



**POLITECNICO
DI TORINO**

number cols
number rows width
 text

Recap System and device programming (01NYHOV)

Jacopo Nasi
Computer Engineer
Politecnico di Torino

II Period - 2017/2018

6 giugno 2018

Indice

1	Review	5
1.1	Processes	5
1.2	Operating System	6
1.3	Kernel	7
1.4	Shell	7
1.5	Threads	7
1.6	Critical section	7
1.7	Semaphore	8
2	Memory Management	8
2.1	Background	8
2.2	Swapping	10
2.3	Contiguous Allocation	10
2.4	Dynamic Storage-Allocation Problem	11
2.5	Fragmentation	11
2.6	Paging	11
2.7	Memory Protection	13
2.8	Shared Pages	14
2.9	Page Table	14

License

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License.

You are free:

- **to Share:** to copy, distribute and transmit the work
- **to Remix:** to adapt the work

Under the following conditions:

- **Attribution:** you must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work)
- **Noncommercial:** you may not use this work for commercial purposes.
- **Share Alike:** if you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.

More information on the Creative Commons website (<http://creativecommons.org>).



Acknowledgments

Questo breve riepilogo non ha alcuno scopo se non quello di agevolare lo studio di me stesso, se vi fosse di aiuto siete liberi di usarlo.

Le fonti su cui mi sono basato sono quelle relative al corso offerto (**System and device programming (01NYHOV)**) dal Politecnico di Torino durante l'anno accademico 2017/2018.

Non mi assumo nessuna responsabilità in merito ad errori o qualsiasi altra cosa. Fatene buon uso!

1 Review

1.1 Processes

An **Algorithm** is a logical procedure that in a finite number of steps solves problem. A **Program** is a formal expression of an algorithm by means of a programming language. The **Process** is a sequence of operations performed by a program in execution on a given set of input data. The structure of a process, figure 1 is made by:

- Text area (source code)
- Data Area (global variables)
- Stack (functions parameters and local variables)
- Heap (dynamic variables allocated during the process execution)
- Registers (program counter, stack pointer, ecc...)

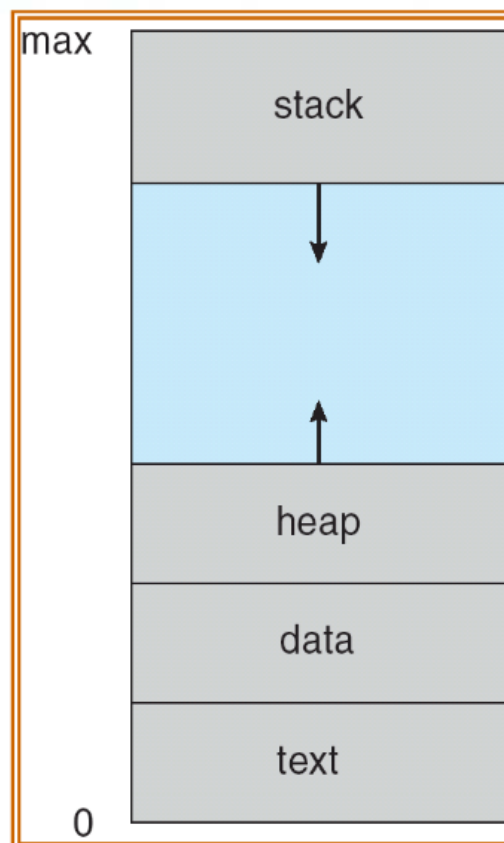


Figura 1: Process memory layout

The trace of a process is its entire line, it represent the state at any time. An important operation is the possibility to freeze the state of a process in order to execute other tasks. This choice is demanded to the CPU scheduler. The possibility to switch from a state to another is called **Context Switching**. Each process has a unique (while the process is active) identifier (**PID**), a positive integer.

The ***fork()*** is the system call used to create a new child process. The child is a copy of the parent excluding the Process ID returned by the fork.

- The parent process receives the child PIDs.
- The child process receives the value 0.

The 2 processes are perfect clones with 2 different heap, 2 stacks, etc... The only shared part is the code. A process can become orphan if the parent dies, in that case it becomes son of the INIT process (PID: 1). Another possible state is the zombie, where the process has terminated its execution but still has an entry in the process state. The zombie state waste resources and must be avoided. During the fork process the child inherits the value of local variable from the parent but they are different variables and they aren't linked, the address space of the 2 processes is different. There are several ways to exit from a process:

- *return*
- *exit*

there are also other not correct termination, like *abort*. The correct behaviour requires that the parent ***wait()*** for the process terminations of its sons, the kernel sends a signal (**SIGCHLD**) to its parent. The parent can manage or ignore it.

The system call ***exec()*** is different from the fork because it runs a different executable, it does not create another process, it substitutes the calling process image with the image of another program.

1.2 Operating System

The OS is not a program, is a set of modules, a big interrupt routine. It reacts to action like mouse moving, etc... The user can't execute all the instruction set. The CPU runs in at least 2 states (mode):

- Kernel
- User

Each mode defines different access rights, of course the kernel mode is the most powerful. There are some instructions that are privileged, like the IO, or over some registers due to concurrency where the control is CPU demanded.

1.3 Kernel

The kernel is a black box, the only way to interact with it is to use the interrupts provided. By receiving and addressing the kernel knows that it must perform, for example, a reading operation and so on. It is not possible to access directly to the kernel memory.

1.4 Shell

The shell is not part of the kernel, is like all other processes. The user performs, through it, system calls that run at the user level.

1.5 Threads

Processes are really "expensive" in case of cooperation, the clone operation involves a significant increase of memory used and the creation time becomes an overhead. Also the context switching can become expensive. A possible solution to all these problems is using **threads**. For the kernel different threads are part of a single process. The context switch is really faster because the context "is the same". The process is the owner of the resources that are used by all its threads. The thread is the basic unit of CPU utilization (and scheduling). They are also called lightweight processes.

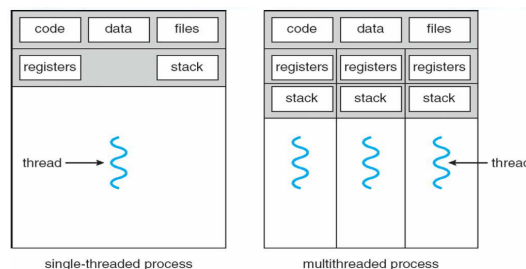


Figura 2: Processes and Threads

1.6 Critical section

Also known as Critical Region, is a section of code, common to multiple threads, in which they can access (read and write) shared objects. A section in which threads are competing for the use (RW) of shared resources. We want to ensure that when one thread is executing in its CS, no other thread is allowed to execute in its CS. The solution is to establish an access protocol to enter the critical section in **mutual exclusion**.

There are different solutions to solve this problem:

- software
- hardware

- System Call (semaphore)

1.7 Semaphore

System call used to manage critical sections and to solve synchronization problem. The semaphore primitives allows thread to:

- Create semaphore (init)
- Be blocked on the semaphore (wait)
- Wakeup if it was blocked (signal)
- Destroy a semaphore (destroy)

The operations on a semaphore are **ATOMIC**, it is impossible for two threads to perform simultaneously operations on the same semaphore.

Another type of semaphore is the MUTEX, or binary semaphore, they are a little bit easier to be managed but they are less powerful. Only the action lock and unlock must be performed.

Semaphores can be also implemented like pipes.

2 Memory Management

Chapter about hardware memory organization, discussing MM techniques like paging and segmentation.

2.1 Background

A program must be brought (from disk) into main memory and placed within a process for it to be run. Main memory and registers are only CPU storage and only it can access directly to them. The register access is performed in one CPU clock, the main memory could require more cycles, the cache instead is between them (speed speaking).

The address binding of instructions and data to memory addresses can happen at three different stages:

- **Compile Time:** If memory location known a priori, absolute code can be generated; must recompile code if starting location changes.
- **Load Time:** Must generate relocatable code if memory location is not known at compile time.
- **Execution Time:** Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps. (e.g. base and limit register)

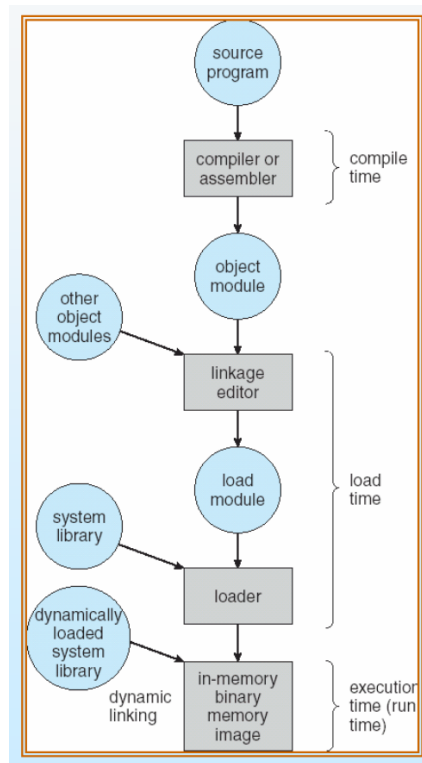


Figura 3: Multistep processing of a user program

The concept of a logical addresses space that is bound to a separata physical address space is central to proper memory management.

- **Logical:** Generated by the CPU; also referred as **virtual addresses**.
- **Physical:** Address seen by the memory unit.

Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme.

The **MMU** (*Memory-Management Unit*) is an hardware device that maps virtual to physical address. In its scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory. Every user program deals only with logical addresses.

Using the **Dynamic Loading** routine is not loaded until it is called, this allows a better memory space utilization (unused routine is never loaded). This function is useful when large amounts of code are needed to handle infrequently occuring cases.

Another important mechanism is the **Dynamic Linking** where the linking is postponed until execution time. Small piece of code, *stub*, used to locate the appropriate memory-resident library routine, replaces itself with the address of the routine, and executes the routine. OS needed to check if routine is in processes memory addresses. This solution is particularly useful for libraries. It is also know as **Shared Libraries**.

2.2 Swapping

A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution.

Backing Store fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images. **Roll out, Roll in** swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed.

The greatest part of swap time is spent in transfer time; total transfer time is directly proportional to the amount of memory swapped. Each system maintains a ready queue of ready-to-run processes which have memory images on disk. AN example of swapping in figure 4.

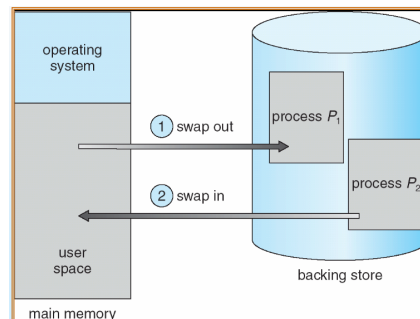


Figura 4: Swapping process

2.3 Contiguous Allocation

Main memory is usually into two parts:

- Resident OS, usually held in low memory with interrupt vector.
- Use processes in the high part.

Relocation registers used to protect user processes from each other, and from changing OS code and data:

- Base register contains value of smallest physical addresses.
- Limit register contains range of logical addresses must be less than the limit register.
- MMU maps logical address *dynamically*.

Multiple-partition allocation is based on **Hole**: block of available memory; holes of various size are scattered throughout memory. When the process arrives, it is allocated memory from a hole large enough to accommodate it. The OS maintains information about:

- Allocated Partitions
- Free Partitions (hole)

2.4 Dynamic Storage-Allocation Problem

There are several solution to how satisfy a request of size n from a list of free holes:

- **First-fit**: Allocate the *first* hole that is big enough.
- **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list (unless ordered by size). It produces the smallest leftover hole.
- **Worst-fit**: Allocate the *largest* hole; must also search the entire list. It produces the largest leftover hole.

First and best fit are better than worst in terms of speed and storage utilization.

2.5 Fragmentation

The **External** is the total memory exists to satisfy a request, but it is not contiguous. The **Internal** instead is the allocated memory that can be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.

Is possible to reduce the fragmentation by using the **compaction**. Shuffle memory contents to place all free memory together in one large block. Compaction is possible only if the relocation is dynamic, and is done at execution time. There are some IO problems:

- Latch job in memory while it is involved in IO.
- Do IO only into OS buffers.

2.6 Paging

The paging is a memory management scheme by which a computer stores and retrieves data from secondary storage for use in main memory.

Logical address space of a process can be non-contiguous; process is allocated in physical memory whenever the latter is available. Divide physical memory into fixed-sized blocks called **frames** (size power of 2, between 512 bytes and 8192 bytes) and keep track of all of them. Divide logical memory into blocks of same size called **pages**. When a program of n pages need to be runned, all the n frames must be found and loaded. Using this solution is important to set up a page table to translate logical to physical addresses. These process of course incurs in internal fragmentation.

Address Translation Scheme divide the addresses generated by the CPU into:

- **Page Number (p)**: Used as an index into a page table which contains base address of each page in physical memory.
- **Page Offset (d)**: Combined with base address to define the physical address that is sent to the memory unit.

Page Number	Page Offset
p	d
m - n	n

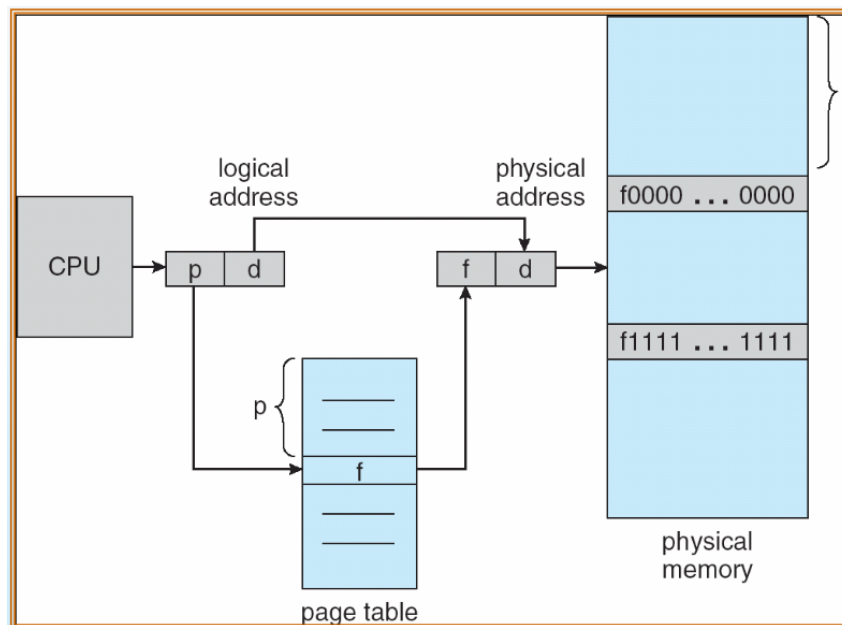


Figura 5: Paging Hardware

Page Table is kept in main memory, it can be of 2 types:

- **PTBR** (Base Register): Points to the page table.
- **PRLR** (Length Register): Indicates size of the page table.

In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction. This double-access problem can be solved by the use of a special fast-lookup hardware cache called **Associative Memory** or **Translation Look-aside Buffers (TLBs)**. Some TLBs store address-space identifiers (ASIDs) in each TLB entry: uniquely

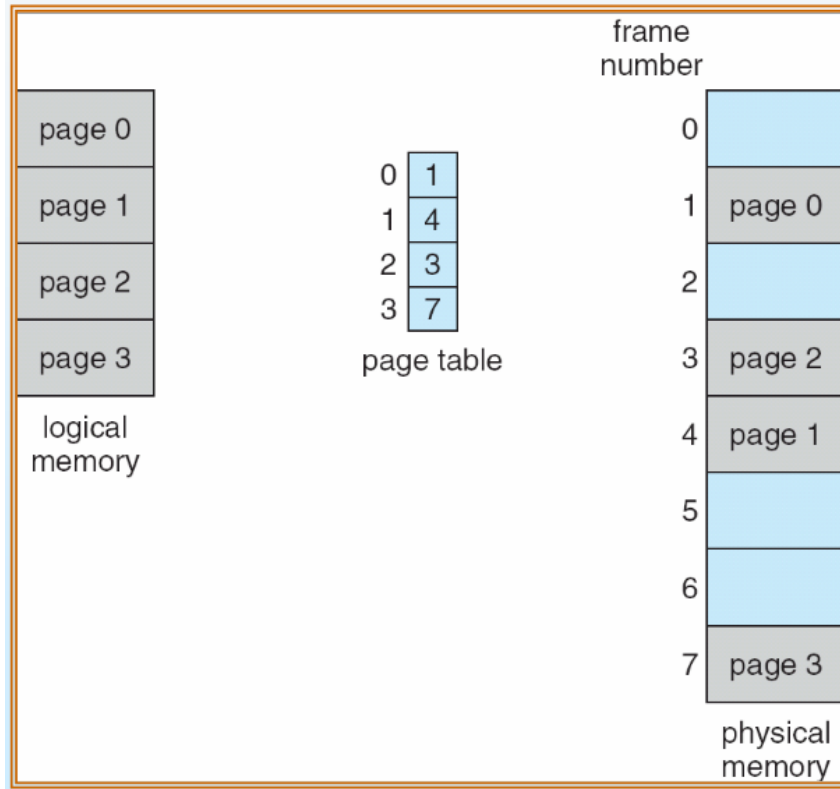


Figura 6: Paging Example

identifies each process to provide address-space protection for that process. In all cases the **Effective Access Time** (EAT) is computed:

$$EAT = (t + \epsilon) * \alpha + (2 * t + \epsilon)(1 - \alpha) \quad (1)$$

where:

- ϵ : Associative lookup time unit
- t : Memory cycle time
- α : Hit ratio

2.7 Memory Protection

The MP is implemented by associating protection bit with each frame. **Valid-Invalid** (means "if is in the code domain") bit attached to each entry in the page table:

- **Valid**: Indicate that the associated page is in the process logical address space, and is thus a legal page.
- **Invalid**: Indicates that the page is not in the process logical address space.

2.8 Shared Pages

The **shared code** is one copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window system). Shared code must appear in same location in the logical address space of all processes. Regarding **private code and data**, each process keeps a separate copy of them. The pages for the private code and data can appear anywhere in the logical address space.

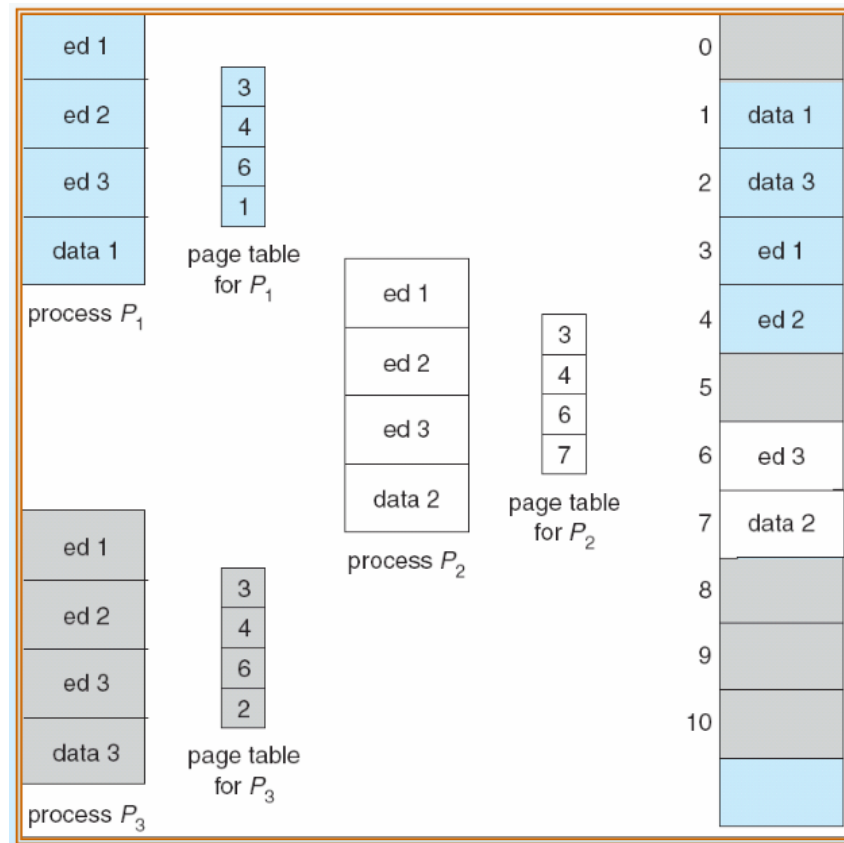


Figura 7: Shared Page

2.9 Page Table

They can be of 3 types:

- Hierarchical
- Hashed
- Inverted

the choice is related to the page size.

The first one, the **hierarchical**, breaks the logical address space into multiple

page tables, a simple technique is a two-level page table. This solution is not actuatable in 64-bit system. A possible example with a Two-Level pagein could be:

- **Page Number** consisting of 22-bit: (since is paged is further divided)
 - 12-bit page number
 - 10-bit offset
- **Page Offset** consisting of 10-bit

Thus the logical address is as follows:

Page Number		Page Offset
p_i	p_2	d
12	10	10

A 3-level architecture is used in order to correctly and completely addressing 64-bit addresses.

The **hashed** architecture is common in address spaces greater than 32 bit. The virtual page is hashed into a page table. This page table contains a chain of elements hashing to the same location. Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical fram is extracted.

The last type, **inverted**, has one entry for each real page of memory. Entry consist of the virtual address of the page stored in that real memory location, with the information about the process that owns that page. Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs. This last solution is affordable only with few memory of course.