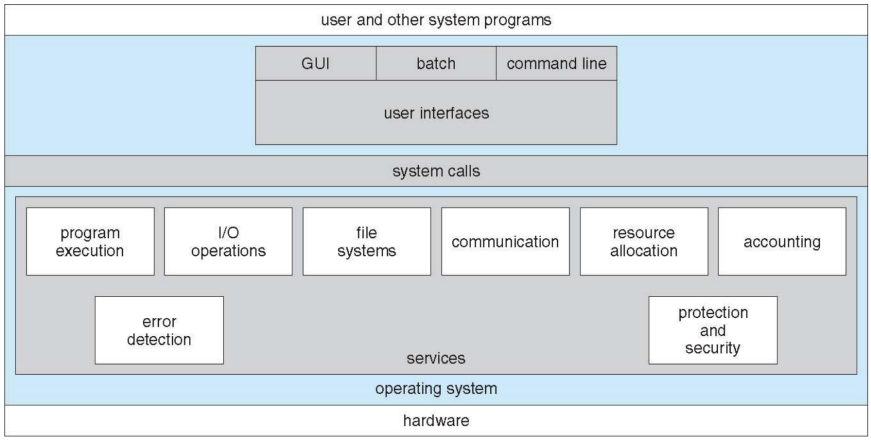

Figure 1.7: Services provided by an operating system

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.

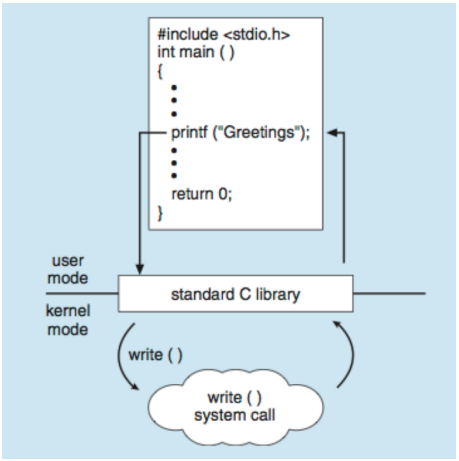

Figure 1.8: The `printf()` function in C uses the `write()` system call to print to the screen

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.

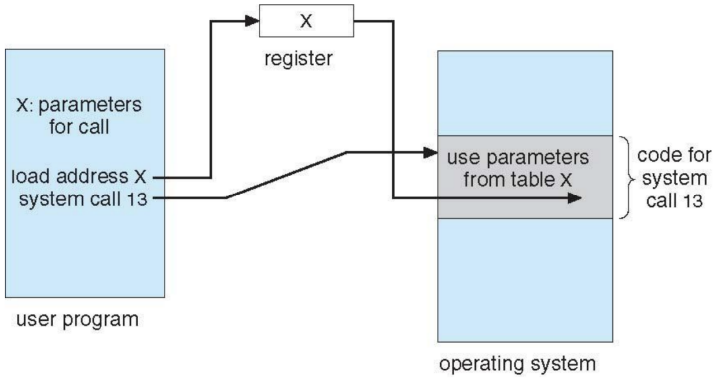

Figure 1.9: Parameter passing

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.

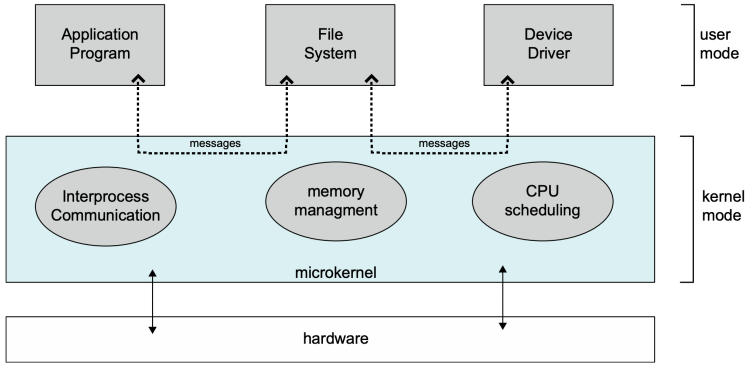

Figure 1.10: Microkernel structure

Chapter 2

Processes

A process is a program in execution. Processes are identified by a **process identifier** (pid). It is composed of multiple parts:

- 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

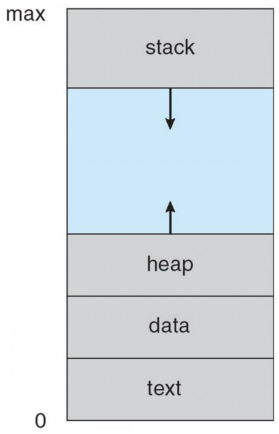

Figure 2.1: Memory layout for a process

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

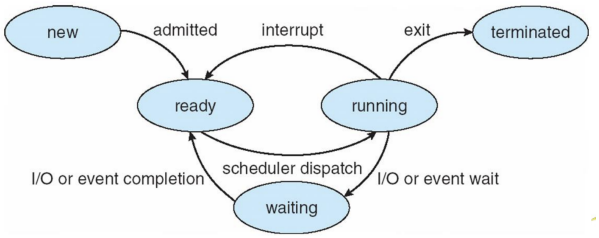

Figure 2.2: Process state machine

The information about 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`.

When a process is stopped it saves its state in the PCB and if reloads it when it resumes executing. The time when the CPU stores the PCB of a process and loads the PCB of another process is called **context switch**. 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 Multithreading

A process can execute multiple instructions at once by using multiple threads. Each thread has its own program counter and uses different registers, therefore all this information has to be also stored in the PCB.

2.2 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

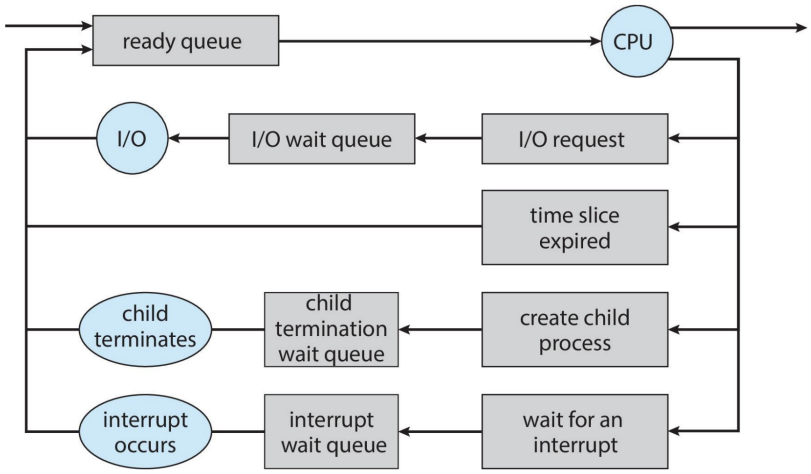

Figure 2.3: Process queues

2.3 Process creation

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

The parent and children have different options for sharing resources:

- 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 the UNIX processes are managed using the following system calls:

- `fork()`: creates a new 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

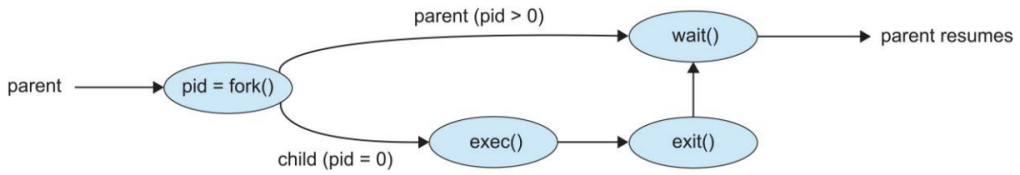

Figure 2.4: Creation of children processes

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

```c
#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

2.4 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

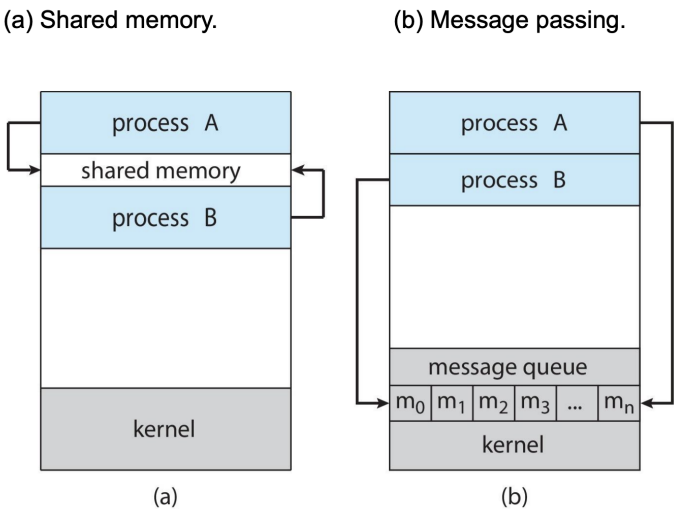

Figure 2.5: Models of communication between processes

2.4.1 Shared memory

Processes can communicate using a feature called shared memory. Processes can allocate an area in memory as shared memory and assign to it a name. Then they can access it by mapping it to their address space[1]. In UNIX memory mapping is achieved by the `mmap()` system call.

```c
#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

2.4.2 Message passing

Processes can communicate without using shared memory by using message passing. This can be physically implemented in the following ways:

- Shared memory (already seen in the previous section)
- 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. In Windows they are called ordinary pipes.

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

```c
#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 without 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