Tutoriat 2 (Operating Systems)
Processes

Contents

1 Introduction

A program is an **executable file** stored on disk; it is a *passive entity.* A **process** is a **program in execution** (an executable file loaded into memory; an *active entity*) which has a **program counter** (a register which contains the address of the next instruction to be executed) and a set of associated resources (allocated CPU time, memory, files, I/O devices). Most operating systems assign a unique integer value to each process, known as a **PID (process identifier)**.

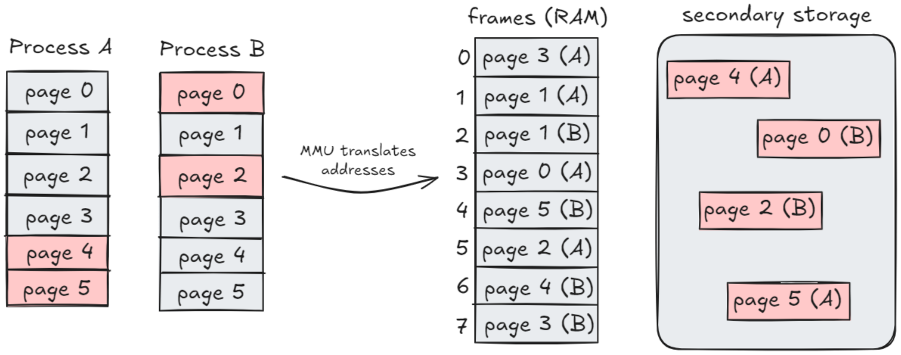


Figure 1: Virtual addresses getting mapped to physical addresses

A **physical memory address** represents a certain location in RAM. A **virtual memory address** is a theoretical memory zone (that does not physically exist), corresponding to a certain process. The **address space** of a process represents the range of virtual memory addresses associated with the process. Virtual addresses are indexed from 0 and an address **X** may be found in multiple processes, but it is not shared.

A **page** is a fixed-size block of virtual memory, whereas a **frame** is a fixed-size block of physical memory. Frames and pages on a system are of the same dimension (which may be 4KB, 16KB, etc.); this is necessary, as frames are used to store pages. The **address space** of a process is composed of pages.

Virtual addresses have the goal of simplifying calculations: each process is under the impression that it is operating with a continuous block of memory. In reality, some pages are stored in frames, and some are placed in secondary storage (when there is not enough RAM available). Sometimes, certain pages in RAM that are not accessed frequently are swapped out for other pages from the disk.

The **MMU (memory management unit)** is a hardware component responsible for translating virtual addresses to physical memory addresses. The way it works (along with some other concepts, such as the **TLB** = translation lookaside buffer, **page tables** and **page faults**, etc.), will all be discussed at another time.

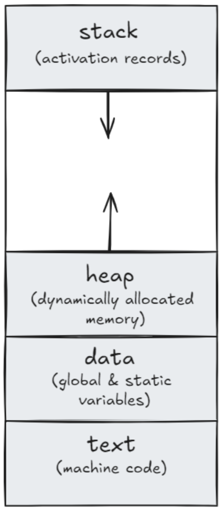


Figure 2: Structure of a process in memory

When a function is called, a block of memory - known as an **activation record** - is allocated for the data necessary to execute the call. This includes function parameters, local variables, and the *return address* (the memory address of the next instruction to be executed once the call is finalized).

A process has the following structure:

- **Stack section (dynamic size):** for the activation records. Each function call is a **push** on the stack, and a call that is finalized is a **pop**.
- **Heap section (dynamic size):** memory that is allocated and deallocated at runtime.
- **Data section (fixed size):** initialized global and static variables.
- **Text section (fixed size):** compiled machine code of the program.

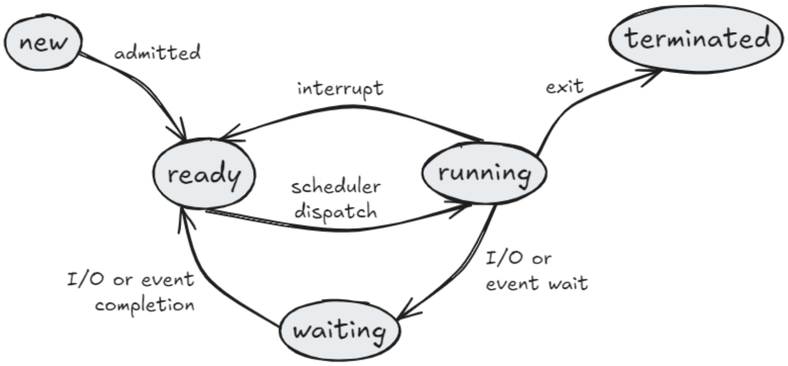


Figure 3: Process states diagram

A process may enter any of the following states:

- **New:** it has just been created.
- **Ready:** it is ready to execute on the CPU.
- **Running:** it has been selected by the *CPU scheduler* to run (this will be further explained immediately).
- **Waiting:** an I/O request or a certain event has occurred, forcing it to temporarily halt its execution.
- **Terminated:** it has finished executing; its resources may be collected by the operating system.

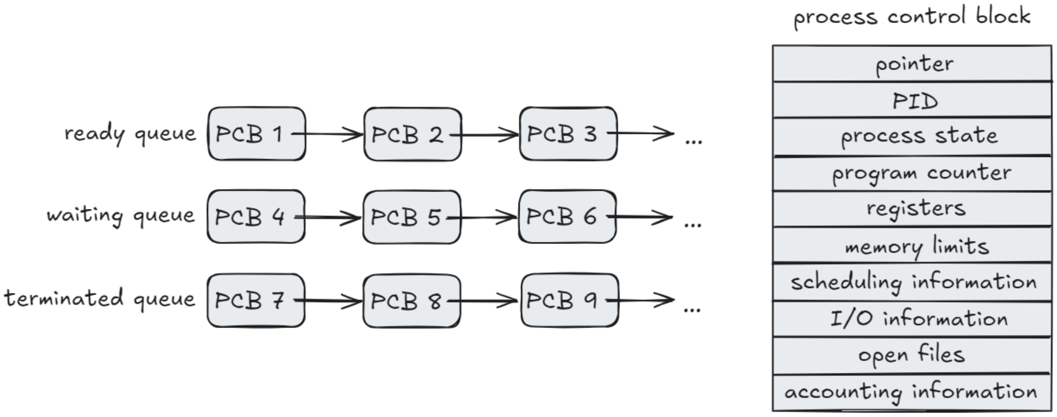


Figure 4: Structure of a PCB and examples of some process queues

A **PCB (process control block) / TCB (task control block)** is a *data structure* used by an operating system to store all information about a specific process, including its state, PID, program counter, register values, memory information (the memory allocated to the process), CPU scheduling information (priorities and scheduling queue pointers; to be discussed), I/O status information (I/O devices allocated to the process, list of open files), and accounting information (CPU time, clock time elapsed since starting its execution, time limits).

Once a process is newly created, the associated PCB is also created and placed in a memory zone that normal users do not have access to. As the process continues to run, the PCB updates its information (e.g., the program counter and the process state). When a process temporarily halts its execution, a **state save** is performed (the PCB keeps track of all the information associated with the process as it pauses execution, a snapshot); once the process resumes execution on the CPU, a **state restore** is performed (register values and other necessary data are brought back from the PCB).

There are a few types of *queues* (data structures) used by the operating system to handle processes, e.g. the **ready queue** holds a list of processes that are in main memory and ready to be executed by the CPU; the **terminated queue** holds the list of processes that have finished their execution and are waiting

2 Process scheduling

A CPU core may execute one instruction at a time. In order for the system to optimize resource use, there must always be a process executing; considering that some processes may be more important than others, and there are usually more processes trying to execute than there are cores available, some processes must wait for their turn to execute, meaning there is a need for the scheduling of their executions. This task is handled by a system program called the **CPU scheduler**.

Each process has a certain time slice (measured in milliseconds) for which it is allowed to execute on the CPU. Once this time slice expires, or the process gets temporarily blocked (I/O request), or it has finished its execution, the **scheduler** must allocate the core to another process that wants to execute.

A CPU scheduler may be **preemptive** (a process which is executing can be forcibly interrupted to allow another process to execute, even if it hasn't finished its execution) or **nonpreemptive** (once a process is assigned to the CPU, it is not interrupted until it completes its execution or moves to a waiting state for an I/O request).

Processes may be viewed as **I/O-bound** (they spend most of their time waiting for I/O operations) or **CPU-bound** (they spend most of their time executing instructions). The number of processes currently in memory that are ready for execution is known as the **degree of multiprogramming**.

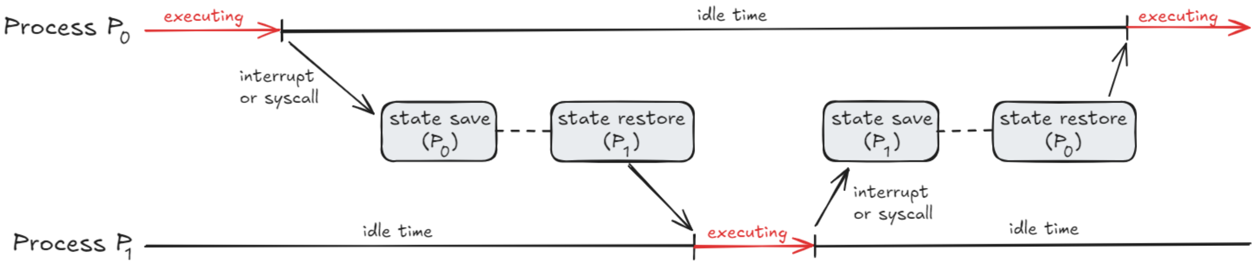


Figure 5: Context switching between two processes

A **context switch** occurs when the CPU switches from one process to another. This means performing a *state save* of the current process, a *state restore* of the other process, and resuming execution. The **context switch overhead** is the amount of time and resources required to perform the context switch; this may the slow the system down, and the overhead varies from OS to OS.

2.1 Scheduling queues

A **scheduling queue** is a data structure that an OS uses to manage processes that are waiting for a resource. These queues hold the PCBs for processes in different states. Examples:

- **Ready queue:** all processes that are in main memory waiting to be executed by the CPU.
- **Wait queue:** holds processes that are temporarily blocked, waiting for an event (such as an I/O request).

- **Job queue:** a more general queue that contains all processes in the system, often stored in secondary memory.
- **Suspended queue:** holds processes that have been temporarily removed from main memory, often due to being swapped out to disk to free memory.

One of the following things may happen when a process is executing on the core:

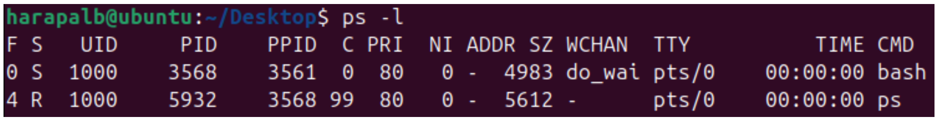
- An I/O request takes place, putting the process in an **I/O wait queue**. At one point, the request will end, transferring the process from the **wait queue** to the **ready queue**.
- The process creates a **child process** (we will soon expand on this topic); while waiting for the child to terminate (finish its execution), the process is placed in a **wait queue**.
- Due to an **interrupt** (or finishing its time slice), the process is placed in the **ready queue**.

3 Working with processes in the terminal and in C

A process may create other processes. Thus, the main process is called a **parent process**, whereas its children are known as **child processes**, ultimately forming a **process tree**. A *child process* may either get its resources straight from the OS or be constrained to a subset of the parent's resources. The *parent process* may have to divide its resources among its children or allow them to access a common subset of its resources.

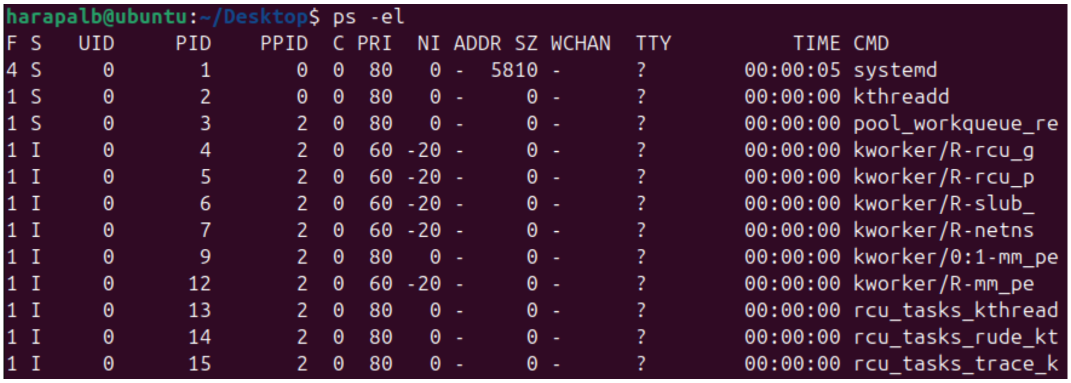
3.1 Listing process information

Firstly, there are a few commands that can be used to list detailed information about all existing processes. For example, **ps -l** (”ps” stands for process status”, ”-l” = long format) lists information about the existing processes associated with the current terminal or session:



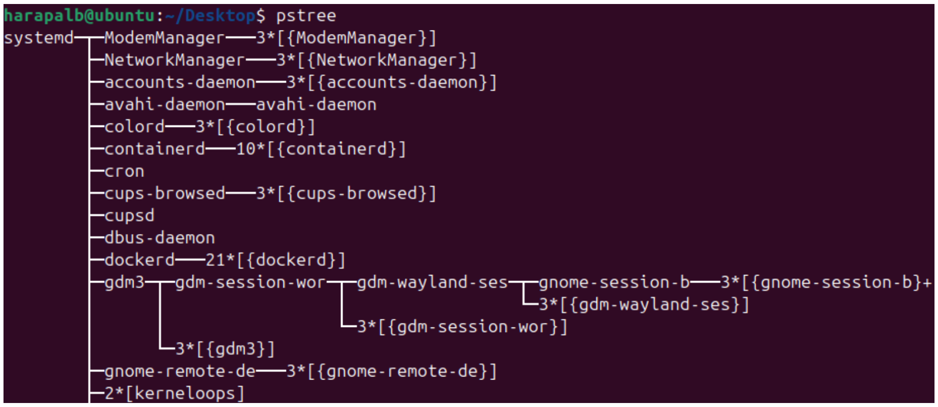
```
harapalb@ubuntu:~/Desktop$ ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000    3568    3561  0  80   0 -  4983 do_wai pts/0    00:00:00 bash
4 R  1000    5932    3568 99  80   0 -  5612 -      pts/0    00:00:00 ps
```

The command can be run without the ”-l” flag, but it will show much less information. The output above consists of process states, PIDs, process priority, number of pages, etc. This information can also be shown for every existing process in the system, by running **ps -el** (where ”-e” stands for extended).



```
harapalb@ubuntu:~/Desktop$ ps -el
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0       1       0  0  80   0 -  5810 -      ?        00:00:05 systemd
1 S     0       2       0  0  80   0 -     0 -      ?        00:00:00 kthreadd
1 S     0       3       2  0  80   0 -     0 -      ?        00:00:00 pool_workqueue_re
1 I     0       4       2  0  60 -20 -     0 -      ?        00:00:00 kworker/R-rcu_g
1 I     0       5       2  0  60 -20 -     0 -      ?        00:00:00 kworker/R-rcu_p
1 I     0       6       2  0  60 -20 -     0 -      ?        00:00:00 kworker/R-slub_
1 I     0       7       2  0  60 -20 -     0 -      ?        00:00:00 kworker/R-netns
1 I     0       9       2  0  80   0 -     0 -      ?        00:00:00 kworker/0:1-mm_pe
1 I     0      12       2  0  60 -20 -     0 -      ?        00:00:00 kworker/R-mm_pe
1 I     0      13       2  0  80   0 -     0 -      ?        00:00:00 rcu_tasks_kthread
1 I     0      14       2  0  80   0 -     0 -      ?        00:00:00 rcu_tasks_rude_kt
1 I     0      15       2  0  80   0 -     0 -      ?        00:00:00 rcu_tasks_trace_k
```

The command **pstree** shows the full process tree of the system. As can be seen, **systemd** is the first process that gets created and runs on the system. Additionally, there are some processes known as **”daemons”**; these are *background processes* (which run in the background independently of any user interaction) that *operate continuously* and *automate* certain tasks.



```
harapalb@ubuntu:~/Desktop$ pstree
systemd─┬─ModemManager───3*[{ModemManager}]
        ├─NetworkManager───3*[{NetworkManager}]
        ├─accounts-daemon───3*[{accounts-daemon}]
        ├─avahi-daemon───avahi-daemon
        ├─colord───3*[{colord}]
        ├─containerd───10*[{containerd}]
        ├─cron
        ├─cups-browsed───3*[{cups-browsed}]
        ├─cupsd
        ├─dbus-daemon
        ├─dockerd───21*[{dockerd}]
        ├─gdm3─┬─gdm-session-wor─┬─gdm-wayland-ses─┬─gnome-session-b───3*[{gnome-session-b}+
        │      │                 │                 └─3*[{gdm-wayland-ses}]
        │      │                 └─3*[{gdm-session-wor}]
        │      └─3*[{gdm3}]
        ├─gnome-remote-de───3*[{gnome-remote-de}]
        ├─2*[kerneloops]
```

3.2 Creating a process

Firstly, **fork** is used to create a new child process identical to the parent process, which will be assigned its own unique PID; however, the parent will not receive the child's PID as a return value for the fork call, but the value **0**. In this way, if the returned value is < 0, then something went wrong; if it is $== 0$, it is the child process, and otherwise it is the parent process.

The purpose of the **exit** function is to terminate a process' execution and return control to the host (the parent process), signaling success or failure via an exit status code. It is not strictly necessary to include this function in the code of the child process; however, omitting it may produce some issues, which will be discussed immediately. After the OS has deallocated the process' resources, its entry will still remain in the **process table**, until the parent calls *wait*.

The **wait** function is used to force the parent to wait until one or more of its children have finished their execution. If the child process calls *exit*, the parent process may receive the exit status code via the *wait* call. If the parent does not call *wait* and it is still running while its children have terminated, then its children become **zombie processes**; if the parent does not call *wait* and it terminates before its children, then they become **orphan processes**.

In order to address the issue of *orphan processes*, some systems have assigned certain processes to automatically become the parents of any orphans. However, some systems do not allow child processes to exist if the parent has terminated; **cascading termination** comes into play, terminating all children/grandchildren/etc.

```c
/* Creating one child process (v1) */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <errno.h>

int main() {
    pid_t pid = fork();

    if (pid < 0) {
        return errno;
    }
    else if (pid == 0) {
        printf("child, PID=%d (parent=%d)\n", getpid(), getppid());
    }
    else {
        wait(NULL);
        printf("parent, PID=%d\n", getpid());
    }
    return 0;
}

/* Output example:
child, PID = 24422 (parent=24421)
parent, PID = 24421
*/
```

Line 9 calls *fork* and creates a new child process, storing the return value in the variable *pid*. The child process' execution starts exactly right after the fork call; thus, it goes into the *if statement*. The child will be assigned to the *pid* == 0 branch, and the parent will be assigned to the *else* branch. The child will execute its code (printing its PID and its parent's PID) before the parent because the parent called *wait*.

Notice how the child did not call *exit*. Here, it is not necessary, as the child will not continue executing the parent's code (because the parent's code is protected by the *else* branch); the exit status of the child is given by the exit status of *main* (which should be 0). Let us slightly edit the code, making an *exit* call necessary:

```
/* Creating one child process (v2) */

pid_t pid = fork();

if (pid < 0) {
    return errno;
}
else if (pid == 0) {
    printf("child, PID=%d (parent=%d)\n", getpid(), getppid());
    exit(0);
}

wait(NULL);
printf("parent, PID=%d\n", getpid());
return 0;
```

The child process will follow the *else if (pid == 0)* branch and terminate within the branch; it is no longer necessary to include the *else* branch for the parent. What if we want to create more than one child process? There is the possibility of taking this approach:

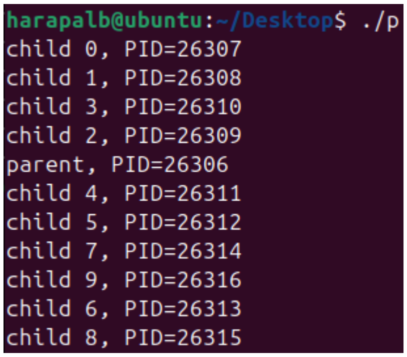
```
/* Creating two child processes */

pid_t pid1 = fork();

if (pid1 < 0) {
    return errno;
}
else if (pid1 == 0) {
    printf("child 1, PID=%d (parent=%d)\n", getpid(), getppid());
    exit(0);
}

pid_t pid2 = fork();

if (pid2 < 0) {
    return errno;
}
else if (pid2 == 0) {
    printf("child 2, PID=%d (parent=%d)\n", getpid(), getppid());
    exit(0);
}

wait(NULL);
wait(NULL);
printf("parent, PID=%d\n", getpid());

/* output example:
child 2, PID=25268 (parent=25266)
child 1, PID=25267 (parent=25266)
parent, PID=25266
*/
```

Whichever one of the children executes first is handled by the CPU scheduler, so the order may differ. However, if we wish to create a larger number of processes (say 10), then *for loops* are a more adequate approach.

In order to experiment with visualizing the process tree, the child processes will each be put to sleep for a few seconds (the first child process will sleep for 5 seconds, the second one for 6, and so on); this way, we will be able to observe different stages of the process tree, with the children gradually terminating.

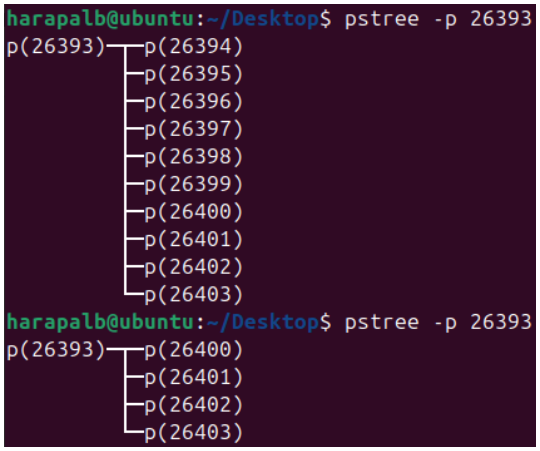
```
/* Creating 10 child processes */

int num_children = 10;
for (int i = 0; i < num_children; ++i) {
    pid_t pid = fork();

    if (pid < 0) {
      return errno;
    }
    else if (pid == 0) {
      printf("child %d, PID=%d\n", i, getpid());
      sleep(i + 5);
      exit(0);
    }
}

printf("parent, PID=%d\n", getpid());
for (int i = 0; i < num_children; ++i) {
    wait(NULL);
}
return 0;
```

The output of this program would look as follows:



```
harapalb@ubuntu:~/Desktop$ ./p
child 0, PID=26307
child 1, PID=26308
child 3, PID=26310
child 2, PID=26309
parent, PID=26306
child 4, PID=26311
child 5, PID=26312
child 7, PID=26314
child 9, PID=26316
child 6, PID=26313
child 8, PID=26315
```

Since the child processes are put to sleep as soon as they print their information, there is plenty of time to open a second terminal to view the process tree derived from the parent process. To do this, run the main program, identify the PID of the parent process and run the command (in the second terminal) **pstree -p <parent_PID>**. Here is an example:



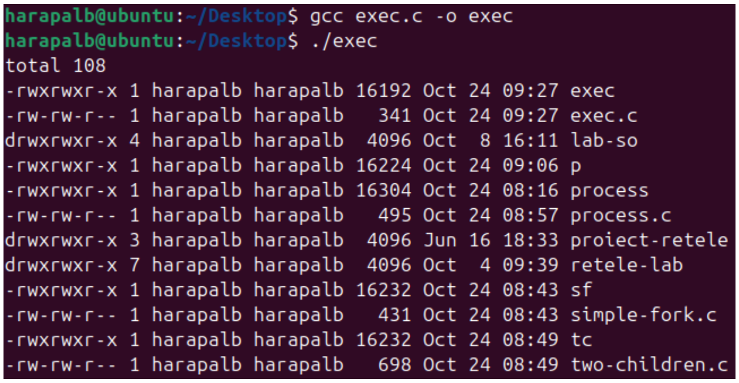
```
harapalb@ubuntu:~/Desktop$ pstree -p 26393
p(26393)─┬─p(26394)
         ├─p(26395)
         ├─p(26396)
         ├─p(26397)
         ├─p(26398)
         ├─p(26399)
         ├─p(26400)
         ├─p(26401)
         ├─p(26402)
         └─p(26403)
harapalb@ubuntu:~/Desktop$ pstree -p 26393
p(26393)─┬─p(26400)
         ├─p(26401)
         ├─p(26402)
         └─p(26403)
```

Running the command in the first 5 seconds shows all ten child processes. Waiting a few more seconds will result in some of the child processes finishing their execution (since they finish their sleep); thus, printing the process tree again would show fewer processes.

3.3 Loading an existing program in the child process

The **execve** command is used to replace an existing process with another program. It takes 3 arguments: the absolute path to the program to execute, a string array of the arguments for the program (the last element has to act as a terminator symbol, being NULL), and an array of environment variables that will be passed to the new program (which can be NULL, in order to keep the current environment). Here is an example of a child process being replaced with a process that runs the **ls** command, listing information regarding the files in the current working directory:

```
/* Create a new process to run the "ls" command */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <errno.h>

int main() {
    char* const child_args[] = {"ls", "-l", NULL};
    pid_t pid = fork();

    if (pid < 0) {
        return errno;
    }
    else if (pid == 0) {
        execve("/usr/bin/ls", child_args, NULL);
        exit(0);
  }

    wait(NULL);
    return 0;
}
```



```
harapalb@ubuntu:~/Desktop$ gcc exec.c -o exec
harapalb@ubuntu:~/Desktop$ ./exec
total 108
-rwxrwxr-x 1 harapalb harapalb 16192 Oct 24 09:27 exec
-rw-rw-r-- 1 harapalb harapalb   341 Oct 24 09:27 exec.c
drwxrwxr-x 4 harapalb harapalb  4096 Oct  8 16:11 lab-so
-rwxrwxr-x 1 harapalb harapalb 16224 Oct 24 09:06 p
-rwxrwxr-x 1 harapalb harapalb 16304 Oct 24 08:16 process
-rw-rw-r-- 1 harapalb harapalb   495 Oct 24 08:57 process.c
drwxrwxr-x 3 harapalb harapalb  4096 Jun 16 18:33 proiect-retele
drwxrwxr-x 7 harapalb harapalb  4096 Oct  4 09:39 retele-lab
-rwxrwxr-x 1 harapalb harapalb 16232 Oct 24 08:43 sf
-rw-rw-r-- 1 harapalb harapalb   431 Oct 24 08:43 simple-fork.c
-rwxrwxr-x 1 harapalb harapalb 16232 Oct 24 08:49 tc
-rw-rw-r-- 1 harapalb harapalb   698 Oct 24 08:49 two-children.c
```

4 Interprocess communication (IPC)

Processes within a system may be **independent** (does not affect or rely on the results of any other processes) or **cooperating** (affects or is affected by other processes, due to data sharing). Benefits of cooperating processes:

- **Information sharing:** useful when certain data are supposed to be shared between multiple applications.
- **Modularity:** structuring a more complex task into simpler subtasks.
- **Computation speedup:** in order to speed up calculations, each subtask may execute on a different core.

Cooperating processes require a way of sharing data, an **interprocess communication (IPC) mechanism**. There are two fundamental ways to achieve this: **shared memory** and **message passing**.

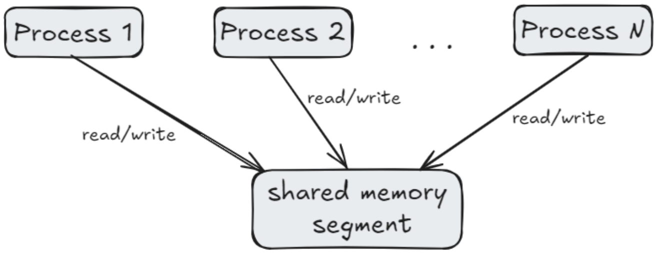


Figure 6: Processes cooperating through a shared memory segment

4.1 Shared memory

A technique for multiple processes to communicate and exchange data efficiently may be **shared memory**. One of the processes establishes a region of memory (a **shared memory segment**) through a system call, which can be attached to the *address spaces* of different processes, allowing them to read and write directly to the segment. Recall that the communication is managed by the cooperating processes, not the OS.

Shared memory can be used as a solution to the **consumer-producer problem:** a **producer process** produces information, which is then consumed by a **consumer process**. Both processes read and write to the same shared memory segment. The produced objects are stored on a **buffer** (a temporary storage area in memory, used to hold data while it is being moved from one place to another) that can be of either type:

- **Unbounded buffer:** there is no threshold for the amount of objects that may be produced. However, the consumer must wait if there are no objects available in the buffer.
- **Bounded buffer:** there is a limit to how many objects can be produced. Thus, the producer must wait when the buffer is full, and the consumer must wait when it is empty.

In order to implement a solution to the *consumer-producer problem*, let us consider the following variables that will be accessed by the *cooperating processes* in a *shared memory segment*:

```
#define BUFFER_SIZE 10

typedef struct {
    ...
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

The buffer is implemented as a *circular array.* The variable **in** points to the next free position in the buffer, whereas **out** points to the first occupied slot. If the buffer is empty, then **in == out** holds true. If **(in + 1) % BUFFER_SIZE == out**, then the buffer is full.

Producer process

```
item next_produced;

while (true) {
    /* produce an item, store it in
        next_produced */

    while (((in + 1) % BUFFER_SIZE) == out);
         /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

Consumer process

```
item next_consumed;

while (true) {
    while (in == out); /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item stored in
        next_consumed */
}
```

Issue: there can only be *BUFFER_SIZE - 1* items stored in the buffer. Assume that there is only one free slot left; storing an item in that slot and incrementing *in* would lead to *in == out*, which means that the buffer is empty (when it is actually full).

The solution for this is simple: to use an extra variable **count = 0**, which keeps track of the number of items currently in the buffer.

Producer process

```
item next_produced;

while (true) {
    /* produce an item, store it in next_produced */

    while (count == BUFFER_SIZE); /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
}
```

Consumer process

```
item next_consumed;

while (true) {
    while (count == 0); /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;

    /* consume the item stored in next_consumed */
}
```

However, there is yet another major problem: multiple processes accessing shared data at the same time may corrupt the data. Consider the increment instruction **count++**. The way this would be implemented in machine code is as follows (the implementation is similar for **count−−**):

```
register1 = count
register1 = register1 + 1
count = register1
```

Simultaneously running **count++** and **count−−** results in a random ordering of instructions in machine code. An example of such a sequence may be as follows (consider **count = 5**):

```
register1 = count           [register1 = 5]
register1 = register1 + 1   [register1 = 6]
register2 = count           [register2 = 5]
count = register1           [count = 6]
register2 = register2 - 1   [register2 = 4]
count = register2           [count = 4]
```

It can be seen that the final value of **count** is 4, instead of 5. This phenomenon is known as a **race condition** and will be studied in depth at another time.

4.2 Message passing

This communication method consists of processes sending and receiving **messages** to exchange data, rather than using shared memory. There are two fundamental operations: **send(message)** and **receive(message)**, where the messages may be of variable or fixed size. If two processes wish to send each other messages, there must be a **communication link**, which may be implemented in a variety of ways.

4.2.1 Naming

Naming refers to how the sender and receiver identify each other. Communication may take place **directly** or **indirectly**. In **direct communication**, the processes must know each other's names, to explicitly identify each other. Thus, the operations become:

- **send(P, message):** send a message to the process P.
- **receive(Q, message):** receive a message from the process Q.

The properties of the communication links are as follows:

- Between each pair of processes, there exists exactly one communication link. As such, processes must only know their identities in order to communicate.
- A link is associated with exactly one pair of communicating processes.

This approach is considered to be a **symmetric approach**, since the sender and receiver must name each other explicitly in order to communicate. An **asymmetric approach** may be followed as well; the operations become:

- **send(P, message):** sends a message to the process P.
- **receive(id, message):** receive a message from any process. The variable **id** receives the sender id.

For an **indirect communication** approach, messages are directed to and received from **mailboxes (ports**; a kernel-managed memory area used for IPC). The operations become as follows:

- **send(A, message):** send a message to mailbox A.
- **receive(A, message):** receive a message from mailbox A.

For this variant, the properties of the communication links are as follows:

- A link is established only if processes share a common mailbox.
- A link may be associated with many processes.
- Each pair of processes may share several communication links.
- The link may be unidirectional or bidirectional.

There is one problem: consider the processes P_1, P_2, and P_3 which share the mailbox A. The process P_1 sends a message to A, while P_2 and P_3 try to *receive* the message at the same time. The question to ask is which of these two processes receives the message? The answer depends on the strategy applied:

- Allow a link to be associated with at most two processes.
- Allow only one process to execute a *receive* operation at a time.
- Allow the system to randomly select a process to receive the message (additionally, a selection algorithm may be used).

4.2.2 Synchronization

This concept refers to how the sending and receiving processes coordinate. The coordination may take place **synchronously (blocking)** or **asynchronously (non-blocking)**. In a **synchronous environment**, this is how the operations take place:

- **Blocking send:** the sender sends a message and blocks until either the receiver or the mailbox gets the message.
- **Blocking receive:** the receiver is blocked until a message is available.

In an **asynchronous environment**:

- **Non-blocking send:** the sender ships out the message and continues its execution.
- **Non-blocking receive:** the receiver gets a valid message or a NULL message.

4.2.3 Buffering

This is the mechanism for temporarily storing messages; they are placed in a queue, which can be implemented in three ways:

- **Zero capacity:** no messages are stored. The sender and receiver must communicate **synchronously** (blocking send and receive).
- **Bounded capacity:** the queue can hold at most **n** messages. If the queue is full, the sender blocks until a slot is available.
- **Unbounded capacity:** the queue can hold an infinite amount of messages. The sender may ship out as many messages as it would like (non-blocking send).

4.3 Pipes

Pipes are an IPC mechanism. Depending on how they are designed, they can send data in one direction (**undirectional**) or both (**bidirectional**). Similarly, they can operate as **half-duplex** (data flows only one way at a time) or **full-duplex** (data can flow both ways simultaneously). They can connect processes that are **related** (such as a parent-child relationship) or **unrelated**.

4.3.1 Ordinary pipes

Ordinary pipes are *unidirectional*. They can only be used between processes that share a **parent-child relationship**. An ordinary pipe is created using the *pipe(int fd[])* syscall, where *fd* is an array containing two file descriptors. The first descriptor is used to **read** data from the pipe, while the second is used to **write** data into it.

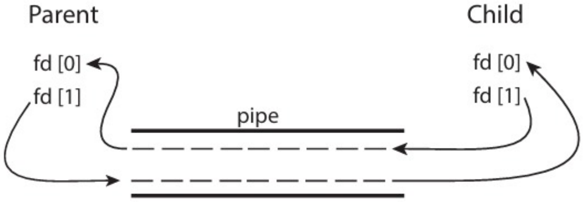


Figure 7: A child process and its parent process communicating using a pipe

In an ordinary pipe, the *parent process* writes data through *fd[1]*, while the *child process* reads data through *fd[0]*. Child processes inherit their parent's file descriptors during creation; this means that after the pipe is set up, both the child and parent have access to both file descriptors.

It is good practice for processes to close their file descriptors, because:

- **Proper EOF (end-of-file) signaling:** the reader (in this case, the child) will only receive an EOF signal once all **write descriptors** referring to the pipe have been closed. If the parent does not close its write descriptor, the reader might remain **blocked** indefinitely.
- **Preventing resource leaks:** every open file descriptor consumes resources; not closing them wastes memory and file table entries.

4.3.2 Named pipes

These are **unidirectional half-duplex** pipes that are used to pass data between unrelated processes. A named pipe exists as a special file in the file system, which allows processes to open it for reading and writing. The data transfer takes place in a **FIFO** manner, ensuring that data are read in the exact order that they were written (which is why named pipes are also called **FIFOs**).

4.4 Sockets

A limitation of pipes is that they provide communication only between processes that exist on the same machine. However, having a **socket** on two different machines can allow them to communicate with each other. **Sockets** are like a doorway for data to travel between applications, either on the same computer or across a network. A socket is defined by a combination of a computer's IP address and a specific port number. They are **bidirectional** and **full-duplex**.

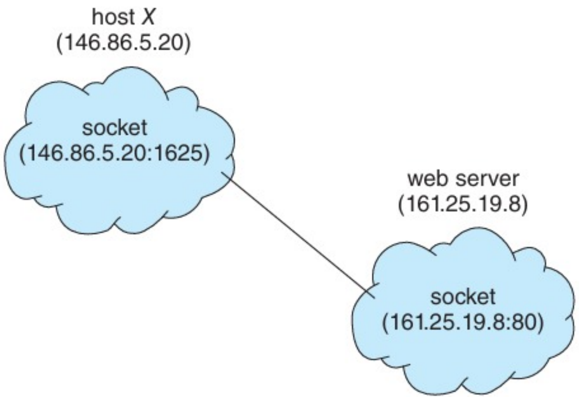


Figure 8: Socket communication