

T3-Memory



Index

- Memory management concepts
- Basic Services
 - Program loading in memory
 - Dynamic memory
 - HW support
 - ▶ To memory assignment
 - ▶ To address translation
- Services to optimize physical memory usage
 - COW
 - Virtual memory
 - Prefetch
- Linux on Pentium

Physical memory vs. Logical memory

Process address space

Addresses assignment to processes

Operating system tasks

Hardware support

CONCEPTS

Physical memory vs. Logical memory

- CPU can access only memory and registers
 - Data and code must be loaded in memory to be referenced
 - Program loading: allocate memory, write the executable on that memory and pass execution control to the entry point of the program
- Type of addresses:
 - Reference issued by the CPU: logical address
 - Position in memory: physical address
 - Both addresses may differ if the system (**OS** and **HW**) provides **translation support**
 - ▶ Current systems offer translation support

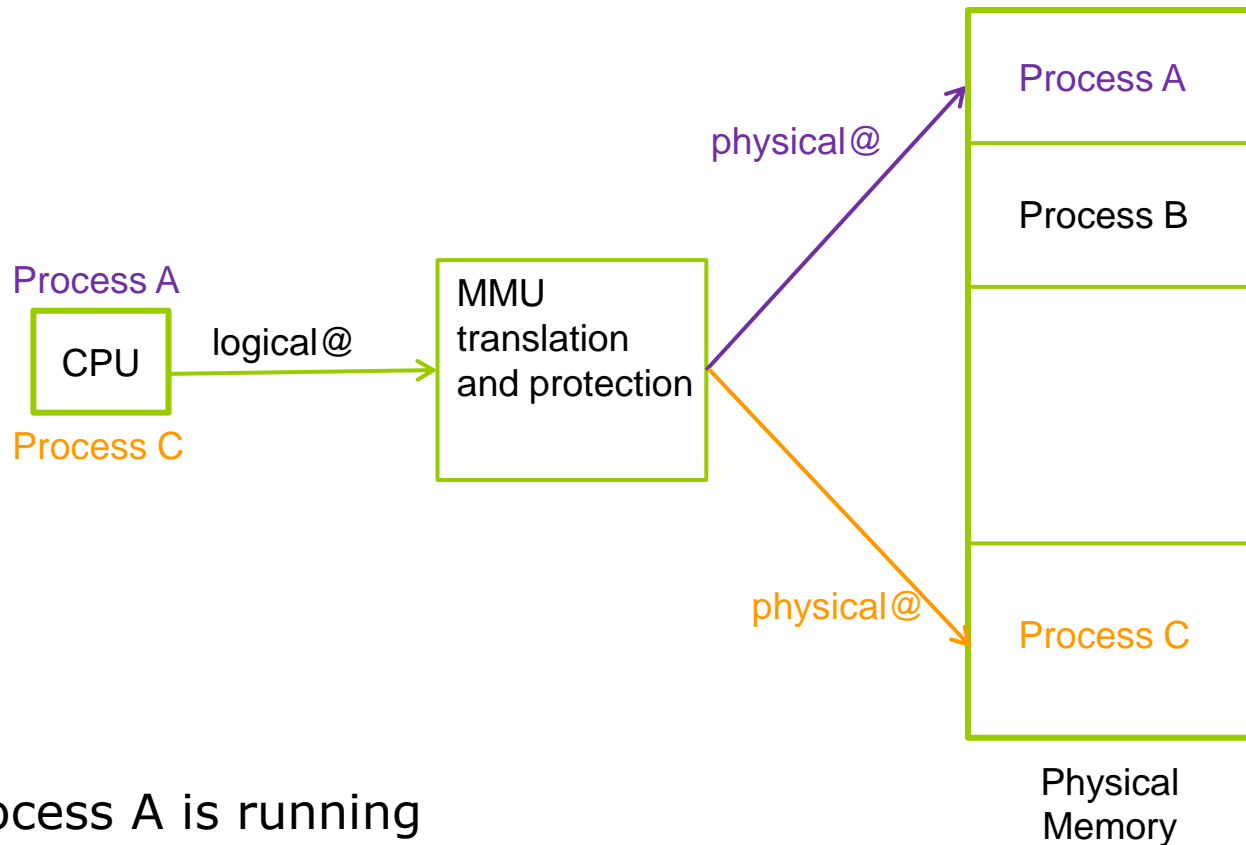
Address Spaces

- **Processor** address space
 - Subset of addresses that the processor can issue (depends on the bus of addresses)
- **Process logical address** space
 - Subset of logical addresses that a process can reference (OS kernel decides which are those valid addresses for each process)
- Process physical address space
 - Subset of physical addresses associated to the process logical address space (decided by the OS kernel too)
- Relationship between logical addresses and physical addresses
 - Fixed: logical address space == physical address space
 - Translation required:
 - ▶ During program loading: kernel decides where to place the process in memory and translate references at program loading
 - ▶ **During program execution: each issued reference is translated at runtime**
 - **Collaboration between HW and OS**
 - » HW offers the translation mechanism
 - » Memory Management Unit (MMU)
 - » **OS kernel configures it**

Multiprogrammed systems

- Multiprogrammed systems
 - Several programs loaded simultaneously in physical memory
 - Ease concurrent execution and simplify context switch mechanism
 - ▶ **1 process on CPU but N processes in physical memory**
 - ▶ When performing a context switch it is not necessary to load again the process that gets assigned the CPU
 - OS must guarantee **physical memory protection**
 - ▶ Each process can access only the physical memory that it gets assigned
 - ▶ **Collaboration between HW and OS**
 - MMU implements the mechanism to detect illegal accesses
 - OS configures MMU
 - Kernel must update MMU according to any change in the running processes:
 - ▶ When performing a context switch, OS updates MMU with the information of the process that gets assigned the CPU
 - ▶ If a process address space grows
 - ▶ etc....

Multiprogrammed systems



- 1-Process A is running
- 2-Context switch to C

Assignment of addresses to processes

- **There exists other choices but... current general purpose systems assign @ to instructions and data at runtime**
 - physical @ != logical @ → requires runtime translation
 - Processes are enabled to change their position in memory without changing their logical address space.
 - ▶ Example: Paging (explained in EC course)

HW support: MMU

- MMU(Memory Management Unit). HW component which at least offers **address translation and memory access protection**. It can also support other management tasks.
- **OS is responsible for configuring the MMU with the correct address translation values for the current process in execution**
 - Which logical @ are valid and which are their corresponding physical @
 - Guarantees that each process gets assigned only its own physical @
- **HW support to translation and protection between processes**
 - MMU receives a logical @ and translates it to the corresponding physical address using its data structures
 - ▶ It throws an exception to the OS if the logical address is not marked as valid or if it has not associated a physical address
 - **OS manage the exception according to the situation**
 - ▶ For example, if the logical address is not valid it can kill the process (SISEGV signal)

HW Support: Translation

- When does the OS need to update the address translation information???
- When assigning memory
 - Initialization when **assigning new memory (mutation, execlp)**
 - **Changes in the address space:** grows/diminishes. When allocating/deallocating dynamic memory
- When switching contexts
 - For the process that leaves the CPU: if it is not finished, then keep in its data structures (PCB) the information to configure the MMU when it resumes the execution
 - For the process that resumes the execution: configure the MMU

HW Support : Protection

- It is performed in the same cases than the memory assignment
- It also enables to implement protection against undesirable accesses/type of accesses
 - Invalid logical addresses
 - Valid logical addresses but wrong type of access (writing on a read-only region)
 - Valid logical address and apparently “wrong” type of access due to some optimization implemented by the OS
 - For example, COW (we will explain it later)
 - For all cases → exception captured by the CPU and managed by the OS
 - OS has all the information about the description of the process address space and can check if the exception is really due to wrong access or not

OS tasks in memory management

- Program loading in memory
- Allocate/Deallocate dynamic memory (requested through system calls)
- Shared memory between processes
 - COW: transparent sharing of read-only regions between processes
 - Shared memory explicitly requested through system calls (out of the scope of this course)
- Optimization services
 - COW
 - Virtual memory
 - Prefetch

Program loading

Dynamic memory

Memory assignment

Shared memory between processes

OS BASIC SERVICES

Basic services: program loading

- Executable file is stored in disk, but it has to be in memory in order to be executed
- OS has to:
 1. Read and interpret the format of the executable file
 2. Prepare in logical memory the process layout and **assign physical memory to it**
 1. **Initialize the process data structures**
 1. Description of the address space
 1. Which logical addresses are valid
 2. Which type of access is valid
 2. Information to configure MMU each time the process resumes the execution
 2. **Initialize MMU**
 3. **Read the program sections** from disk and write them to memory
 4. Load **program counter** register with the address of the **entry point instruction**, which is defined in the executable file

Basic services: program loading

- Optimizations on program loading
 - On-demand loading
 - Shared libraries and dynamic linking
- Program loading in Linux is caused by executing an **exec** system call

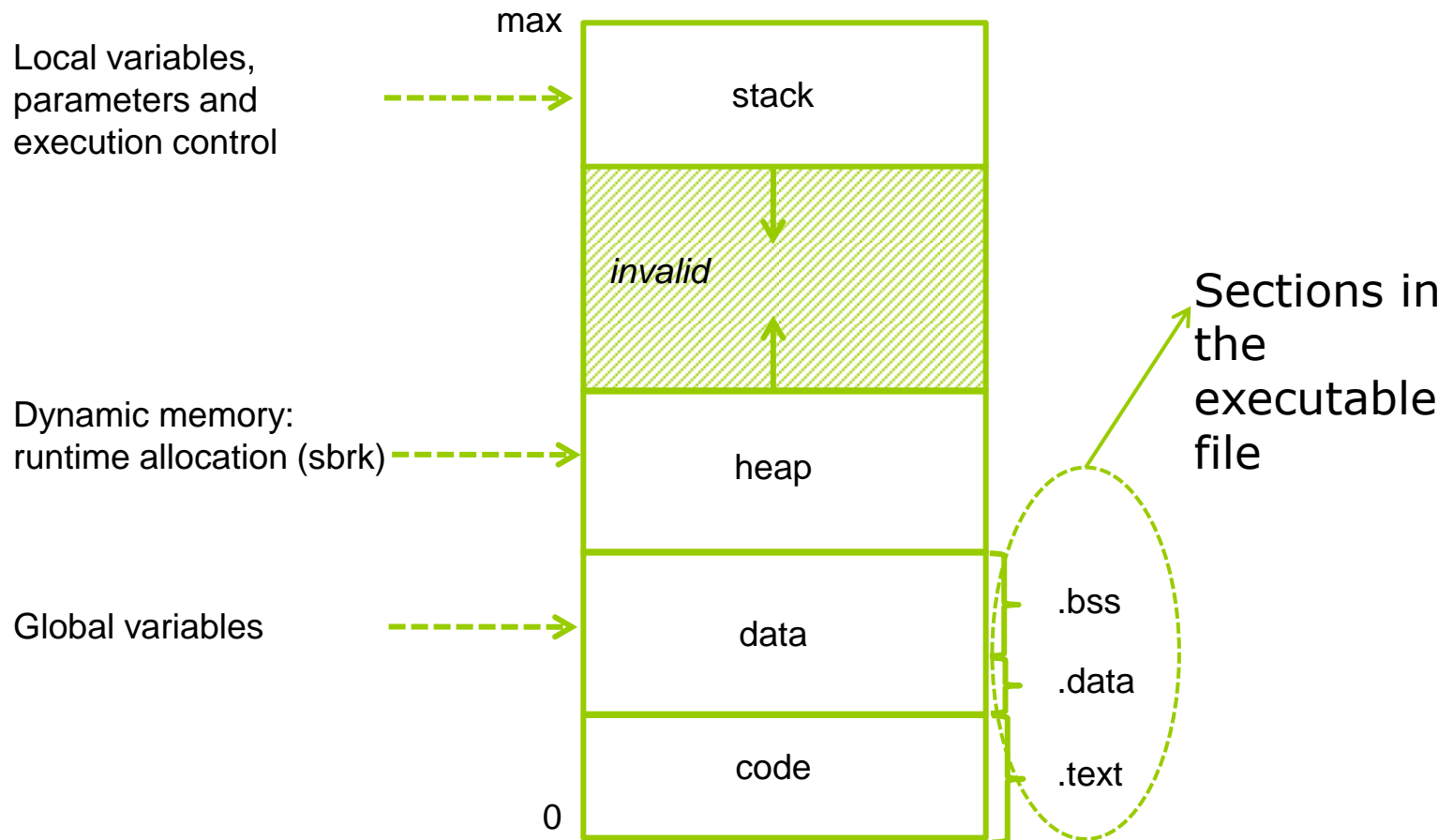
Program loading: executable format

- STEP 1: Interpret executable format in disk
 - If address translation is performed at runtime, which kind of address is in the executable file? Logical or physical?
 - Header in the executable file defines sections: type of section, size and position in the file (try *objdump -h program*)
 - There exists several executable file formats
 - ▶ ELF (*Executable and Linkable Format*): is the most widespread executable format in POSIX systems

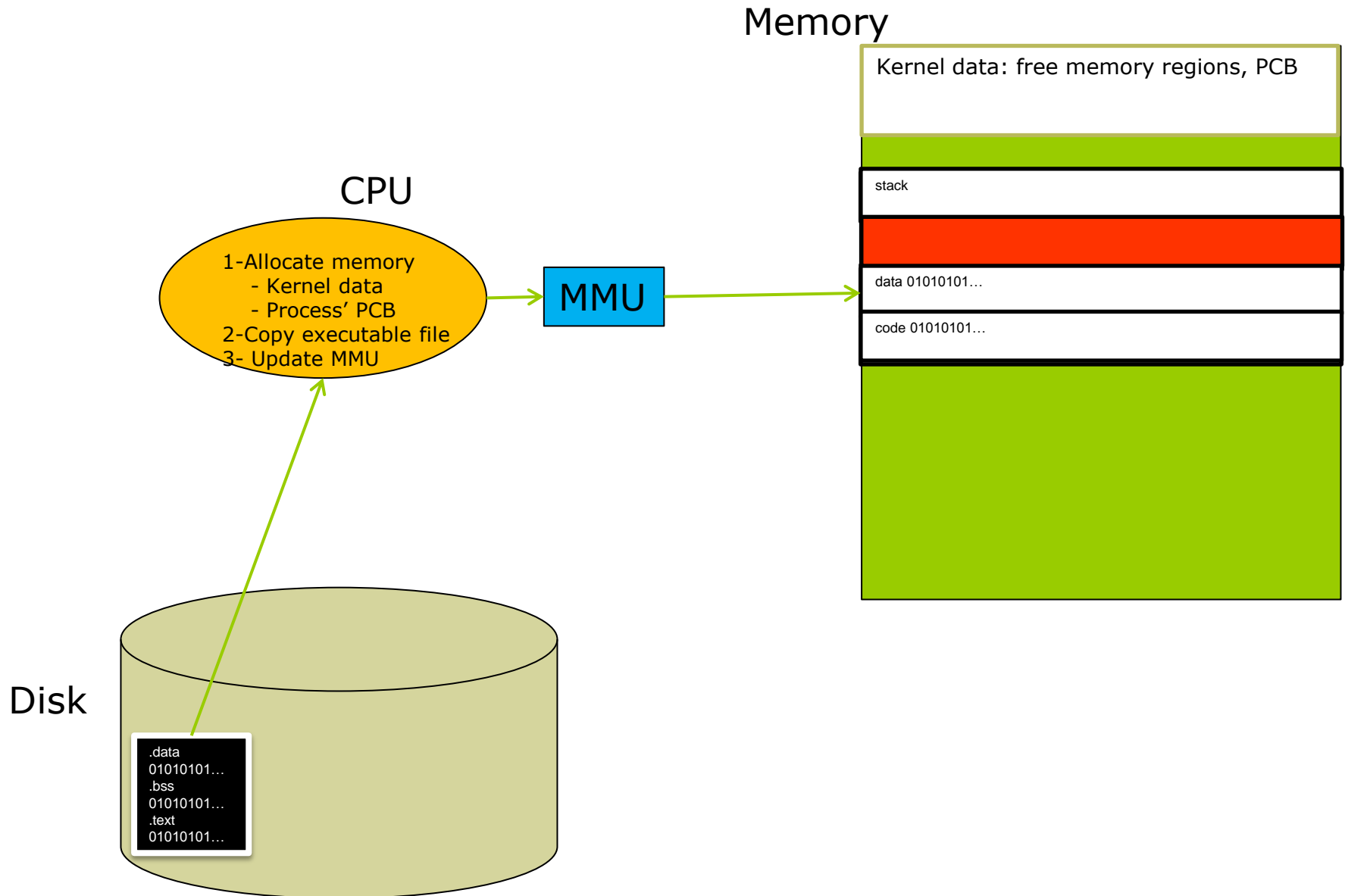
Some sections in ELF files	
.text	Code
.data	Global data with initial value
.bss	Global data without initial value
.debug	Debug information
.comment	Control information
.dynamic	Information for dynamic linking
.init	Initialization code for the process (contains @ of the 1 st instruction)

Program loading: Process layout in memory

- STEP 2: Prepare the process layout in memory
 - Usual layout



Program loading



Program loading: On-demand loading

- Optimization: **on-demand loading**
 - **Loading of routines is delayed until they are called**
 - Avoid memory wasting: those functions that are never used are not loaded in memory (for example, error management functions)
 - Speed-up loading task (but loading time is distributed along the execution of the process)
 - It requires a mechanism to detect if a routine is already in memory or not.
For example:
 - ▶ OS:
 - Registers in its data structures all valid regions and where are their content
 - Leaves empty the translation field in the MMU
 - ▶ When the process references the address, MMU throws an exception to the OS because it is not able to translate the address
 - ▶ OS checks in its data structures that it is a valid access, performs the loading of the involved region and restart the execution of the instruction that caused the exception

Program loading: shared libraries

- Executable files (in disk) do not contain the dynamic library code but just a reference to it
→ **Save disk space**
 - **Link phase is delayed until runtime**
 - How many programs use libC code? How many disk space would be necessary to keep an identical copy of the library for each program that uses it?
- Processes (in memory) can share those memory areas holding the same code (it is read only) and the code of libraries → **Save memory space**
- It makes easy to update programs to use new library versions: at runtime they will be automatically linked with new versions (it is not necessary to compile them again)
- Mechanism
 - Executable file contains a stub routine that:
 - ▶ Loads the library if it is not already loaded in memory(due to a previous request from another process on execution)
 - ▶ Updates the process code to substitute the call to the stub by the call to the routine in the shared library

Dynamic memory allocation/deallocation

- Required when the size of a variable depends on runtime parameters
 - In this situation, it is not desirable to fix sizes at compiling time
 - ▶ Causes over allocation (memory wasting) or under allocation (runtime error)
- OS exports system calls to allocate new memory regions at runtime: **dynamic memory**
 - Heap area: region in the process address space that holds dynamic memory allocations
- Implementation
 - Physical memory assignment can be delayed until the first write access to the region
 - ▶ Temporal assignment of a 0 filled region to manage read accesses. Depending on the definition of the interface, the region will be initialized with 0 or not.
 - Updates MMU according to the memory assignment policy

Dynamic memory allocation/deallocation

■ Linux on Pentium

- Unix traditional interface is not user friendly
 - ▶ `brk` and `sbrk` (we will use `sbrk`)
 - ▶ Both system calls just update the heap limit. OS does not control which variables are store in the heap, it just increases or decreases heap size.
 - ▶ Programmer is responsible of controlling the position of each variable in the heap → complex task

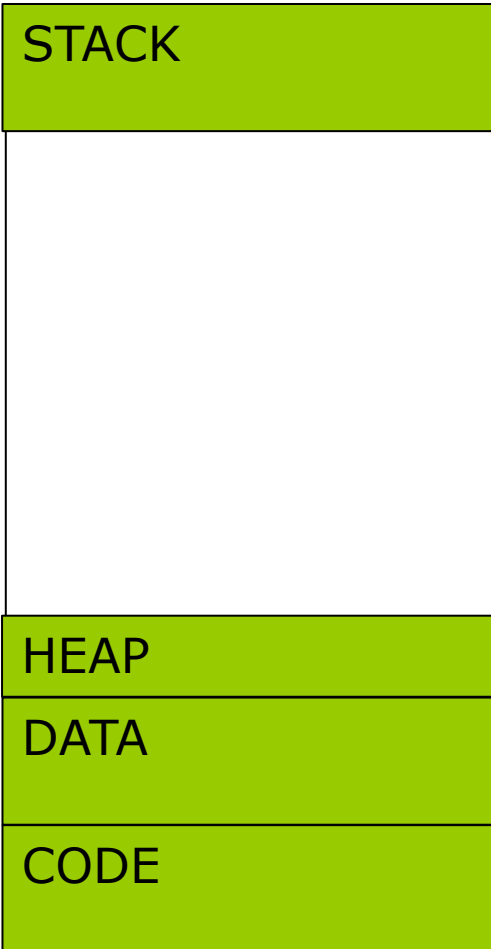
```
former_heap_limit(int) sbrk(size);
```

- ▶ Returns: former heap limit
- ▶ size is an integer
 - `>0` increases the heap limit by size bytes
 - `<0` decreases the heap limit by size bytes
 - `==0` it does not modify the heap limit

Sbrk:example

```
int main(int argc, char *argv[])
{
    int procs_nb = atoi(argv[1]);
    int *pids;
    pids = sbrk(procs_nb * sizeof(int));
    for(i=0; i<10; i++){
        pids[i] = fork();
        if (pids[i] == 0){
            ....
        }
    }
    sbrk(-1 * procs_nb * sizeof(int));
}
```

max



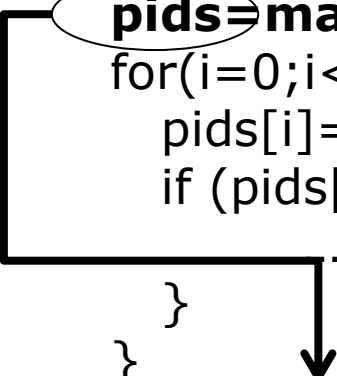
If we have only a single variable it is a very simple code. Let's consider the situation with several dynamically allocated variables, how can we deallocate a variable in the middle of the heap?

Dynamic memory allocation/deallocation

- C library enables the association between variables and position in the heap
 - This association is transparent to the OS kernel
- **C library interface.** Memory allocation: `malloc(size_in_bytes)`
 - ▶ If there is enough **consecutive** space in the heap, it is marked as allocated and returns the initial address.
 - ▶ If there is not enough consecutive space, it requests to the OS to increase the heap size
 - ▶ C library manages the heap, recording which regions are free due to user deallocations. It optimizes the allocation code reducing the amount of system calls performed
 - When it is necessary to increase the heap limit, it request for more memory than the actual allocation required. The goal is trying to satisfy the next allocation request without performing a system call
- **C library interface.** Memory deallocation: `free(memory_region)`
 - ▶ Input parameter is just the initial address returned by a previous call to `malloc` (C library keeps in its data structures the size for each allocated region)
 - ▶ C library keeps lists of free regions in the heap. When the programmer deallocates a region, C library adds it to a free region list and decides if it is suitable to reduce the heap size

malloc/free: example

```
int main(int argc, char *argv[])
{
    int procs_nb = atoi(argv[1]);
    int *pids;
    pids = malloc(procs_nb * sizeof(int));
    for(i=0; i<10; i++){
        pids[i] = fork();
        if (pids[i] == 0){
            ..
        }
    }
    free(pids);
}
```



malloc interface like sbrk interface.
free interface needs as input parameter
a pointer to the base address of the region

Dynamic memory: examples

- How does the heap change after executing the following examples?

- Example 1:

```
...  
new = sbrk(1000);  
...
```



- Example 2:

```
...  
new = malloc(1000);  
...
```



- Does the heap size change in both examples?

Dynamic memory: examples

- How does the heap change after executing the following examples?

- Example 1:

```
...  
ptr = malloc(1000);  
...
```



- Example 2:

```
...  
for (i = 0; i < 10; i++)  
    ptr[i] = malloc(100);  
...
```



- Do both examples allocate the same logical memory addresses?
 - Example 1: requires 1000 consecutive bytes
 - Example 2: requires 10 regions of 100 bytes each one

Dynamic memory: examples

- Which errors are in the following codes?

- Code 1:

```
...  
for (i = 0; i < 10; i++)  
    ptr = malloc(SIZE);  
  
// uso de la memoria  
// ...  
  
for (i = 0; i < 10; i++)  
    free(ptr);  
...
```



- Code 2:

```
int *x, *ptr;  
  
...  
ptr = malloc(SIZE);  
...  
x = ptr;  
...  
free(ptr);  
  
sprintf(buffer, "...%d",  
        *x);
```



- Code 1: What does happen while executing the second iteration of second loop?
- Code 2: Does the access to “*x” produce always the same error?

Basic services: memory assignment

- It is executed each time a process needs physical memory:
 - In Linux: creation (fork), load of executable files (exec), dynamic memory usage, implementation of some optimization (on-demand loading, virtual memory, COW...).

- Steps
 - **Select free physical memory** and mark it in the OS data structures as in-use memory
 - Update MMU with the mapping information logical @ → physical @
 - ▶ Necessary to implement **address translation**

 - **Problems: fragmentation**

Memory assignment: fragmentation

- **Fragmentation problem:** when it is not possible to satisfy a given memory request although the system has enough memory to do it. There are free memory but cannot be assigned to a process.
 - It appears in the disk management too
- **Internal fragmentation:** memory assigned to a process that is not going to use it.
- **External fragmentation:** free memory that cannot be used to satisfy any memory request because it is not contiguous.
 - It can be avoided compacting the free memory. It is necessary the system to support address translation at runtime.
 - ▶ Slows down applications

Basic services: memory assignment

- First approach: **contiguous assignment**
 - Physical address space is contiguous
 - ▶ The whole process is loaded on a partition which is selected at loading time
 - It is not flexible and complicates to apply optimizations (as, for example, on-demand loading)
- **Non-contiguous assignment**
 - Physical address space is not contiguous
 - Flexible
 - Increases granularity to the memory management of a process
 - Increases complexity of OS and MMU
- Based on
 - **Paging** (fixed partitions)
 - **Segmentation** (variable partitions)
 - Combined schemes
 - ▶ For example, paged segmentation

explained in EC course

Assignment: Paging

■ Paging based scheme

- Logical address space is divided into fixed size partitions: **pages**
- Physical memory is divided into partitions of the same size: **frames**
- Assignment
 - ▶ For each page look for a free frame
 - List of free frames
 - ▶ Can cause internal fragmentation
- The frames assigned to a process go back to the free frames list when the process ends the execution
- **Page: working unit of the OS**
 - ▶ Facilitates on-demand loading
 - ▶ Enables page-level protection
 - ▶ Facilitates memory sharing between processes
 - ▶ Usually, a page belongs to just one memory region to match region protection requirements (code/data/heap/stack)

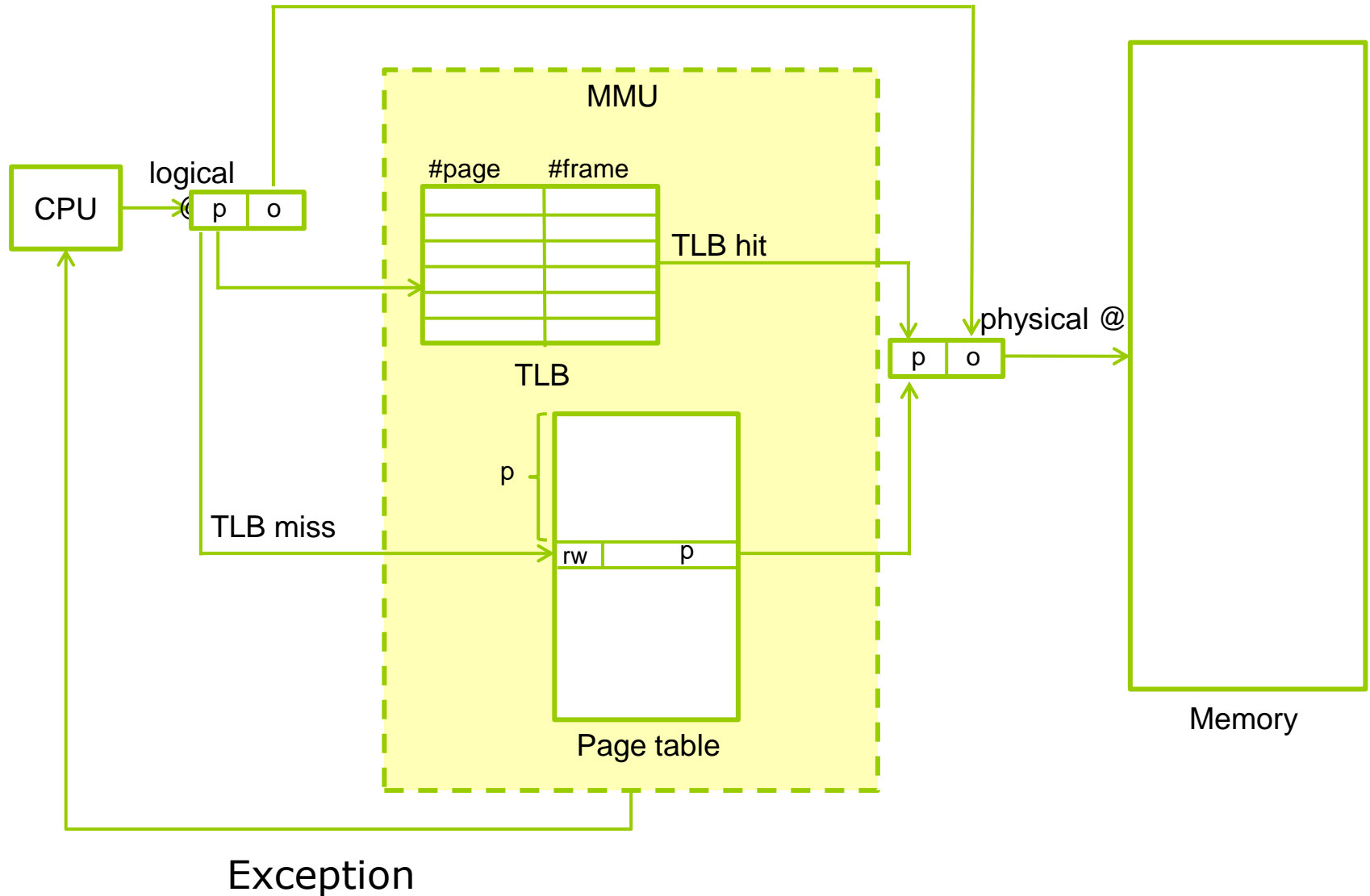
Assignment: Paging

■ MMU

● Page Table

- ▶ To keep page-level information: validity, access permissions, associated frame, etc.
- ▶ One entry per page
- ▶ One table per process
- Usually it is stored on memory. OS needs to keep the base address of the page table of each process (for example, in each PCB)
- Current processors also have a TLB (*Translation Lookaside Buffer*)
 - ▶ Associative memory (cache) of **faster access** than RAM to keep translation for *active pages*
 - ▶ It is necessary to update/invalidate TLB for each change in the MMU
 - Hardware management / Software management (OS)
 - Dependent on the architecture

Assignment: Paging



Assignment: Paging

- PROBLEM: Page table size (stored in memory)
- Page size is usually power of 2
 - Typical size 4Kb (2^{12})
 - Affects to
 - ▶ Internal fragmentation and management granularity
 - ▶ Page table size
- Scheme to reduce memory needed by PT: multi-level PT
 - PT is divided into section and more sections are added as process address space grows

	Logical address of the processor	Number of pages	PT size
32 bits Bus	2^{32}	2^{20}	4MB
64 bits Bus	2^{64}	2^{52}	4PB

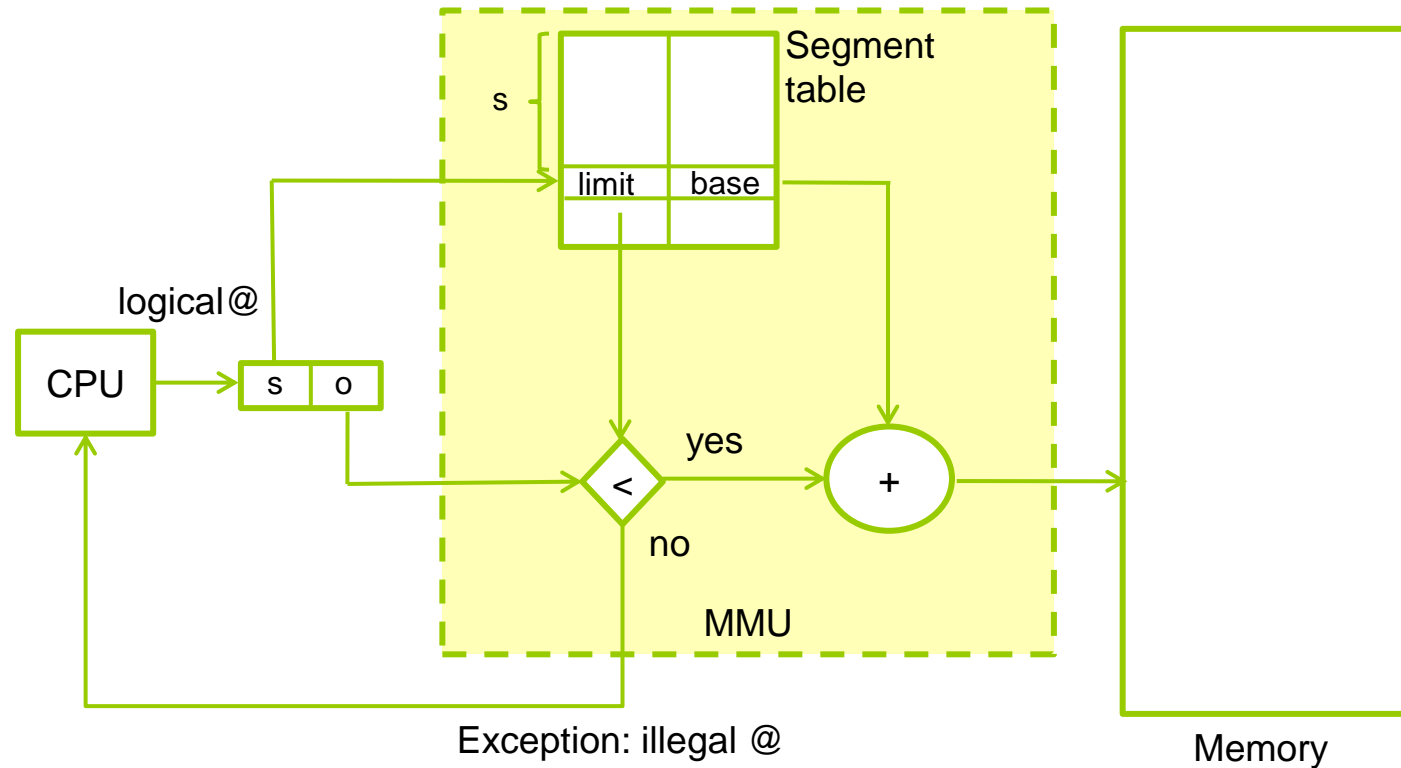
Assignment: Segmentation

■ Segmentation based scheme

- Logical address space divided into regions, considering the type of content (code, data, etc.)
 - ▶ Approximates memory management to user
- Logical address space divided into variable size partitions (**segments**), that fit the size that is really need
 - ▶ At least 3 segments: one for code, one for stack and one for data
 - ▶ References to memory are composed of segment and offset
- All physical contiguous memory is an available partition
- Assignment: for each segment in a process
 - Look for a partition big enough to hold the segment
 - Possible policies: first fit, best fit, worst fit
 - Select from the partition just the amount of memory needed to hold the segment and the rest of the partition is kept in a free partitions list
 - ▶ Can cause external fragmentation

Assignment: Segmentation

- MMU
 - Segment table
 - ▶ For each segment: base @ and size
 - ▶ One table per process



Assignment: Esquema mixto

■ Mixed schemes: paged segmentation



- Logical address space is divided into segments
- Segments are divided into pages
 - ▶ Segment size is multiple of page size
 - ▶ Page is OS working unit

Basic services: sharing

- Memory sharing between processes
 - Specified at page-level or segment-level
 - Useful for processes executing the same code: it is not necessary to keep several copies in physical memory (read-only access)
 - ▶ **Shared libraries (implicit)**
 - Useful as a method to share data between processes
 - ▶ Shared memory as a **communication mechanism between processes (explicit)**
 - ▶ OS must provide programmers with system calls to manage shared memory regions: allocate memory regions and mark them as sharable, thus other processes can map them into their address space
 - Rest of the address space of a process is **private**

COW

Virtual Memory

Prefetch

SERVICES TO OPTIMIZE PHYSICAL MEMORY USAGE

Optimizations: COW (Copy on Write)

- Goal: to delay allocation/initialization of physical memory until it is really necessary
 - If a new zone is never accessed → it is not necessary to assign physical memory to it
 - If a copied zone is never written → it is not necessary to replicate it
 - Save time and memory space
- Example: fork
 - **Delays copy of each region (code, data, etc.) until it is accessed for writing**
 - Avoids physical copy for those regions that are only read (for example, code region)
 - It is usually implemented at page-level: frames are allocated/copied when pages are accessed to write
- It can be applied
 - In the address space of one process: dynamic memory case
 - Between processes: fork case

COW: Implementation

- **Overview: OS needs a mechanism to detect writes and to perform the physical memory allocation and the copy**
- When the logical memory region is allocated:
 - OS registers in the MMU the new region with the same physical memory than the source region
 - OS registers the new region in the data structure describing the address space of the process (in the PCB), indicating which are the real permissions of access
 - OS marks in the MMU both new region and source region as read-only regions
- When a process tries to write on the new region or on the source region:
 - MMU throws an exception to the OS. OS management code performs the actual allocation and copy, updates MMU with the real permission for both regions and resets the instruction

COW: example

- Process A physical memory assignment:
 - Code: 3 pages, Data: 2 pages, Stack: 1 page, Heap: 1 page
- Let's consider that process A executes a fork system call. Just after fork:
 - Total physical memory:
 - ▶ Without COW: process A= 7 pages + child = 7 pages = 14 pages
 - ▶ With COW: process A= 7 pages + child = 0 pages = 7 pages
- Later on the execution... depends on the code executed by the processes, for example:
 - If child executes an exec (and its new address space uses 10 pages):
 - ▶ Without COW: process A= 7 pages+ child = 10 pages= 17 pages
 - ▶ With COW: process A= 7 pages+ child A=10 pages= 17 pages
 - If child does not execute an exec, at least code will be always shared between both processes and the rest of the address space depends on the code. If only the code is shared:
 - ▶ Without COW: process A= 7 pages+ child A= 7 pages= 14 pages
 - ▶ With COW: process A= 7 pages+ child A=4 pages= 11 pages
- **In general, in order to compute the amount of required physical memory it is necessary to compute how many pages are modified (and thus cannot be shared) and how many pages are read-only (and thus can be shared)**

Optimizations: Virtual memory(I)

- Virtual memory
 - Extension for the on-demand loading optimization
 - In addition to load pages on-demand, it enables the system to take out pages that are not needed at a given time
 - Goal
 - ▶ To reduce amount of physical memory assigned to a process
 - **A process only needs physical memory to hold the current instruction and the data that this instruction references**
 - ▶ To increase potential multiprogramming grade
 - Amount of concurrent processes

Optimizations: Virtual memory (II)

■ First approach: swapping

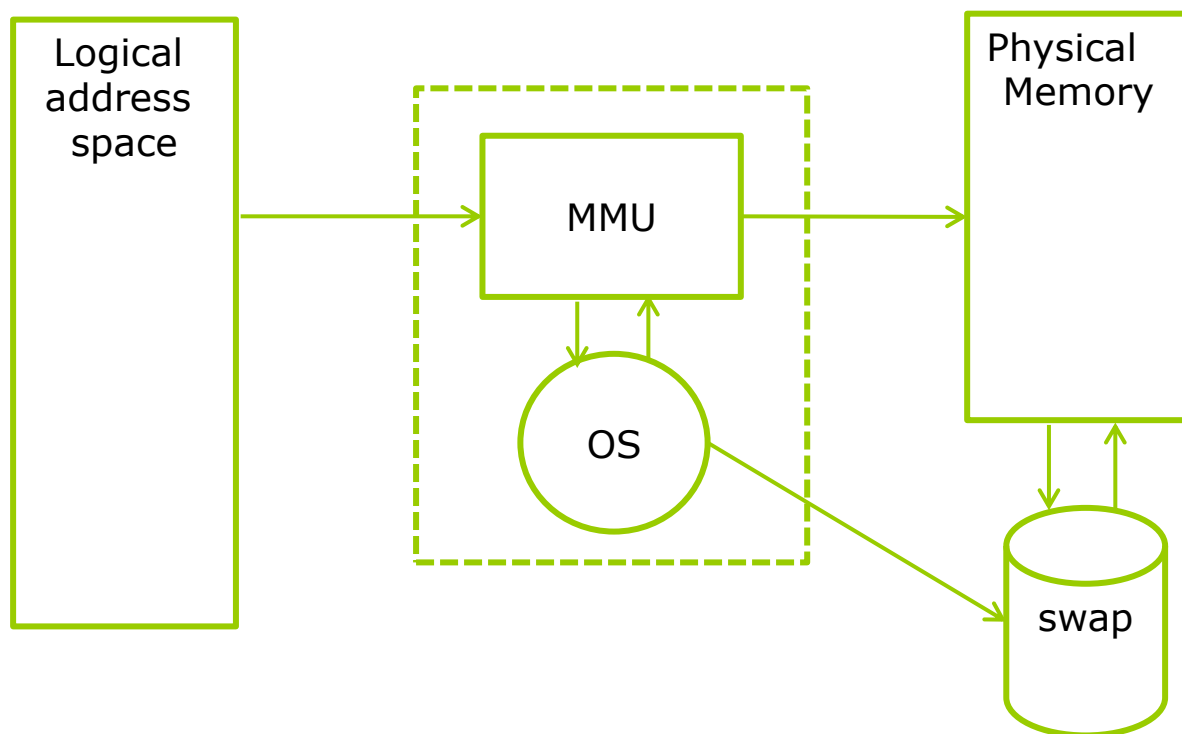
- Overview: it is necessary to keep in memory just only the active process (that with the CPU assigned)
 - ▶ If there is not enough free physical memory to hold the active process, OS takes out from memory temporarily other already loaded process (swap out)
 - ▶ Backing storage:
 - Storage device that can keep logical address spaces of those process that are waiting for the CPU
 - » Bigger size than physical memory
 - It is usually a region in the disk: swap area
 - ▶ New process state: swapped out
 - ▶ When the system resumes the execution of a swapped-out process it is necessary to load it again
 - It slows down the process execution

■ Next approach

- To reduce penalty caused by reloading a process when it resumes the execution: if free memory is needed to avoid swapping out a whole process
- Based on granularity supported by paging

Optimizations: Virtual memory (III)

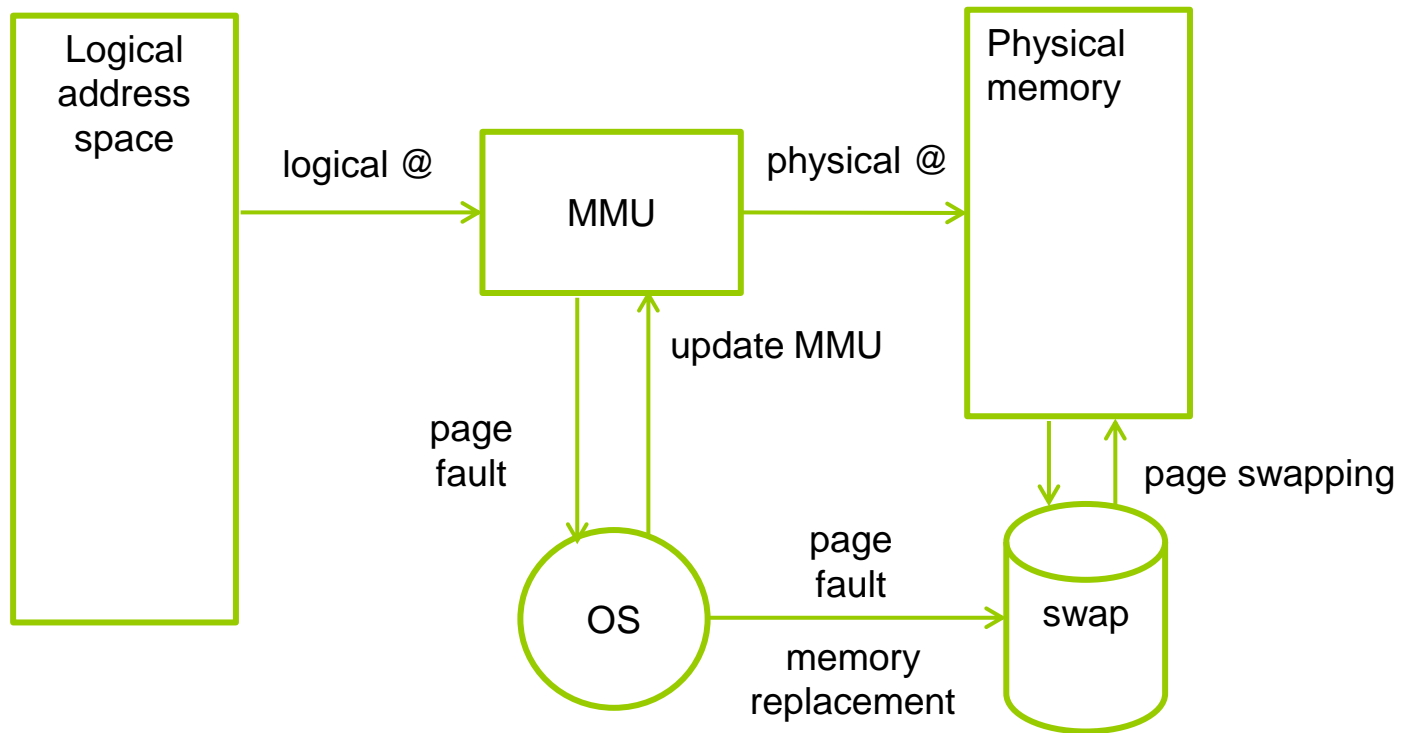
- Virtual memory based on paging
 - **Logical address space of a process is distributed across physical memory (present pages) and swap area (non-present pages)**



Optimizations: Virtual memory(IV)

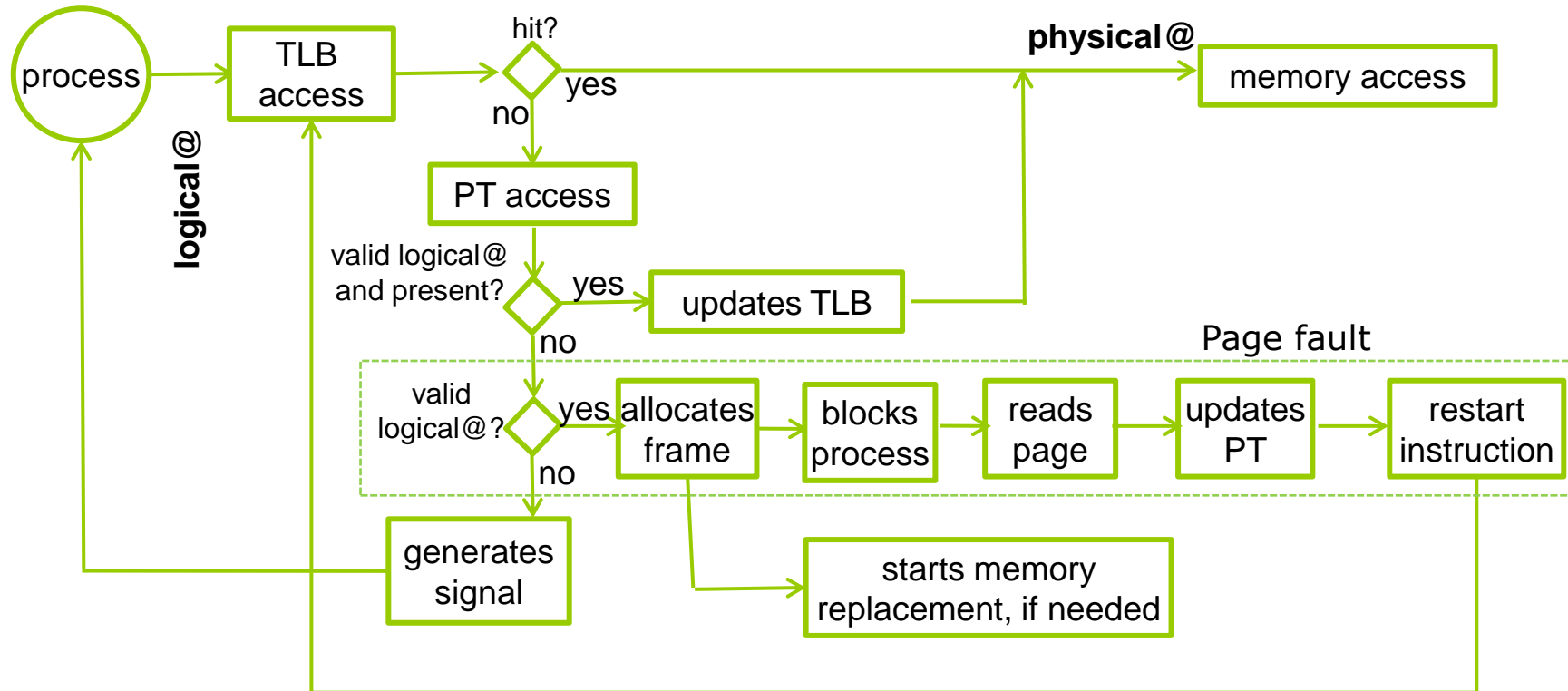
- **Memory replacement:** executed when OS needs to free frames
 - Selects a *victim page* and deletes its translation from MMU
 - Stores its contents in the swap area
 - Assigns the free frame to that page that requires it
- When a non-present page is referenced
 - MMU throws an exception to the OS as it cannot perform the translation
 - ▶ **Page Fault**
 - OS
 - ▶ Checks if the access is valid (accessing data structures of the process)
 - ▶ Assigns a free frame to the page (starts the memory replacement algorithm if it is necessary)
 - ▶ Searches for the content of the page in the swap area and writes it into the selected frame
 - ▶ Updates MMU with the physical address assigned

Optimizations: Virtual memory (V)



Optimizations: Virtual memory (VI)

■ Memory access steps:



Optimizations: Virtual memory (VII)

- Effects of using virtual memory:
 - Physical memory can be smaller than the sum of the address spaces of the loaded processes
 - Physical memory can be smaller than the logical address space of a single process
 - Accessing to non-present pages is slower than accessing to present pages
 - ▶ Exception + page loading
 - ▶ It is important to reduce the number of page faults

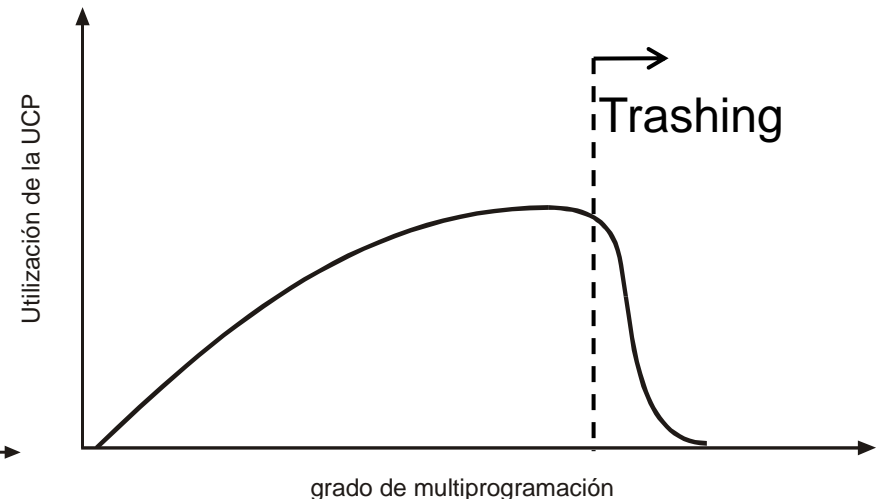
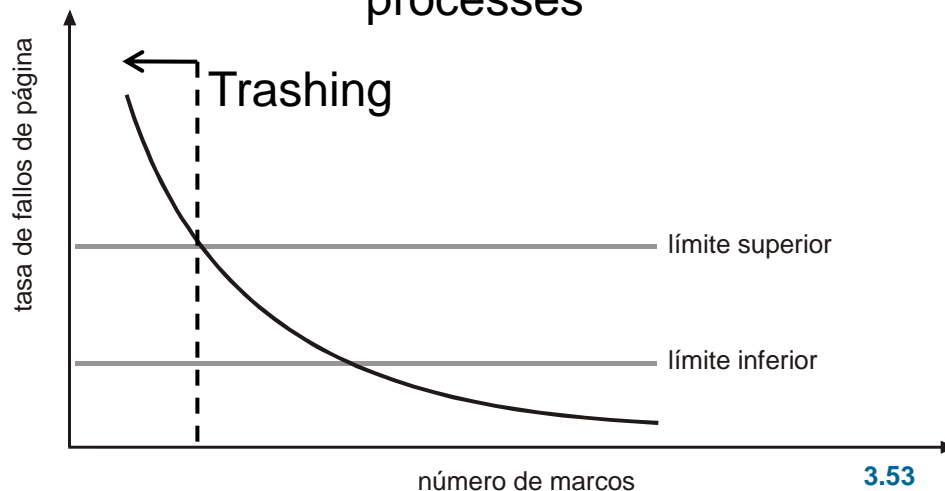
Optimizations: Virtual memory (VIII)

- Changes needed in OS
 - To add data structures and algorithms to manage swap area
 - ▶ Allocation, deallocation and access
 - **Memory replacement algorithm**
 - ▶ When should be executed? Criteria to select victim pages? How many victim pages per algorithm execution?
 - ▶ Goal: to minimize number of page fault and to accelerate page fault management
 - **Try to select victim pages that are no longer necessary or that will take long time until be needed**
 - » Example: Least Recently Used (LRU) or approximations
 - Try that for each page fault will be always an available frame to solve it
- Changes needed I MMU: depends on the virtual memory management algorithms
 - For example, replacement algorithm may need a referenced bit per page

Optimizations: Virtual memory (IX)

■ Thrashing

- Process is in thrashing when
 - ▶ **It spends more time performing page swapping than executing program code**
 - ▶ It is not able to keep simultaneously in memory the minimum number of pages required to advance with the execution.
- Memory system is overloaded
 - ▶ Detection: to control page fault rate per process
 - ▶ Management: to control multiprogramming grade and to swap out processes



Optimizations: Prefetch

- Goal: to minimize number of page faults
- Overview: to predict which pages will need a process and load them in advance
- Parameters to consider:
 - Prefetch distance: time between the page loading and the page reference
 - Number of pages to load in advance
- Some simple prediction algorithms:
 - Sequential
 - Strided

Summary: Linux on Pentium

- **exec** system call: **loads** a new program
 - PCB initialization with the description of the new address space, memory assignment,...
- Process creation (**fork**):
 - PCB initialization with the description of its address space (which is a parent copy)
 - Uses COW
 - Creation and initialization of the new process PT
 - ▶ Base address of the PT is kept in the PCB of the process
- Process scheduling
 - **Context switch**: updates MMU with the base address of the current PT and invalidates TLB
- **exit**:
 - Deletes process PT and deallocates process frames (if those frames are not in use by other process)

Summary: Linux on Pentium

- Virtual memory based on paged segmentation
 - Multi-level page table (2 levels)
 - ▶ One per process
 - ▶ Stored in memory
 - ▶ A CPU register keeps the base address of the PT for the current process
 - Memory replacement algorithm: LRU approximation
 - ▶ Executed periodically and each time the number of free frames reach a threshold
- COW at page level
- On-demand loading
- Support to shared libraries
- Simple Prefetch (sequential)

Storage hierarchy

