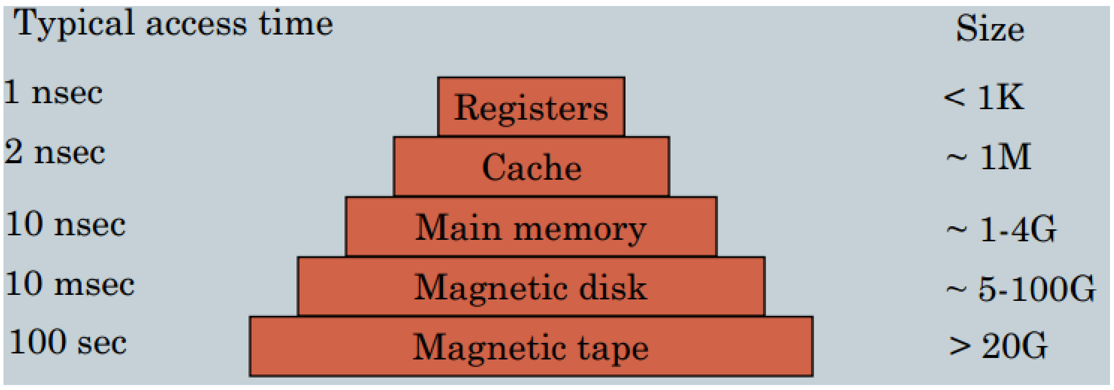
Cairo University
Faculty of Computers & Information
Computer Science Department

Operating System Course
Lab 6 - Memory Management

Theoretical part: (45 minutes)

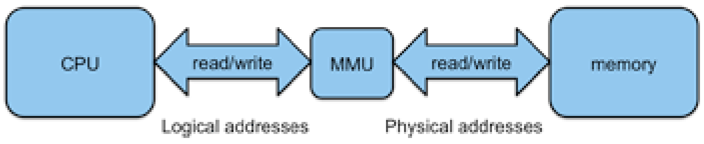
A program resides on a disk as a binary executable file. To be executed, the program must be brought into memory and placed within a process. Depending on the memory management in use, the process may be moved between disk and memory during its execution. The processes on the disk that are waiting to be brought into memory for execution represent the input queue.

Memory management is about making sure that there is enough memory available within the system to run the various processes on offer. Each process needs memory to run, and some processes require more than others to run. It is the role of the memory management algorithm to allocate memory to processes in such a way as to satisfy one or more policies.



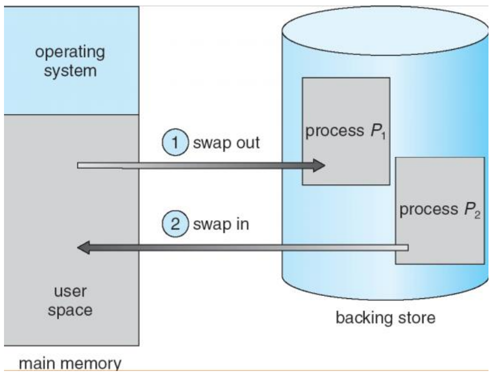
1. <u>Logical and physical addresses:</u>

- The logical address (virtual address): An address generated by the CPU that the user program uses (e.g. pointers).
- The physical address: where programs are really loaded in. the address actually seen by the memory management and loaded in the memory-address register
- The objective for using these types of addresses is to protect the operating system and user processes from any illegal access, by providing a separate legal address space for each process
- Address Binding is the mapping from one address space to another(from logical to physical or vice versa), binding can occur in compile or load or execution time



2. <u>Swapping:</u>

- A process can be swapped temporarily out of memory to a backing store (fast desk) and then brought back into memory for continued execution.



3. <u>Memory Allocation Policies</u>

The memory is usually divided into two partitions: one for the resident operating system and one for the user processes.
The operating systems allocates space and loads a process in memory according one of the following policies

- First fit - allocate the first hole that's big enough.
- Best fit - allocate smallest hole that's big enough.
- Worst fit - allocate largest hole.

Memory allocation can cause fragmentation either external or internal:

- **External fragmentation** occurs when there is no enough total memory space to satisfy a process request.
- **Internal fragmentation** occurs when the memory allocated to a process may be slightly larger than the requested memory.

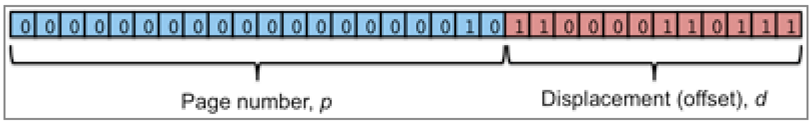
*First fit is fastest, worst fit has lowest memory utilization.

4. Basic memory management strategies

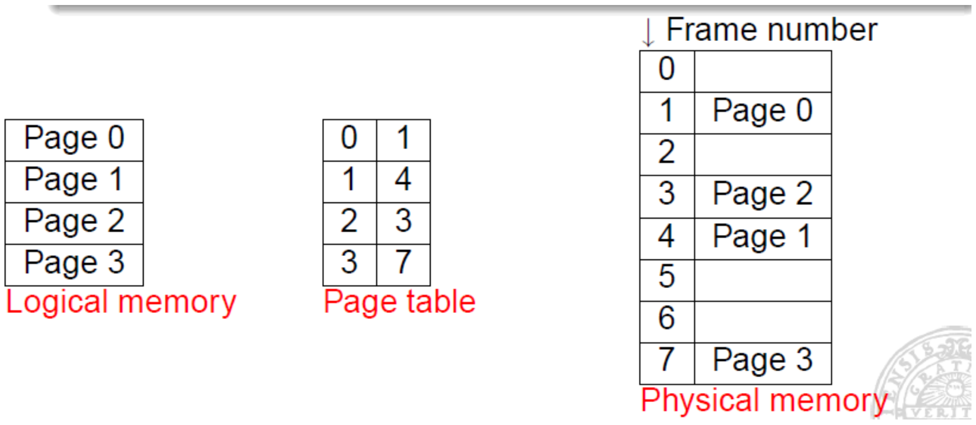
4.1. Paging

- Paging permits the physical address space a process to be noncontiguous. Paging avoids external fragmentation and the need for compaction.

- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8192 bytes).
- Divide logical memory into blocks of same size called **pages**.
- Pages and frames are of the same size.
- Keep track of all free frames.
- To run a program of size *n* pages, need to find *n* free frames and load program.
- Set up a page table to translate logical to physical addresses
- Address generated by CPU is divided 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 memory address that is sent to the memory unit



    - Causes internal fragmentation.



Page Replacement Algorithms:

- **FIFO Algorithm**: associates with each page the time when that page was brought into memory, when a page must be replaced, the oldest page is chosen.

- **Optimal Replacement Algorithm**: Replace the page that will not be used for the longest period of time.
    - Difficult to implement, because it requires future knowledge of the reference string.
    - Generates the lowest possible page fault rate for a fixed number of frames.

- **Least Recently Used (LRU) Algorithm**: Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter, when a page needs to be changed, look at the counters to determine which are to change (replace the one with minimum counter).

4.2 Segmentation:

- View memory as a collection of variable-sized segments, rather than a linear array of bytes ("paging with variable page size").
- Causes External Fragmentation.

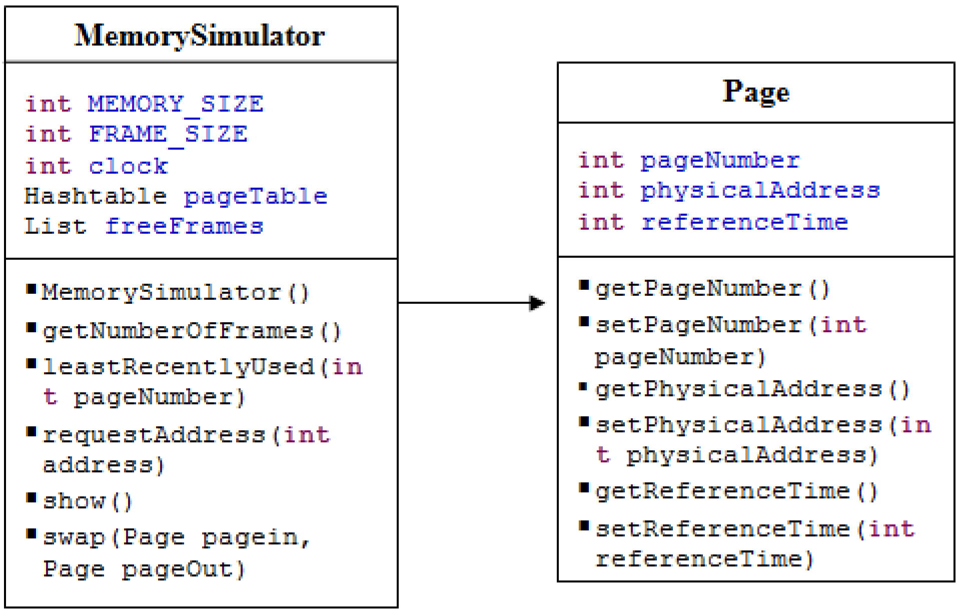
4.3 Virtual memory (5 minutes)

- **Virtual memory** – separation of user logical memory from physical memory.
    - Only part of the program needs to be in memory for execution.
    - Logical address space can therefore be much larger than physical address space.
- Virtual memory can be implemented via:
    - Demand paging: Bring a page into memory only when it is needed
    - Demand segmentation Bring a segment into memory only when it is needed

Technical part: (45 minutes)

Design a system that simulates demand paging with least—recently-used replacement technique.

Memory Simulator Design:



Classes Description:

MemorySimulator	
getNumberOfFrames()	Returns the total number of frames available in memory
leastRecentlyUsed(int pageNumber)	Returns the page that was least recently used according to its counter
requestAddress(int address)	Requests a specific address if page not loaded load in memory, if no free frames a process swap is performed between the requested page and the least recently used page
show()	Display page table content
swap(Page pagein, Page pageOut)	Swaps two pages one in and one out of memory
Page	
getPageNumber()	Returns current page number
getPhysicalAddress()	Returns physical address of the current page
setPhysicalAddress(int	Sets the physical address of the

physicalAddress)	page
getReferenceTime()	Returns number of references the page was referenced
setReferenceTime (int referenceTime)	Sets number of times the page was referenced

Memory Simulator Implementation

```java
public class Page {
	private int pageNumber;
	private int physicalAddress;
	private int referenceTime ;

public int getPageNumber() {
		return pageNumber;
	}
public void setPageNumber(int pageNumber) {
		this.pageNumber = pageNumber;
	}
public int getPhysicalAddress() {
		return physicalAddress;
	}
public void setPhysicalAddress(int physicalAddress) {
		this.physicalAddress = physicalAddress;
	}
public int getReferenceTime() {
		return referenceTime;
	}
public void setReferenceTime(int referenceTime) {
		this.referenceTime = referenceTime;
	}
}
```

```java
import java.util.ArrayList;
import java.util.Hashtable;
import java.util.Iterator;
import java.util.List;

public class MemorySimulator {
private final int MEMORY_SIZE = 50;

	private final int FRAME_SIZE = 10;

	private int clock=0;

	private Hashtable<Integer, Page> pageTable = new Hashtable<Integer, Page>();

	private List<Integer> freeFrames = new ArrayList<Integer>();
public MemorySimulator() {

		for (int i = 0; i < getNumberOfFrames(); i++)
			freeFrames.add(i);
	}
private int getNumberOfFrames() {
		return MEMORY_SIZE / FRAME_SIZE;
	}
public Page leastRecentlyUsed(int pageNumber) {
		int min = 10000;
		int maxPageNumber = -1;
		Iterator<Page> pageIterator = pageTable.values().iterator();
		while (pageIterator.hasNext()) {
			Page page = pageIterator.next();
			if (min > page.getReferenceTime()) {
				maxPageNumber = page.getPageNumber();
				min = page.getReferenceTime();
			}
		}
		return pageTable.get(maxPageNumber);
	}
public void requestAddress(int address) {

		//Increament the clock
		clock++;
		int pageNumber = address / FRAME_SIZE;
		int offset = address % FRAME_SIZE;
		Page page = pageTable.get(pageNumber);
		int physicalAddress;
		// Page is not in the memory
		if (page == null) {

			Page pagein = new Page();
			pagein.setPageNumber(pageNumber);
			// If no free frams Swap
			if (freeFrames.size() == 0) {
				Page pageOut = leastRecentlyUsed(pageNumber);
				pagein.setPhysicalAddress(pageOut.getPhysicalAddress());
				pagein.setReferenceTime(clock);
				swap(pagein, pageOut);
			} else {
				//There are free frames no need for swap
				int frameNumber = freeFrames.get(0);
				pagein.setPageNumber(pageNumber);
				pagein.setPhysicalAddress(frameNumber * FRAME_SIZE);
				pagein.setReferenceTime(clock);
				pageTable.put(pagein.getPageNumber(), pagein);
				freeFrames.remove(0);
			}
			physicalAddress = pagein.getPhysicalAddress() + offset;
		} else {
			// Page in the memory just increament the refernce counter
			page.setReferenceTime(clock);
			physicalAddress = page.getPhysicalAddress() + offset;
		}

		System.out.println("Request Logical Address " + address + " Physical
Address " + physicalAddress );
	}
```

```
      public void show() {

            System.out.println("----------------------------------------");
            Iterator<Page> pageIterator = pageTable.values().iterator();
            while(pageIterator.hasNext())
            {
                  Page page = pageIterator.next();
                  System.out.println("At " + page.getPhysicalAddress() + " Page 
Number: " + page.getPageNumber() + " Last Reference Time: " + 
page.getReferenceTime());
            }
            System.out.println("----------------------------------------");

      }
private void swap(Page pagein, Page pageOut) {
            pageTable.remove(pageOut.getPageNumber());
            System.out.println("Swap Out Page Number: " + 
pageOut.getPageNumber());
            pageTable.put(pagein.getPageNumber(), pagein);
            System.out.println("Swap In Page Number: " + 
pagein.getPageNumber());
      }

public static void main(String[] args) {

            MemorySimulator memory = new MemorySimulator();
            memory.requestAddress(22);
            memory.show();
            memory.requestAddress(33);
            memory.show();
            memory.requestAddress(13);
            memory.show();
            memory.requestAddress(55);
            memory.show();
            memory.requestAddress(6);
            memory.show();
            memory.requestAddress(8);
            memory.show();
            memory.requestAddress(53);
            memory.show();
            memory.requestAddress(11);
            memory.show();
            memory.requestAddress(23);
            memory.show();
            memory.requestAddress(67);
            memory.show();


      }

}
```