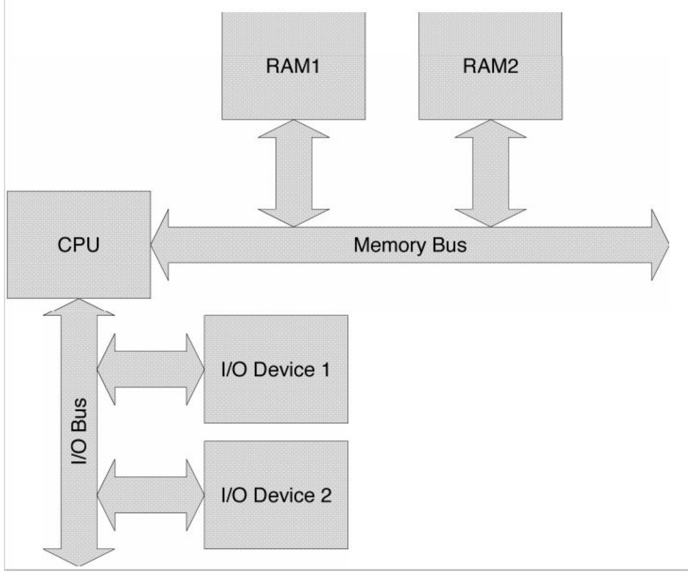
DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

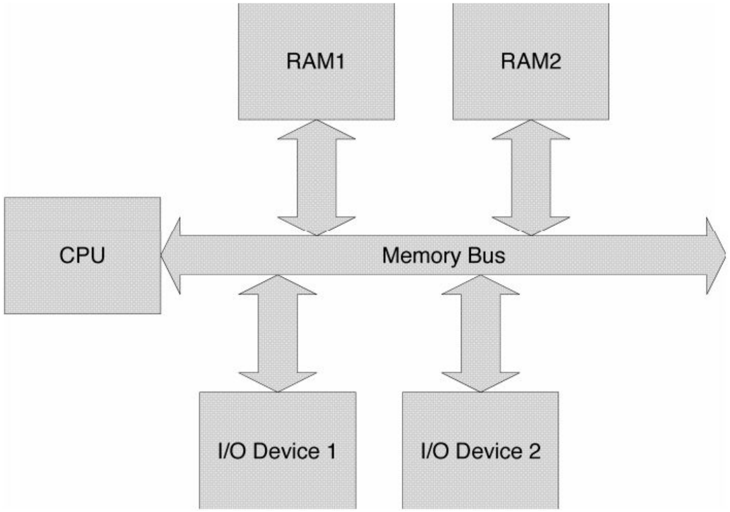
Operating Systems review

- I/O operations - Communication buses
- I/O synchronization - Polling, Interrupts, DMA
- Kernel and user mode
- Virtual Memory
- Memory cache
- Pipelining

Input/Output (I/O) in computers

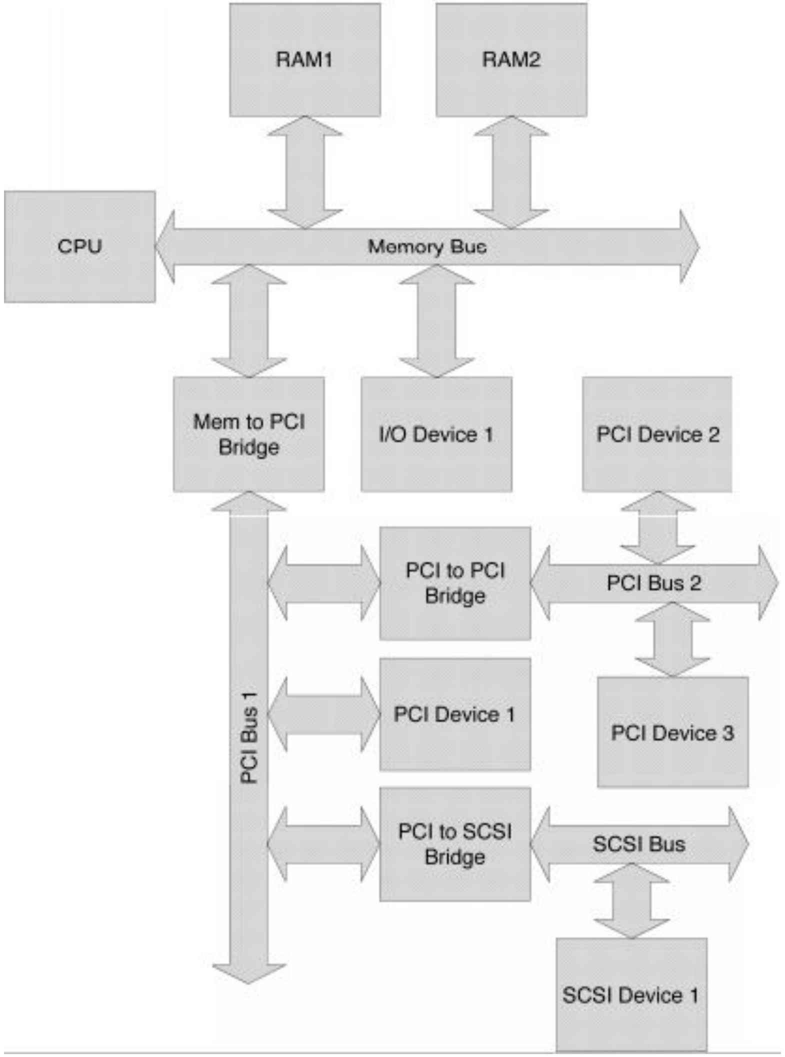
- I/O requires a **Communication bus**
  - Data lines
  - Address lines
  - Handshaking lines
- A separate bus for I/O may be defined, different from the bus used for memory access
- Most computer architectures define a common bus for memory and I/O operation (Memory mapped I/O)
- In memory mapped I/O the address range for connected devices must be separate from the address range for memory
- Every device shall present itself as a set of **registers** at given ‘memory’ addresses
- Every register shall carry out a given functionality that depends on the given device type
- In general devices shall define one or more
  - status registers, whose bits bring information on the current device statis;
  - command registers that, when written shall issue commands to the device
  - Configuration registers
  - Data registers





PCI/PCI-e bridges

- A common bus for I/O in computers i the Peripheral Component Interconnect (PCI) bus.
  - I is a parallel bus. This means that the same bus lines are shared among the connected components
  - At most 2 devices can be connected to the same PCI bus
  - At system startup, devices connected shall be requested to provide the required address range(s)
- PCI express (PCIe) represents an evolution of PCI using fast serial links in place of the shared parallel bus.
  - PCIe is further characterized by the number of lanseused for serial communication(×1, ×2, ×4, ×8, ×12, ×16 and ×32)
- PCI and PCIe buses are then connected to the memory bus via **bridges**
- Every bridge recognizes a bus access in the assigned range and translates it in a new bus cycle on the connected bus



Synchronizing I/O

- In order to perform correct I/O operations, the computer needs to know when the device is ready to provide/consume a new data item.

- The simplest method is to define a bit in the status register that specifies whether the device is ready

- The computer will repeatedly read the status register until it detects that the device is ready, and then it will read/write the corresponding data register.

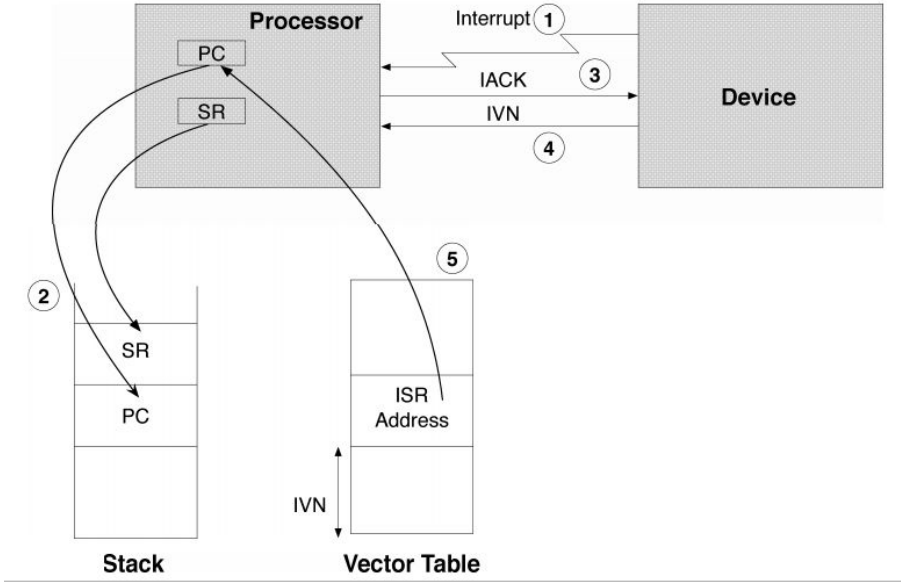
- This method however may lead to a waste of computer resources
  - For example a 9600 bus serial communication link can reach a maximum 1200 Byte/s rate and therefore will provide a new byte at least every 0.83 millis
  - Assuming that 100 ns are required for a memory access (a raw estimate not taking into account cache effects) this means that on average 8000 read operation are performed on the status register before being able to exchange a byte, this wasting 99.99% CPU time

- For this reason other techniques exist that allow the CPU perform independent work while the device is processing I/O

- In this case a new mechanism is required in order to inform asynchronously the availability of the device.
  - This implies new handshaking lines in the communication bus and new hardware/Software functionality implemented in the CPU and Operating System (OS)

Interrupt management

- When a device needs to communicate an asynchronous event it raises one of the **interrupt lines** defined in the communication bus
- Interrupts are ordered by priority and a separate line is normally defined for each priority
- In order to specify which action to be taken, a number shall be transmitted along the bus via an **Interrupt Acknowledge Cycle (IACK)**
- This number shall be used as an offset in the Interrupt Vector Table in order to retrieve the address of the first machine instruction of the **Interrupt Service Routine (ISR)**
- Before executing the ISR it is however necessary to save the context of the program (possibly a ISR for a lower priority interrupt).
- The HW normally saves The Program Counter (PC) and the Status Register (SR) on the stack. The ISR shall then save the used CPU registers.



Required SW actions for properly handling Interrupts

- The SW device driver is the code for managing a given device. This code is not directly accessible to users, but shall be plugged within the OS

- Among other actions, the driver shall specify and ‘connect’ the code to be executed when the given device issues an interrupt.

- During device installation (normally at system boot) the driver shall update the vector table by inserting the ISR address and communicating to the device the ISR number to be returned when a IACK is issued during interrupt (normally a register is devoted to this information)..

- It is worth noting that during the execution of a ISR no other activity (including those carried out by the OS) can be executed.

- Interrupts are heavily used by the OS itself in order to be able to take control of the computer regardless the operation carried out by user code (such as an infinite loop)

- In normal operation, on average, hundreds or even thousands of interrupts per second occur

Are Interrupts an efficient way of transferring data?

- At every interrupt a sequence of operations is required to handle it Let’s consider a mouse which communicates its current position by interrupting the processo 30 times per second and let’s assume that 400 processor cycles are required for the dispatching of the interrupt and the execution of the interrupt service routine. The number of processor cycles which are dedicated to the mouse management per second is $400 * 30 = 12000$. For a 1 GHz clock, the processor time dedicated to the management of the mouse is 12000 ns, that is, 0.0012% of the processor load.

- Consider now a hard disk that is able to read data with a transfer rate of 4 MByte/s, and assume that the device interrupts the processor every time 16 bytes of data are available. Let’s also assume that 400 clock cycles are still required to dispatch the interrupt and execute the associated service routine. The device will therefore interrupt the processor 250000 times per second, and 100000000 processor cycles will be dedicated to handle data transfer every second. For a 1 GHz processor this means that 10% of the processor time is dedicated to data transfer, a percentage clearly no more acceptable.

- Interrupt driven data transfer is therefore not the best solution for high throughput devices. In this case the **Direct Memory Access (DMA)** technique is used

Direct Memory Access (DMA)

- In DMA it is the device responsibility to transfer data from/to memory

- In this case the device must become the owner of the bus in order to autonomously initiate a data transfer cycle
    - A Bus ownership transfer cycle must be issued first

- Before the data transfer, start address of target data in memory and number of data items to transfer must be communicated by the device driver. Memory Address (MAR) and Word Count (WC) device register are used for this operation.

- Whenever a data block is available the device shall take bus ownership and transfer the block. During overall transfer the bus ownership will likely be exchanged many times between the processor and the device(s).

- In the meantime, the processor will operate independently, and the only interference is the possible request for bus ownership for memory access (normally negligible)

- When DMA terminates, the CPU shall be informed wia an interrupt.

User and Kernel mode

- In most OS, users are not allowed to perform I/O operations directly and this operation is left to the OS itself.

- Different devices are integrated in the OS using the device layer abstraction, i.e. a set of rules to which the device code must adhere in order to be integrated in the OS itself (e.g. in Linux via the **insmod** command)

- Operations carried out by the OS are in general executed in **privileged (kernel) mode,** while user programs run in **user mode**.

- The CPU switches in kernel mode **only** upon the occurrence of an interrupt. In order to switch from user mode to kernel under program control, the CPU instruction set includes the **Software Interrupt** instruction whose effect is exactly the same of an hardware interrupt (excluding the IACK cycle)

- The vector number associated with the software interrupt is passed as argument in the instruction itself

- This is the way the OS code is organized: the non privileged code is available as a library, and the privileged one as a set of ISR associated with a set if software interrupt numbers.

The history of a **printf()** call

1.The program, running within a given process, calls routine printf(), provided by the C run time library.

2.The printf code carries out the required formatting of the passed string and the other optional arguments, and then calls the operating system specific system service for writing the formatted string on the screen;

3.The system routine executes initially in user mode, makes some preparatory work and then switches to kernel mode issuing a software interrupt

4.The ISR is eventually activated by the processor in response to the software interrupt.

5.After some work to prepare the required data structures, the ISR routine will interact with the output device. To do this, it will call specific routines of the device driver;

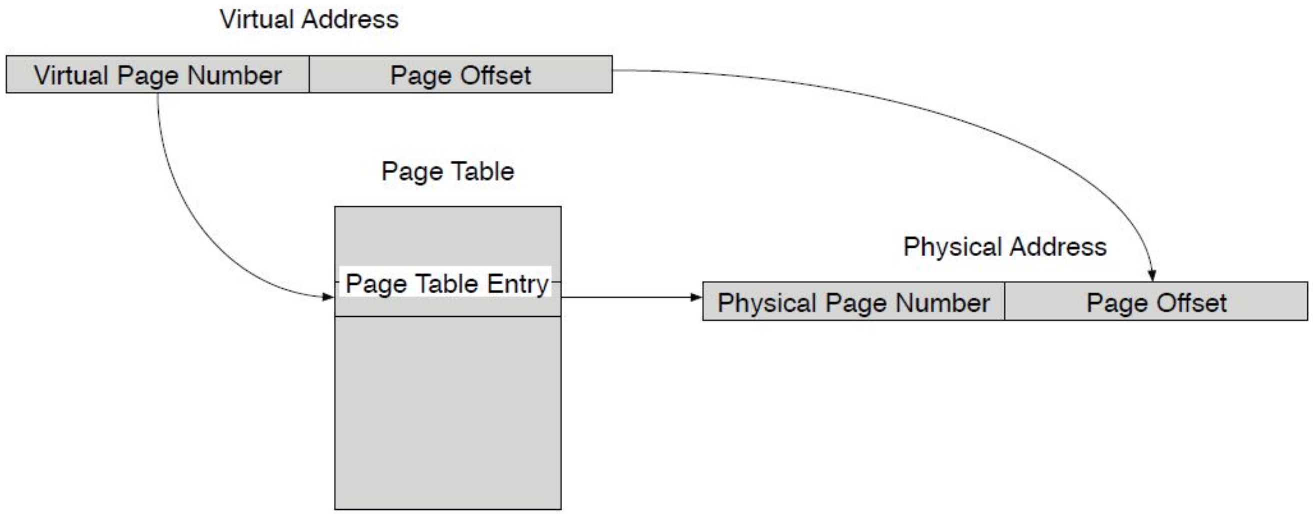
6.The activated driver code will write appropriate values in the device registers to start transferring the string to the video device. In the meantime the calling process is put in wait state

7.A sequence of interrupts will be likely generated by the device to handle the transfer of the bytes of the string to be printed on the screen;

8.When the whole string has been printed on the screen, the calling process will be resumed by the operating system and printf() will return.

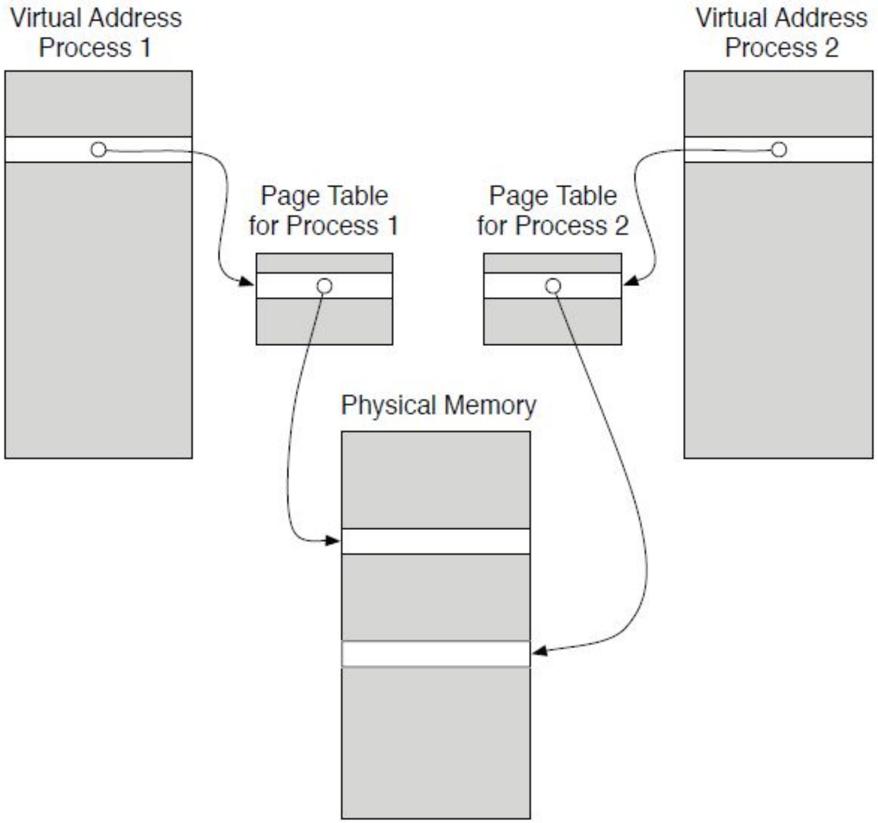
Virtual Memory

- Virtual memory, supported by most general-purpose operating systems, is a mechanism by which the memory addresses used by the programs do not correspond to the addresses the CPU uses to access the RAM memory in the same instructions.
- The address translation is performed by a component of the processor called the Memory Management Unit (MMU). The memory address managed by the user program, called Virtual Address (or Logical Address) is translated by the MMU, first dividing its N bits into two parts, the first one composed of the K least significant bits and the other one composed of the remaining N − K bits, The most significant N −K bits are used as an index in the Page Table, which is composed of an array of numbers, each L bits long.



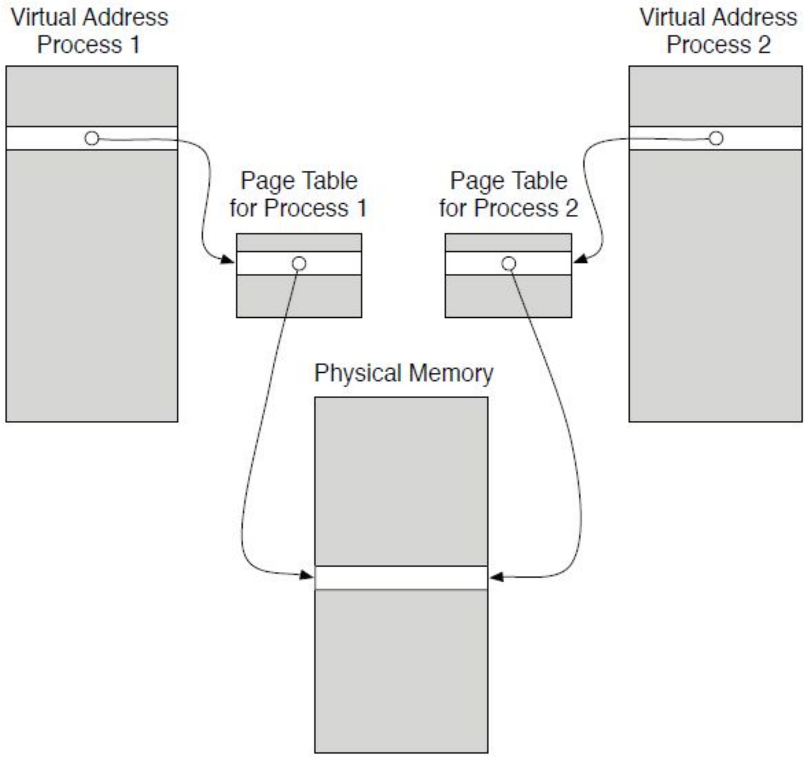
Reasons for Virtual Memory

- Consider two identical programs running in two separate processes on the same Computer.
- If no address translation mechanism were performed, the memory locations used by the first process would be affected by the second one
- When Memory translation based on the content of the Page Table is performed, the two processes do not interfere each other, provided the entries in the Page Table are properly updated when the CPU is assigned to each process
- The OS shall keep track of the proper memory mapping and allocation as well as updating the Page Table at every context switch



Shared Memory management

- The address translation mechanism can be used also to let separate processes share a section of memory
- Memory sharing is always orchestrated by the OS under request from the involved actors.
- Shared memory represents a possible Interprocess Communication (IPC) mechanism but it is in general not enough to ensure a correct communication

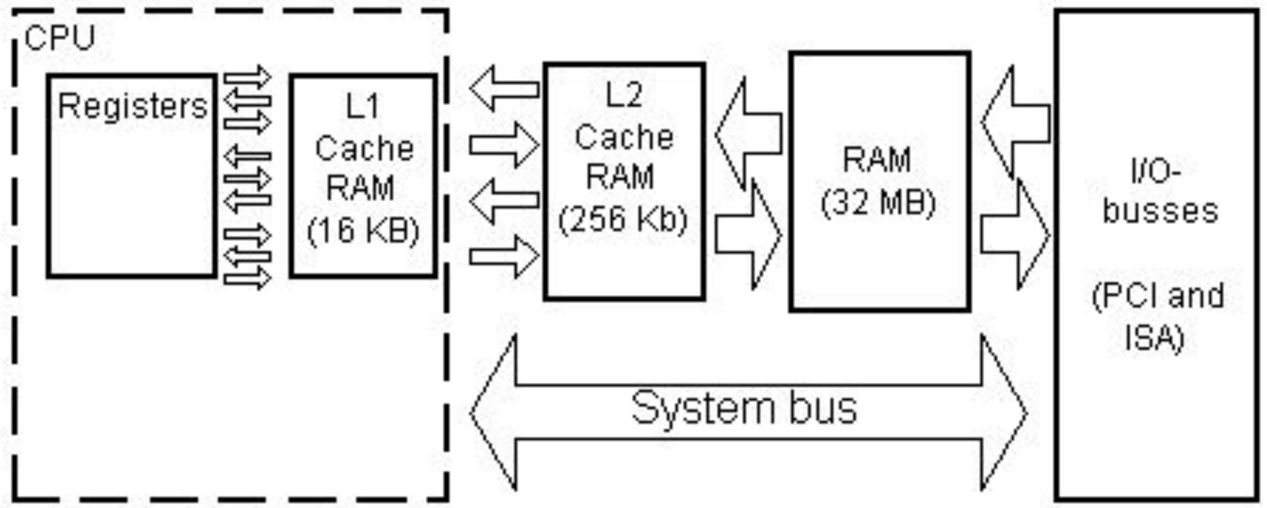


Memory swapping

- Additional information can be defined in each Page Table Entry (PTE), such as protection bits that define whether the page is accessible in User mode.
    - In case an illegal page is accessed, an exception is generated by the MMU HW itself
- It is also possible to handle a virtual address range that is larger than the physical one. In this case a bit in the PTE shall specify whether the page is resident in memory.
    - If in memory, the physical page number is contained in the PTE
    - If not in memory, the disk address of the corresponding page on disk is contained in the PTE
- When a page non resident in memory is accessed, the MMU shall issue a **Page Fault** Exception. Unlike most other exception, this does not signal an error condition, but it triggers a sequence of actions orchestrated by the OS:
    - The current process is suspended
    - A free page in memory is retrieved
    - A I/O operation (DMA read) is started for copying the page from the disk into the memory page
    - The PTE content is updated to reflect the new configuration
    - The suspended process is made ready again
- When the number of free memory pages goes under a given threshold, a swapping action begins in order to copy one or more memory page into the corresponding disk blocks

Caching

- RAM access is costly (10-100ns) in respect of internal register access
- For this reason, the sections of memory most frequently accessed are stored in a local fast memory called cache.
- When the processor accesses a part of memory that is not already in the cache it loads a chunk of the memory around the accessed address into the cache, hoping that it will soon be used again
- The chunks of memory handled by the cache are called **cache lines**. The size of these chunks is called the cache line size. Common cache line sizes are 32, 64 and 128 bytes.
- A cache can only hold a limited number of lines, determined by the cache size. For example, a 64 kilobyte cache with 64-byte lines has 1024 cache lines.



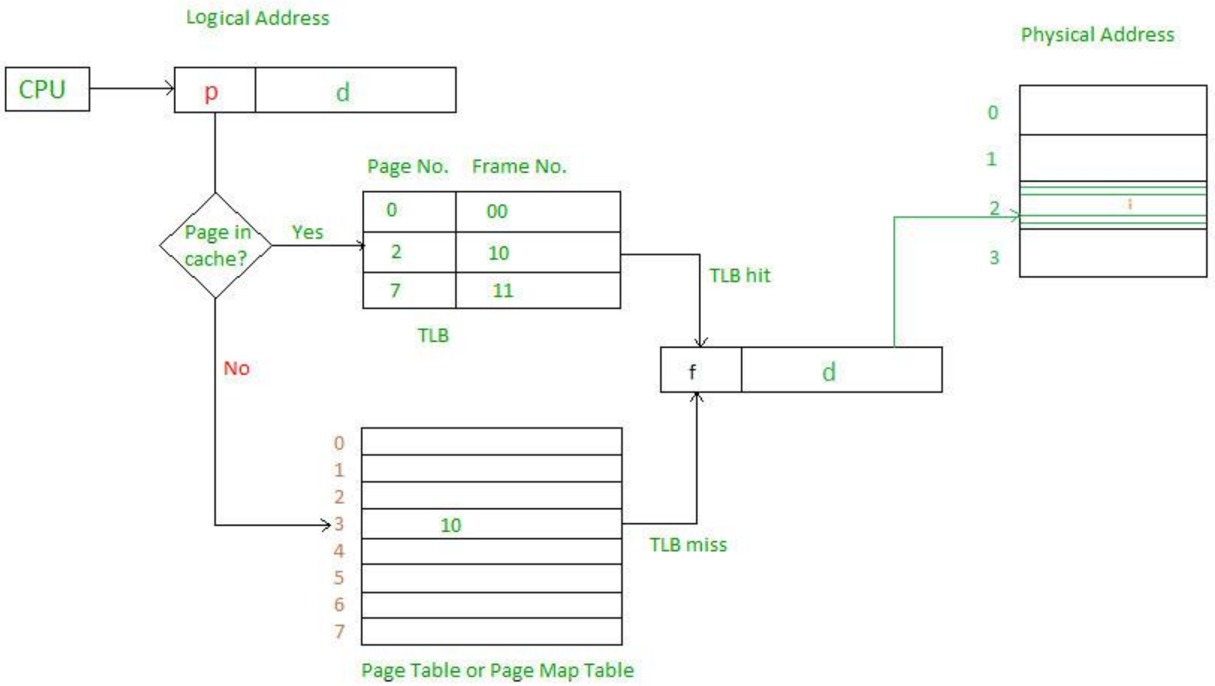
Locality in memory access

- Both swapping and caching take advantage from the locality in memory access
- When a given address is used, it is likely that the next access will be at an address close to the previous one
  - Sequential programs imply sequential memory accesses in instruction fetch
  - Arrays normally accessed in loops with increasing/decreasing indexes
- Therefore if in memory access, the data item is not found in the cache (cache **miss**), an entire cache line is copied from the RAM into the cache, and the next access will likely find the data item in the cache (cache **hit**)
- In the code snippet below, inverting indexes **`i`** and **`j`** may compromise locality because of the way C language organizes matrix memory (row by row):

```
#define MATRIX_SIZE 10000
int matrixSum(int m[][MATRIX_SIZE])
{
    int i, j, sum = 0;
    for(i = 0; i < MATRIX_SIZE; i++)
    {
        for(j = 0; j < MATRIX_SIZE; j++)
        {
            sum += m[i][j];
        }
    }
    return sum;
}
```

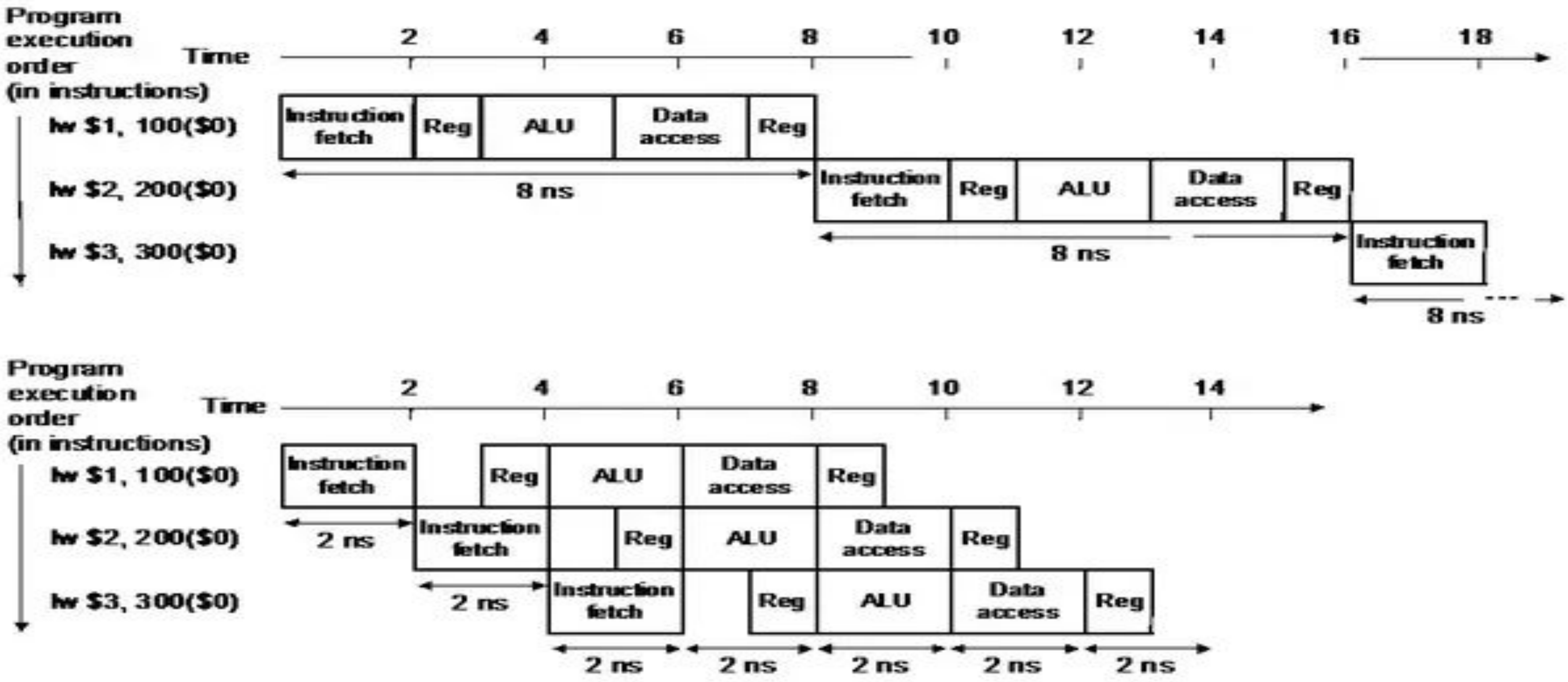
Translation Lookaside Buffer (TLB)

- Associative memory with the same cache technology is used to speed Virtual address translation
- A line in the TBL is searched (in parallel) with a tag corresponding to the virtual page number
- If found, the Physical page number is appended to the physical address
- Otherwise, the MMU is accessed and the TLB updated



Pipelining

- The execution of a machine instruction is composed of several stages
  - Instruction Fetch
  - Instruction Decode
  - Arguments readout
  - Execution
  - Output write
- When a given stage has finished execution, it is ready to process the next instruction, while the former one is being processed
- For N stages, the ideal speedup is N



Pipeline Hazards

- There are conditions preventing the ideal pipeline execution

- **Data Hazard**: when an instruction depends on the instruction just preceding it in the program. For example

```
ADD #2, R1, MEM1
SUB MEM1, R2, R4
```

- In this case the result of the ADD instruction must be stored on memory before the following instruction can read arguments.

- **Branch Hazard**: the address of next instruction after a conditional branch is not known until the branch condition has been evaluated

- **Cache misses**: may suspend an instruction for several clock cycles

Data Hazard prevention technique

- To mitigate data hazards and the effect of pipeline miss, the processor re-schedules on the fly the order of the instruction
- The correctness of the program must be preserved

For example

```
MOVE #2, R1;
ADD R2, R3, MEM1;
SUB MEM1, R2, R3;
```

can be arranged as

```
ADD R2, R3, MEM1;
MOVE #2, R1;
SUB MEM1, R2, R3;
```

Branch Hazard prevention: branch prediction

- After fetching a conditional branch instruction, the next instruction to fetch depends on the evaluation of the branch instruction

- Nevertheless, the address of the next instruction is assumed to be the most recently used in the same instruction (or the next one if it is the first time that branch instruction is executed).

- A cache-like table is maintained keeping the recent history of the branch instructions so that a decision is taken soon. A line is searched where the instruction address corresponds to the current fetch address. If found, the second element is taken as address for the next fetch stage

- When the branch condition is evaluated at a later stage and the assumption was not correct, the pipeline is reverted and the next correct instruction is fetched

- This operation is performed ***entirely*** by the HW

Instruction Address	***Branch Target Address***

Code Optimization techniques

Mostly carried out by the compiler optimizer:

- **Register allocation**: use as far as possible CPU register to hold the content of the program variables. Moves away from the stack-based call frame organization.

- **Code reduction**: perform at compile time all the possible computation ad use more efficient operations (e.g. replacing multiplications by sums), move away from loops operations that need to be performed only once

- **Loop unfolding**: when conveniente unfold loops in order to reduce the overhead of branch condition check and to reduce branch hazards

- **Code reordering**: in order to reduce the effects of data hazards

Can the programer help the compiler optimizer in some way?

- Very sophisticated speculation is carried out by the compiler optimizer, especially when the program semantics is defined using program variables.

- When pointers and memory access are used instead, the compiler cannot speculate what is going to happen to the memory locations that may be changed from external actors

- Therefore it is up to the programmer to make good usage of memory organization, favoring as far as possible localized memory access in order to make best usage of caching.

- As a rule of thumbs, using a large number of variables in place of pointers (e.g. mapping a sliding memory region in a set of variables) allows the optimizer make a much better work.