Operating systems

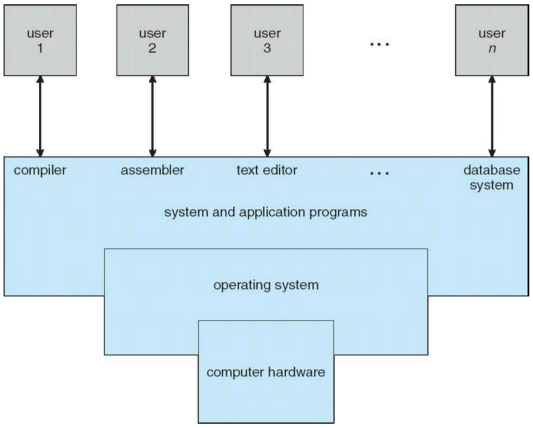
Salvatore Andaloro

Contents

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:

- execute programs that solve problems
- make the computer easy to use
- use the hardware in an efficient manner



To do so it coordinates access to the hardware among the different applications and users.

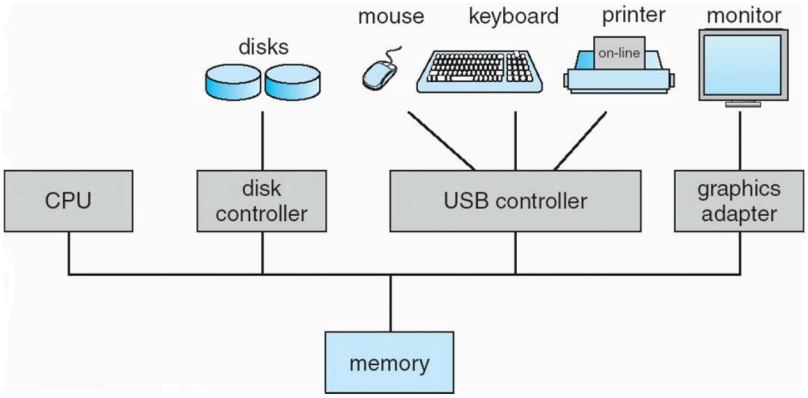
The program that runs all the time is the **kernel**, everything else is either a system program (ships with the operating system) or an application program.

1.1 Computer startup

The first program that runs on startup is the **bootstrap program**. This is stored in 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. different programs from kernels but that must be started at startup. For example in UNIX there is *systemd*, that in turn starts other daemons.

1.2 General computer architecture

The general model of a 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.



1.2.1 Interrupts

In interrupt driven operating systems, device controllers communicate with the CPU using **interrupts**. While waiting for the CPU device controllers store information in local buffers, that are then read/written by the CPU and the stored data is transferred from/to the main memory.

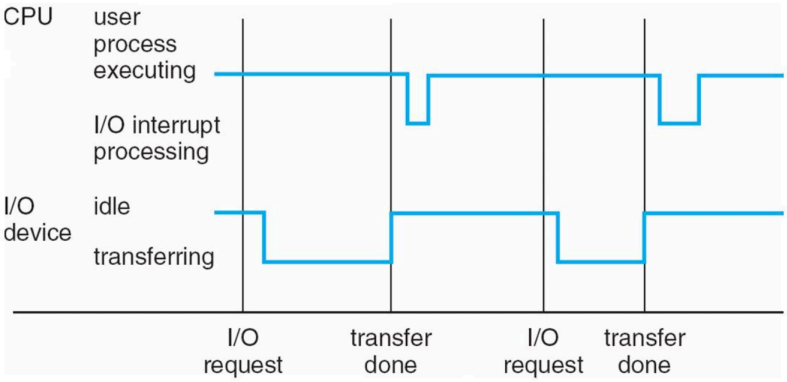


Figure 3: Interrupt timing diagram

When an interrupt happens, the OS saves the current program counter (PC), jumps to the routine that handles the interrupt and then resumes normal execution by jumping back to where the PC was pointing to. The position to where to jump to when an interrupt is received is stored in the **interrupt vector or table**, which is a map where each interrupt is linked to an instruction memory address.

A **trap** or **exception** is a software-generated interrupt caused by an error or a 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. This can be very very wasteful, therefore a feature called DMA (Direct Memory Access) has been introduced. In this technique the CPU receives an interrupt only after all the data has been transferred successfully.

1.2.2 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.

The main memory of a computer, which can be accessed directly by the CPU is DRAM (dynamic random access memory, based on charged capacitors) or SRAM (static random access memory, based on inverters), which is usually volatile. Secondary storage has much bigger capacity and is non-volatile (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.

1.2.2.1 Caching

Caching is a very common technique for speeding up access to commonly used data. Information is copied from slower to faster storage temporarily and then every time that information is needed the OS will first check if it is present in cache. Due to cost constraints, cache is very often smaller than the storage being cached, therefore cache management must be properly optimized.

1.3 Modern system architectures

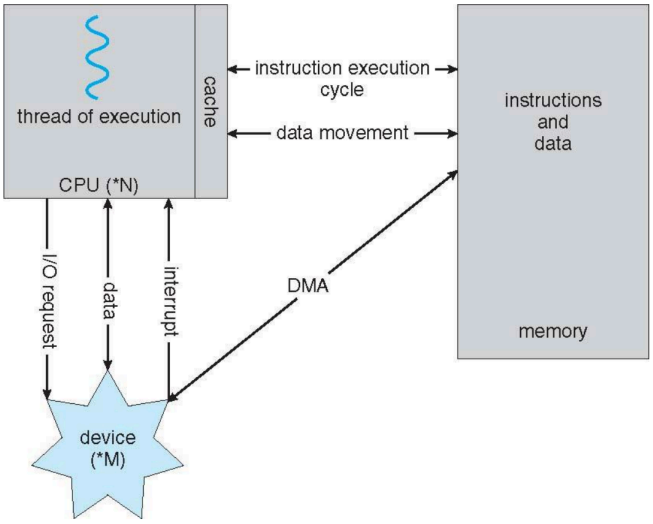


Figure 4: A Von-Neumann architecture

Currently most systems are multiprocessor and/or multi-core. In these systems, we can allocate tasks to processors in different ways:

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

1.3.1 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.

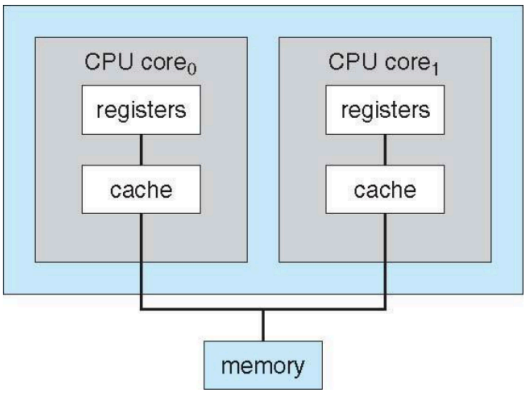


Figure 5: A multi-core processor

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

1.3.2 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.

1.3.3 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.

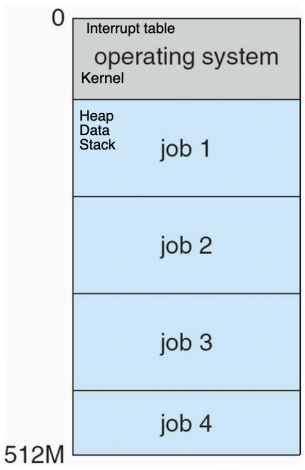


Figure 6: Memory layout for multiprogramming systems

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. Intel processors have for modes of operation, where 0 is fully privileged and 3 is fully restricted.

1.3.4 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.

1.3.5 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

1.3.6 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

- 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.

1.3.7 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 OS protection

OS must provide mechanisms to defend the system against external attacks. An attack is anything posing a threat to:

- Confidentiality
- Availability
- 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.

1.5 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).

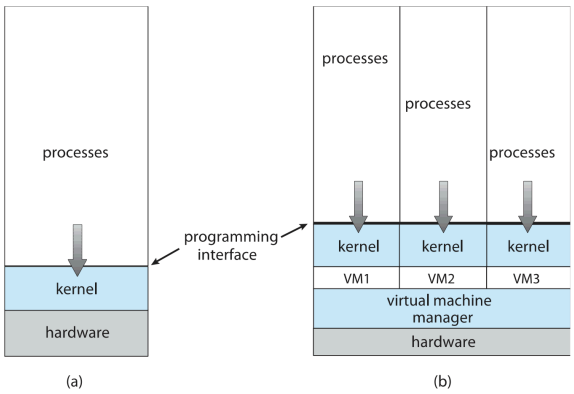


Figure 7: Virtualization

1.6 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

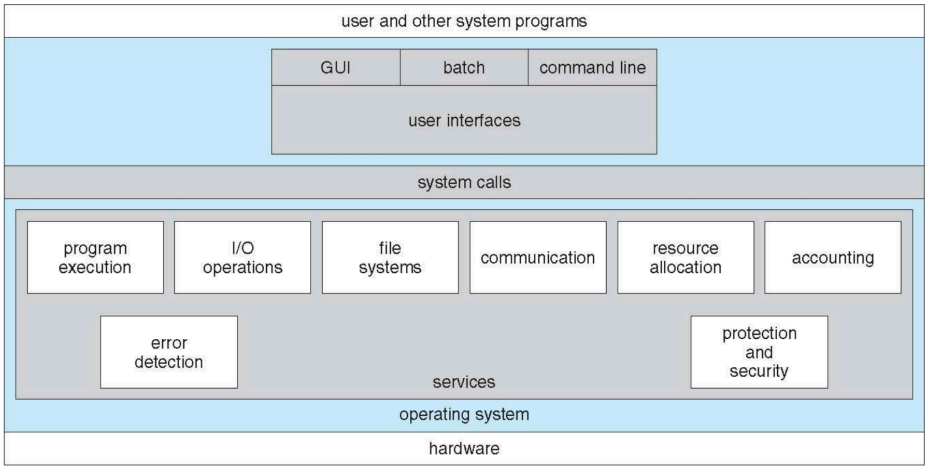


Figure 8: Services provided by an operating system

1.7 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.

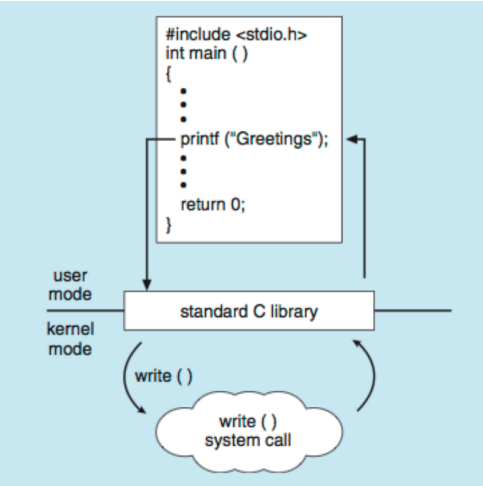


Figure 9: C program invoking printf() library call, which calls write() system call

1.7.1 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.

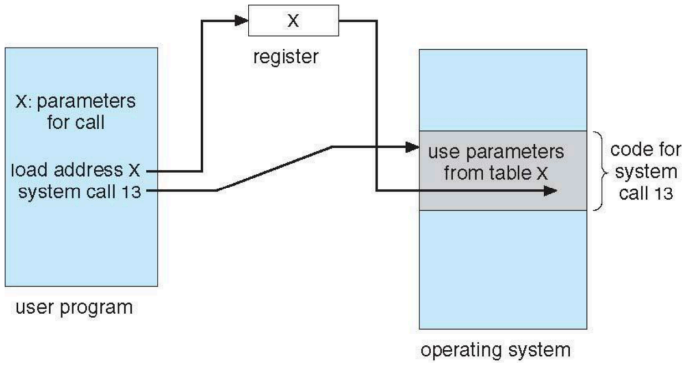


Figure 10: Parameter passing

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

1.8 OS design

When designing an operating system, the following aspects need to be taken into consideration:

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

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.

1.9.2 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.

1.9.3 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 & 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 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.

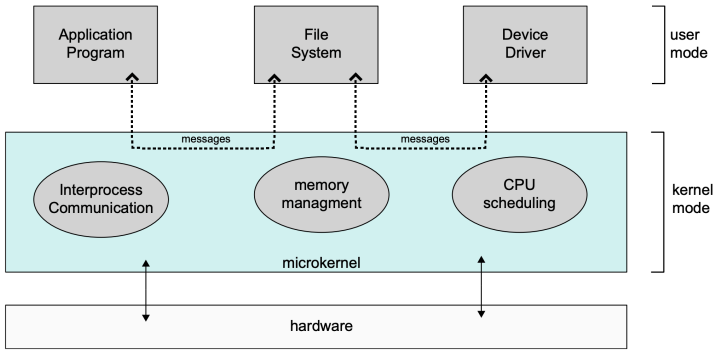


Figure 11: Microkernel structure

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

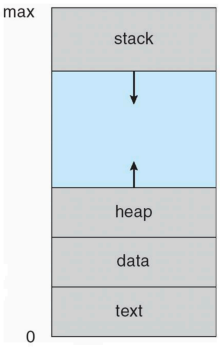


Figure 12: Memory layout for a process

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

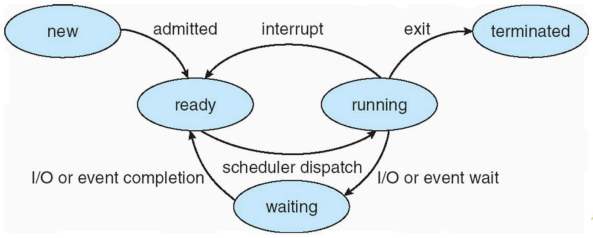


Figure 13: Process state machine

The information about the state of the process is stored in the RAM in the process control block (PCB). It 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 c. s.: the process stops itself because needs to wait for a resource
- nonvoluntary c. s.: 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

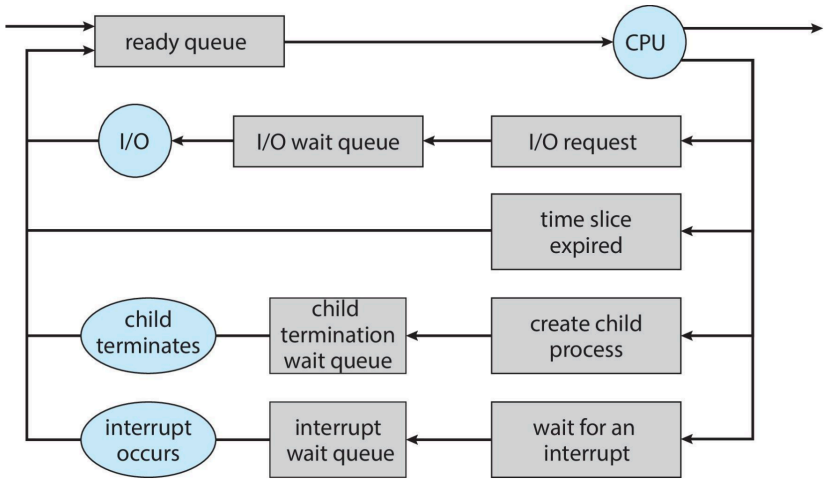


Figure 14: Process queues

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

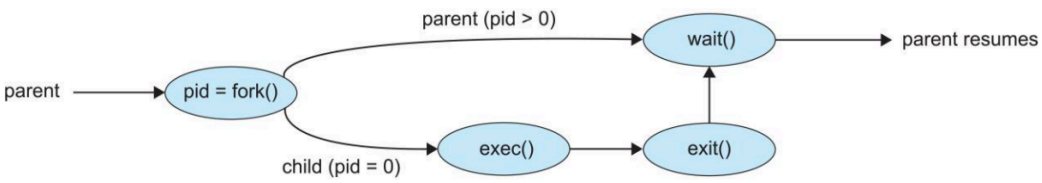


Figure 15: Creation of children processes

```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

2.4 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

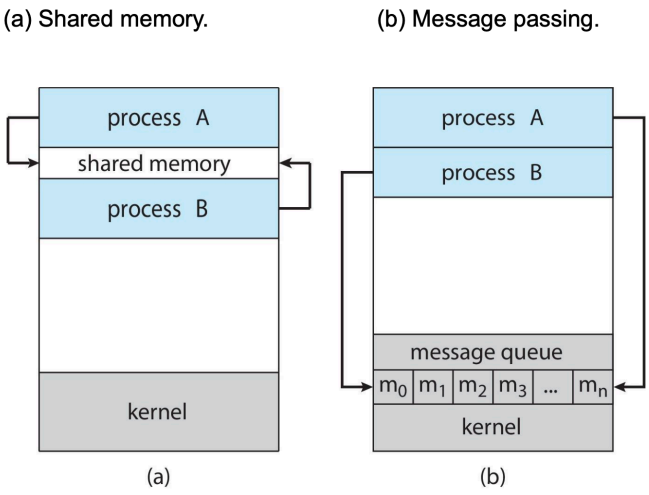


Figure 16: Models of communication between processes

2.4.1 Shared memory

Processes can communicate using shared memory by creating a dedicated area in memory and writing and reading to it. To access shared memory it processes need to map it to memory using memory mapping. Shared memory can be accessed by their name.

2.4.1.1 Memory-mapped files

In modern operating systems files, shared memory objects and other resources that can be addressed by a file descriptor can be accessed by processes using a procedure called "memory mapping". When memory mapping is used, a byte-to-byte correlation with a part of memory and the resource is established. This means that the processor can access the file very quickly, as if it was stored in memory. In UNIX this feature is provided 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);

  /* now 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;
  }

  /**
   * Now 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>
```

```c
#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);
  }

  /* now 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);
  }

  /* now 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

2.5 Message passing

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

- Shared memory (we already saw that)
- 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

2.5.1 Pipes

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

2.5.1.1 Ordinary 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.

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

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

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

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

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

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

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

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

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

    /* read from the pipe */
    read(fd[READ_END], read_msg, BUFFER_SIZE);
```

```
        printf("child read %s\n",read_msg);

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

    return 0;
}
```

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

2.5.1.2 Named pipes

Named pipes can be accessed without a parent-child relationship. They are bidirectional and multiple processes can read and write to it.

```
#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 the content from the standard input in the pipe

```
#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 6: A process that opens a named pipe and prints the content in the pipe to the standard output