CIS-350
INFRASTRUCTURE TECHNOLOGIES

<u>THE INTERNAL OPERATING SYSTEM</u> - (Chapter 18)

Read chapter 18 to the extent it was covered in class and the lecture notes below.

The chapter provides an in-depth discussion of
- starting the computer system – the bootstrap
- process and thread management (control)
- CPU scheduling and dispatching
- memory management
- other OS services/functions
  - secondary storage scheduling (see pp. 10-12 of the lecture notes for Chapter 10)
  - network operating system services
  - deadlock resolution (see p. 15 of lecture notes for Chapter 15)
  - virtual machines
- file management services (chapter 17)

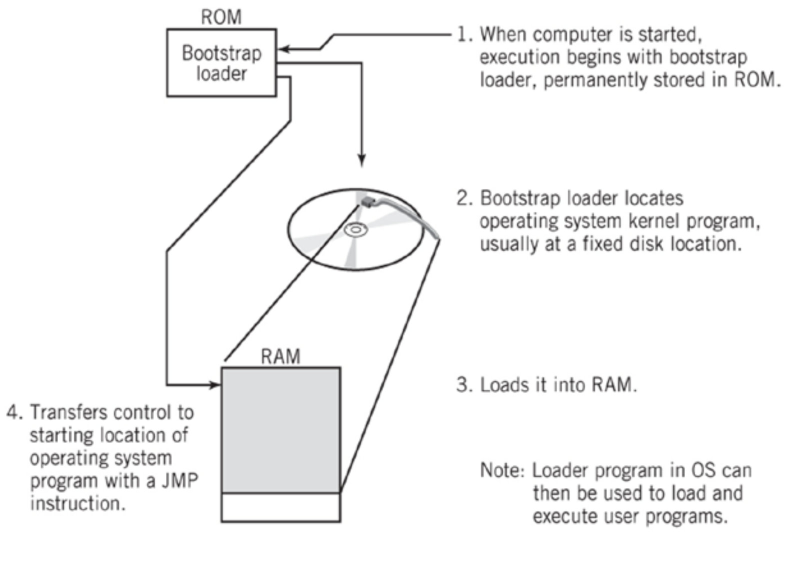
The above services are executed in the protected mode (kernel mode). In other words, if the user or executing program (process) requests any of these services, control is surrendered to the OS.

I will complement the material presented in this chapter with the material from other books on OS.

STARTING THE COMPUTER SYSTEM: THE BOOTSTRAP

When the computer is turned on, the RAM is empty (unknown contents). There is no OS in it. The OS has to be loaded first.

Fig 18.5 Bootstrapping the computer


ROM
Bootstrap loader
1. When computer is started, execution begins with bootstrap loader, permanently stored in ROM.
2. Bootstrap loader locates operating system kernel program, usually at a fixed disk location.
RAM
3. Loads it into RAM.
4. Transfers control to starting location of operating system program with a JMP instruction.
Note: Loader program in OS can then be used to load and execute user programs.

PROCESSES AND THREADS

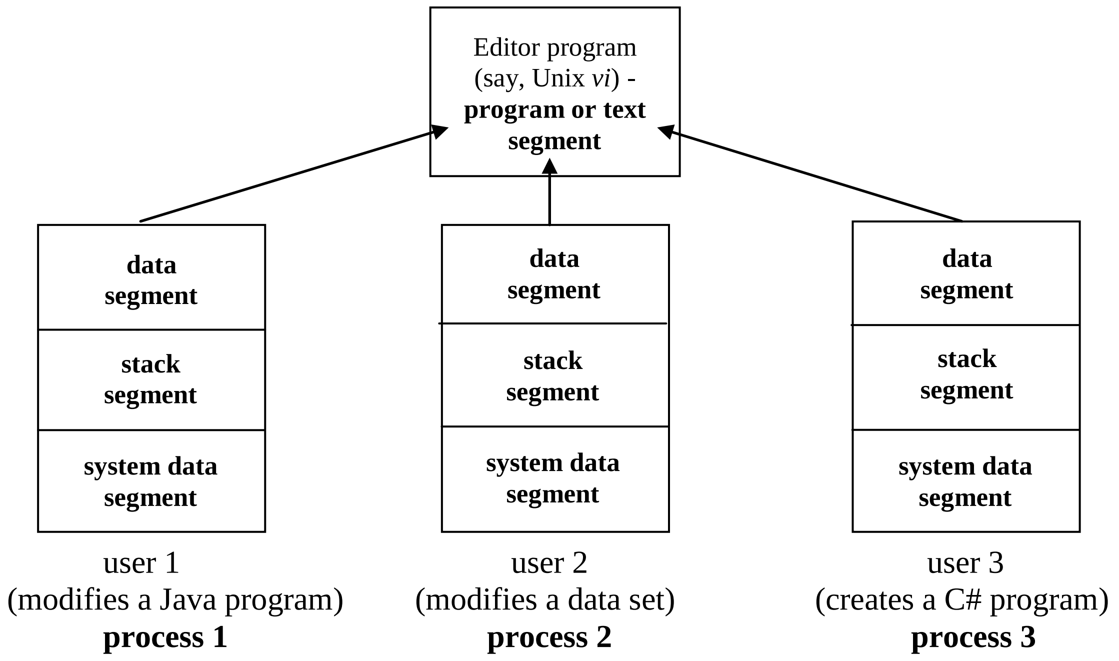
A **process** is an executing program/task/command together with all the resources allocated by the OS to that program/task.

Resources: CPU time, memory (stack, buffers), I/O devices, data files etc.

Processes are created and terminated (or killed) by the OS. Processes can be in a variety of states. Processes can be divided into threads.

When a process is admitted to the system, the OS is responsible for every aspect of its operation. The OS allocates the above resources to that process. When the process is completed, it releases these resources under control of the OS, so they can be used by other processes.

Program vs. Process (the difference to the OS is significant). The program is a static entity residing on disk and not consuming much of the resources, except the disk space. The process is a dynamic entity with all the resources allocated to it by the OS.



In a multitasking system, a single copy of an editor program can be shared by many users. Because users perform different tasks (see above) using an editor, each user will have its own data segment created by the OS.

Each process will have its own data segment but all of them will be sharing the same program segment which must be protected from change. This way, storage can be saved. (In addition, in UNIX, the OS will maintain a separate stack segment and a system data segment for each process.)

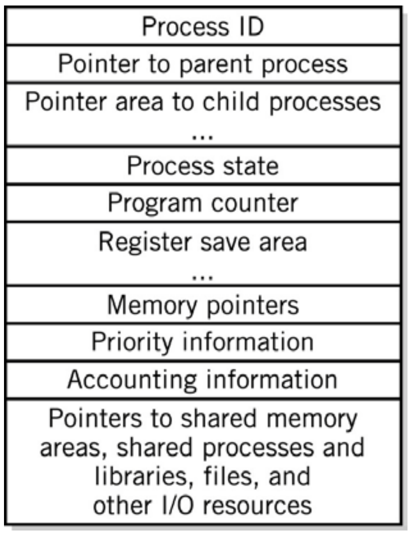
As a result, the OS will create 3 processes, one for each user.

Independent processes are those that do not interact with each other.

Cooperating processes are those that work together. The OS must provide a mechanism for synchronizing and communicating between processes.

The OS maintains a **process control block (PCB)** for each process. The PCB contains all relevant information about the process. The PCB is implemented as a table and stored in memory. It simply represents a block of data for each process in the system.

Fig. 18.7


Process ID
Pointer to parent process
Pointer area to child processes
...
Process state
Program counter
Register save area
...
Memory pointers
Priority information
Accounting information
Pointers to shared memory areas, shared processes and libraries, files, and other I/O resources

Process ID in Unix, for example, is a random 5- or 6-digit # assigned to the process.

Process creation

In UNIX, almost each command you type from a shell prompt creates a process.

$>ls -al

The above command lists **<u>a</u>ll** files residing in the current directory and does it in a **<u>l</u>ong** form. When the command is done, the process is killed.

In UNIX, a log in operation also creates a process. A running shell is a process as well. The OS creates and kills processes.

The Unix *ps* command displays all processes running on your terminal.

When an OS module is executing to service the request, it is sometimes called a system process. All other processes are usually called user processes.

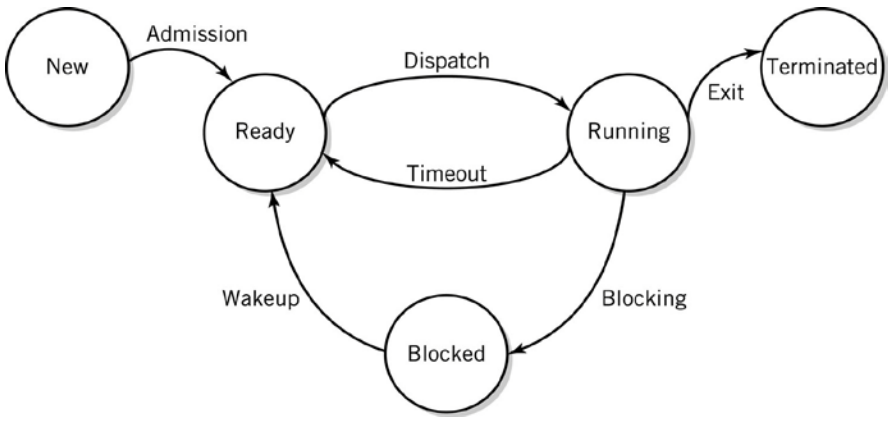
UNIX creates a new process from an older one by operation called **forking** or **spawning**. The forking process is called a **parent**. The forked process is called a **child**. Initially, the child process is a clone of the parent process and has the same PCB as its parent, but then the child process goes its own way. The parent process goes to sleep (or executes) and waits for the child process to die. (See also pp. 604-605) and lectures as well as the labs on Unix which we will cover later.

Process states

Three primary states for a process (the # depends on the OS)

- **ready** state
- **running** state
- **blocked** state

Fig 18.8 The Major Process States



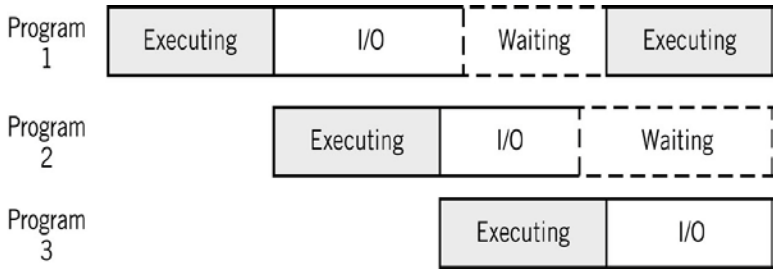
The circles and arcs represent the process states and process transitions, respectively. See pp. 605-606.

In Unix operating system the user can suspend the running process by using the *Ctrl-Z* keys. The user may also resume running of the suspended process by the *fg* command followed by the process ID #.

Nonpreemptive systems will allow a process to continue running until it is completed (voluntarily gives up the CPU) or blocked (issues I/O request). Used in older systems – batch systems.

<u>I/O bound programs</u> do not consume too much of the CPU time. As a result, the surrender voluntarily control to the processor. They can execute in nonpreemptive environments.

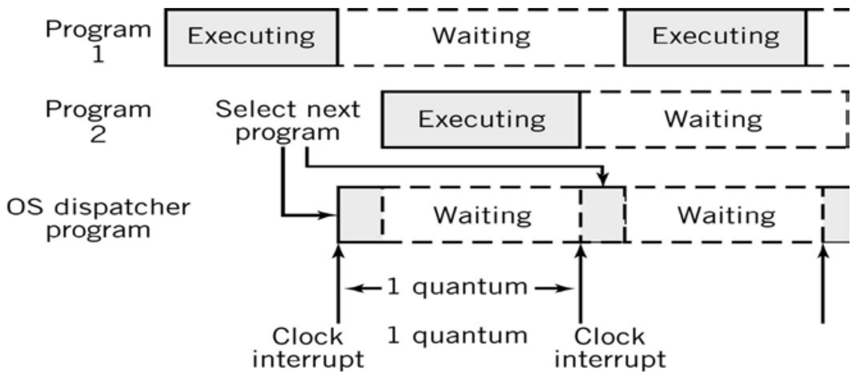
Fig. 15.3 Sharing the CPU during I/O breaks.



Preemptive systems impose the clock interrupt - use <u>time slicing</u>. The program is given the CPU for a <u>slice (quantum) of time</u> (10 ms, 20 ms, or 50 ms). If the program exceeds this time, the time out routine kicks in and the program is suspended.

Many users and programs can share the computer simultaneously. Allowing a single user to dominate the system is unacceptable. All users have to be treated fairly. If users are to be treated fairly, 5 minutes of continuous CPU time cannot be granted to a single user.

Fig. 15.4 Time-sharing the CPU.



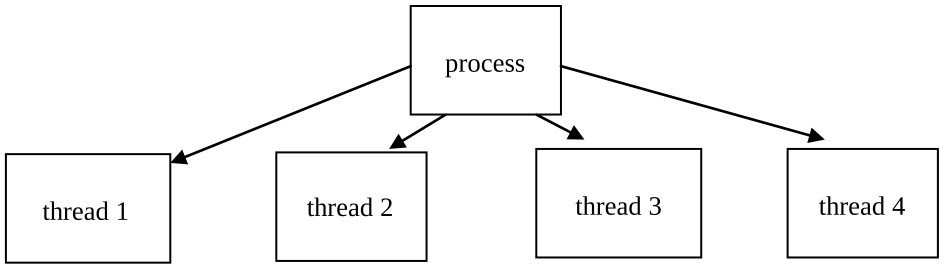
The processor switches so quickly between different tasks/programs that the user has got the impression that the computer works exclusively for him. In fact, it does not.

Compute (CPU) bound programs use little I/O and a great deal of the CPU, and do not generate interrupts too often. CPU bound programs need to be executed in preemptive environments.

Context Switching – transferring control to the program being dispatched.

Threads

A **thread** is a piece of a process (a miniprocess) that can be executed independently of other parts of the process. A process can be divided into a couple of threads.



Threads are created in event-driven programs using a GUI. For example, clicking a mouse on an icon generates an event and the OS creates a thread for this event. Events are usually too small to justify creation of a new process.

The OS maintains one PCB for a process regardless the number of threads the process has. The threads use the same program code and data space as the process. This approach reduces an overhead and makes context switching simpler. However, the OS maintains an area to store the program counter and registers for each thread. Like processes, threads have their own state; they are also created and killed. Threads can operate concurrently.

CPU SCHEDULING

It concerns optimization of the system performance by allowing several processes to execute concurrently.

Done primarily in two or three phases:

- **high-level (long-term) scheduler**
  - admits processes into the system
- **short-term scheduler (dispatcher)**
  - allocates the CPU time to those processes
- **middle-level scheduling** (used in some OS)
  - allows one to monitor/improve the system performance by suspending and **swapping out** processes which hold resources that other processes may need.

Long-term scheduling

- determines which processes are to be admitted to the system
- concerns processes which reside on disk
- admission is based on resources' availability (memory), system load, and priorities of processes

The role of the long-term scheduler is minimal for interactive systems and more important for batch systems.

Interactive systems

Processes are admitted automatically unless the system is very seriously overloaded. The scheduler may refuse a user to log in if the system is too overloaded, but it will rarely deny a service to a user who is already logged in to the system. However, it may take more time for the OS to deliver the requested service.

Batch systems

The OS has routines which allow one to determine the resources needed by processes. (MVS/JCL system has a JES2 routine - Job Entry Subsystem). The OS also keeps a dynamic track of resources that are available or occupied. Processes may be denied admission to the system if the resources are not currently available. Then processes usually reside on a disk and wait until the resources they need become available.

Dispatching

concerns selecting processes that are ready to run and allocating the CPU time to them

Nonpreemptive dispatching

- processes are allowed to run until they are completed or blocked (issue I/O request)
- time-out routines for endless loops
- little overhead (less context switching)
- not always fair, used in older systems

Preemptive dispatching

- allocating the **slice of the CPU time** (100 or 10 milliseconds) to each process
- if a process is completed before the time allocated to it elapses, it surrenders the remaining time to the OS. If a process does not finish before within the time allocated, the time out routine kicks in and the process is suspended.
- more overhead because of context switching
- improves the overall system performance
- all processes are treated "fairly" because each one has a chance to get the CPU time

Context switching

Let's deal with two processes only: Process 0 and Process 1. The OS maintains PCB0 and PCB1 in support for Process 0 and Process 1, respectively. Assume that Process 0 is suspended and Process 1 is in the ready state and is going to get the CPU time. Before Process 1 starts running, the dispatcher has save the state of Process 0 in PCB0 and reload the state of Process 1 from PCB1.

(See page 3 in the document titled "CIS-350 (Real Memory Management and Virtual Memory Organization and Management) - Additional Figures.pdf" posted on Blackboard.)

Different dispatching algorithms for selecting processes are used to optimize the system performance because computer systems have different objectives:

- Batch systems
  - maximize throughput and minimize turnaround time
- Interactive systems
  - minimize response time
- other systems
  - maximize CPU and/or other resource utilization, protect processes from starvation

Dispatching algorithms

Nonpreemptive algorithms

FIFO

- processes are executed as they arrive
- simple
- not always fair - penalizes short jobs and I/O-bound jobs
- may result in long waits
- may poorly utilize resources such as CPU time and I/O.

SJF (Shortest Job First)

- maximizes throughput
  - first it selects jobs which will consume a small amount of CPU time [the dispatcher goes by the value specified on the TIME parameter: TIME=(,2) - MVS/JCL]. If a job "cheats", it will receive a severe penalty
  - starvation possible for jobs which need more CPU time
  - inconsistent turnaround time

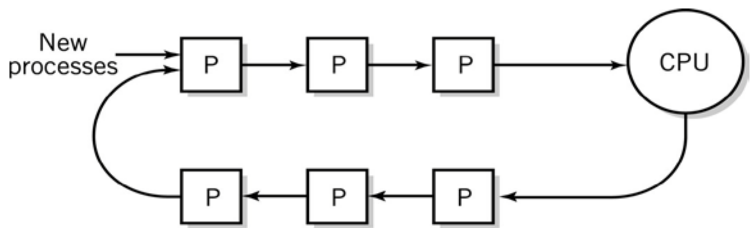
Priority scheduling

- each job has a priority assigned to it
    - priorities may be granted by the system administrator to
        - particular users (the more you pay for CPU time the higher priority you get)
        - "hot" jobs entering the system which need an immediate attention
    - priorities may determined by the OS based on the overall amount of resources the process requests (more resources less priority)
    - the dispatcher will allocate CPU time to the job with the highest priority
    - priorities can be dynamically increased or reduced by the dispatcher while the job is running

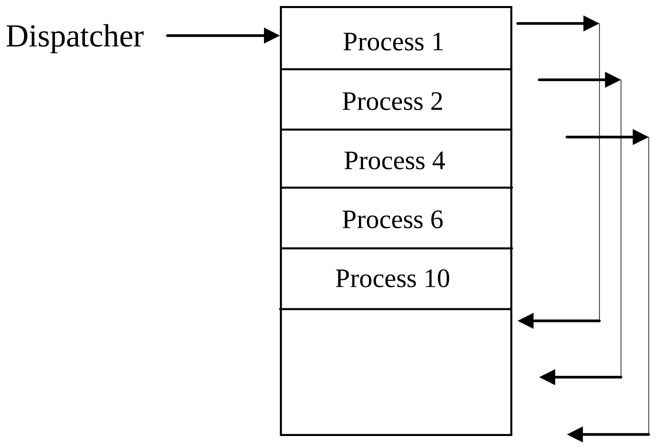
Preemptive dispatching

Simple round robin polling algorithm

- gives each process a quantum of CPU time
- simple and fair
- maximizes throughput
- penalizes processes which use a lot of I/O



<u>PCBs are polled in the polling cycle</u>. Say, Process 1 gets the slice of the CPU time. When Process 1 finishes or exceeds its time limit it gets to the end of the line, and Process 2, Process 4, and Process 6 will get the CPU time. Process will get the CPU time again after all programs are polled.



Modifications to simple round-robin polling:

1. **Two tables**

High priority processes

Process 1
Process 5
Process 15
Process 3
…
Process 8

Low priority processes

Process 10
Process 20
…

2. Programs may have **repeated entries** in the table, and get the CPU time more often since the program is polled more frequently.

Process 4
...
...
Process 4
...
...
Process 4

3. In Linux, Unix, Windows (2000, XP, Vista, 7), the following **ratio** is computed for each process to determine the dynamic priority of every process.

$$\text{Process} = \frac{\text{total CPU time received}}{\text{time of residency in memory}}$$

This is computed for every process. The lower the ratio is the higher priority the process has. The priority is recomputed every 2-3 seconds.

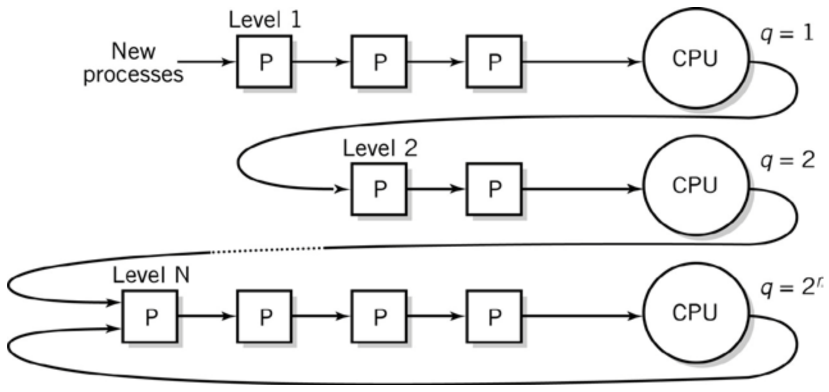
$$\text{Process 4} = \frac{0.1\text{s}}{10\text{s}} = 0.01$$

$$\text{Process 5} = \frac{0.01\text{s}}{10\text{s}} = 0.001$$

Process 5 gets the higher priority.

Multilevel feedback queues

There are a number of queues. New processes enter the top priority queue and get CPU time almost immediately. Short processes will complete at this point. Processes that are not completed are sent to the second-level queue. They even receive more CPU time but only when the first-level queue is empty. The final level is a round robin queue and processes stay there until they are completed.



REAL MEMORY MANAGEMENT, VIRTUAL MEMORY ORGANIZATION, VIRTUAL MEMORY MANAGEMENT, and ADRESS TRANSLATION

Real Memory Management

The goals of the OS
- find space for programs in memory
- minimize memory wastes
- improve overall system's performance

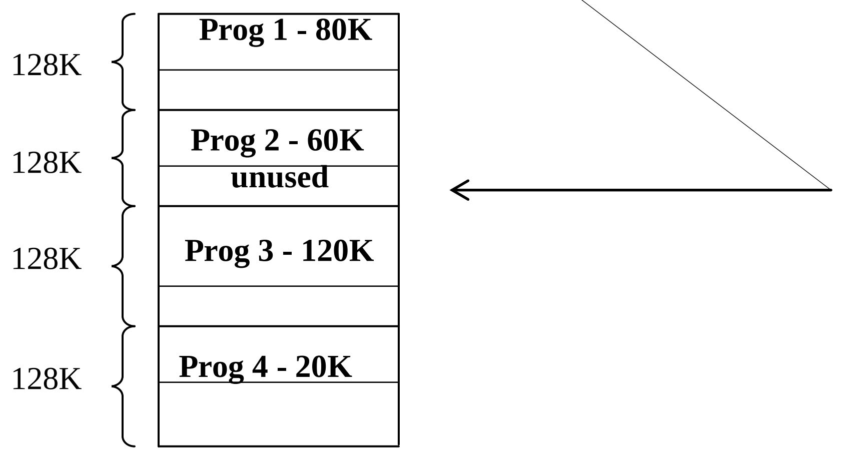
Memory management is simple for single-tasking systems. However, if they apply overlays (used more in old systems when memory was scarce), it may be a little complicated. It is more complex in multitasking systems.

Techniques
- fixed partition
- dynamic
- segmentation, paging, segmentation and paging
- virtual memory

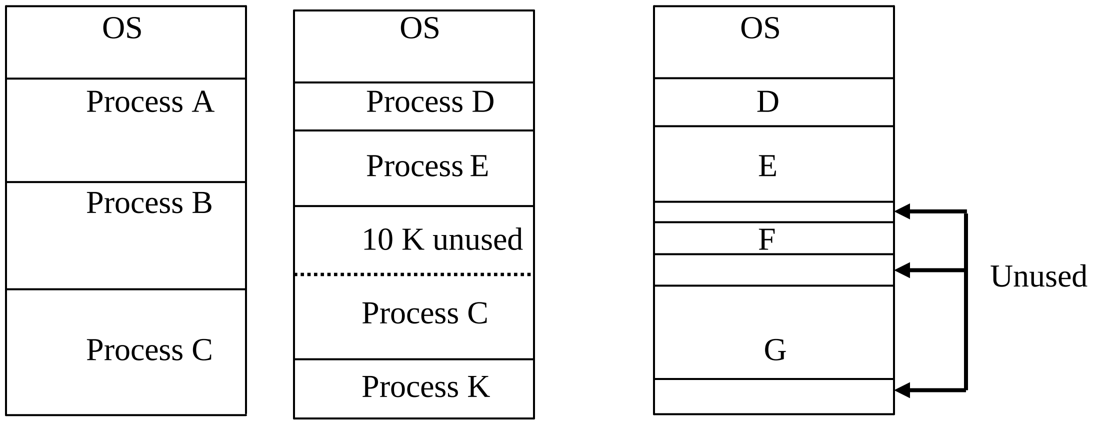
Fixed partitioning (used on IBM/360, MFT –Multiprogramming with the Fixed Number of Tasks)

- memory is divided into **equal partitions** (say, 128KB each)
- each partition holds one program
- partitions are established at the IPL time (before programs are loaded)
- **simple concept** but **memory** space is **wasted** (if a program occupies 60KB, 68KB is unused)
- **internal fragmentation**
- # of partitions depends on memory available

`



Dynamic Partitioning (MVT - Multiprogramming with the Variable number of Tasks)



Processes are loaded to memory. Each process is assigned as much space as it needs. Initially no memory is wasted.

Processes A & B are replaced by D and E. 10 KB is an unused fragment.

As processes are exchanged, memory becomes **fragmented**. It is a problem called **external fragmentation**

Many **strategies** can be used to determine which **fragment** (**hole**) is the best.

first-fit - allocate the first hole that is big enough to hold a process.

best-fit - allocate the smallest hole that is big enough to hold a process.

worst-fit - allocate the largest hole to hold a process.

The OS maintains the list of holes (**free memory list**), sorted, for example, by the hole size. The first-fit technique is the fastest, whereas best-fit may be the slowest.

Deitel *et al*. text, for example, discusses various techniques of combining holes, i.e., reducing external fragmentation. These are

- coalescing
- memory compaction, sometimes called burping memory (garbage collection)

Address Translation

Recall the LMC language covered in chapter 6. Below is the LMC program from Test 2. The program reads in two numbers, say, 30 and 45, and outputs the larger of the two. You can see that the program statements start at address 00 and end at 10, and that program instructions use memory addresses (50 and 51), chosen by you arbitrarily, for memory locations. (Any other memory locations could have been used.) In other words, the program uses relative (logical or virtual) addresses.

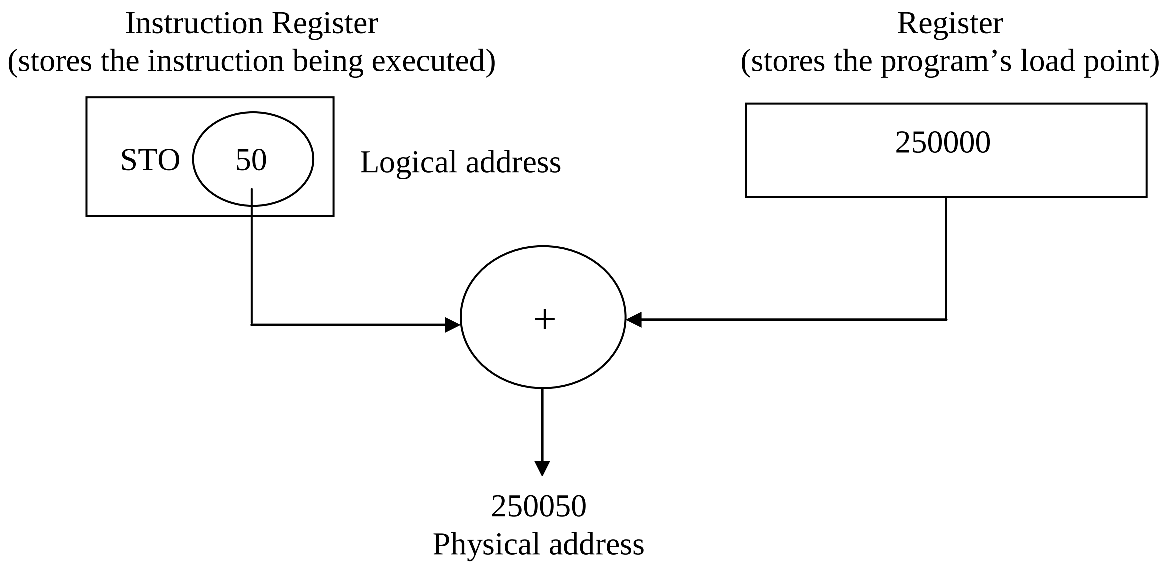
```
     00  IN
     01  STO 50
     02  IN
     03  STO 51
     04  SUB 50
     05  BRP 08                    program area
     06  LDA 50
     07  BR 09
     08  LDA 51
     09  OUT
     10  HLT
----------------
----------------
     50  30                        data area
     51  45
```

Say, that this program (program area and data area) is stored on disk and the OS loaded this program to memory starting at address, say, 250000. The program is ready to execute. Before any instruction of the program is executed, the program's operands have to be translated to physical (real or absolute) addresses. [Here the operands represent relative (logical or virtual) addresses - memory locations (50 and 51) and branches to the instructions at addresses 08 and 09]. References to memory locations 50 and 51 are irrelevant because now the data is stored at memory locations 250050 and 250051. The same is true for branching to the instructions at addresses 08 and 09 as now these instructions are at memory locations 250008 and 250009, respectively.

The program load point, which is 250000 is loaded into one of the registers in the CPU and the instructions' operands, including two branch instructions, have to be modified by the program's load point.

```
250000        IN
250001        STO 50
250002        IN
250003        STO 51
250004        SUB 50
250005        BRP 08               program area
250006        LDA 50
250007        BR 09
250008        LDA 51
250009        OUT
250010        HLT
------------------------
------------------------
```

200050	30	data area
200051	45	



This address translation is done after the fetch phase of the instruction cycle and before the execute phase.

The example above illustrates a simple case of address translation. It assumes that the entire program is loaded into memory and executed. The process of address translation is somewhat more complex because programs are divided into segments, pages, or segments and pages. We will discuss it very soon.

The program above or any other program can be swapped out of memory and swapped back in to memory at a different memory location, say, 500000, determined by the OS. The register which contains the program's load point will now contain 500000, and all program's operands will be modified by 500000. Programs can be <u>relocated</u> in memory. This happens all the time. After a program is compiled, the program's relative (logical or virtual) addresses never change. However, real (physical or absolute) addresses change and depend on the program's load point.

Virtual Memory Management

Virtual Memory

It is a hypothetical memory created by a carefully orchestrated work of the OS and the computer hardware. It is a memory management technique - a concept. When it was introduced in 1970s, it was a technological breakthrough in computing. Still used in today's computers.

In this technique processes use **virtual memory address space**, a range of virtual addresses that processes may reference. The range of real addresses available on a particular computer system is called that computer's **real address space**.

Processes do not actually reside in virtual memory because it does not exist. The entire programs are physically stored in real memory and on disk. Active part of programs reside in real memory (active pages), whereas inactive pages are stored on disk.

Virtual addresses are different from real (physical) addresses and special-purpose hardware, **Memory Management Unit** (**MMU**), quickly maps virtual addresses to real addresses. This is called **dynamic address translation** (**DAT**). Dynamic address translation is necessary because virtual addresses are different from physical addresses.

Disadvantage

- overhead concerning address translation

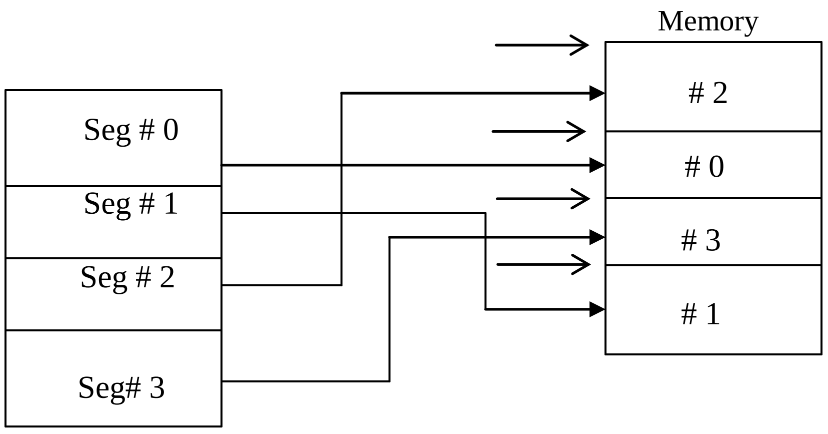
Advantages

- size of programs is not restricted by size of real memory
- numerous programs can be executed concurrently

Under virtual memory management, programs could be divided into

- **segments**
- **pages**
- **segments and pages**

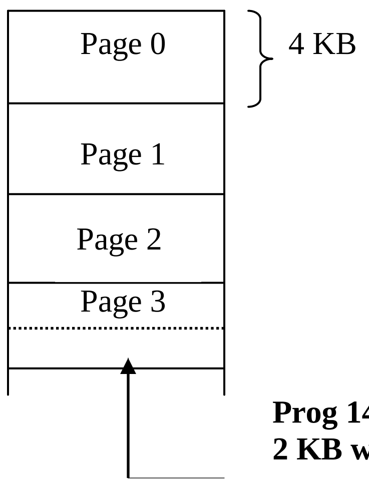
Segmentation



- a program can be segmented
- segments follow the logic of the program, may have various sizes
- programmer does segmenting

Each segment is loaded into available fragments (space) in memory. Instructions within a segment are addressed relative to the segment's beginning address, so segments can be loaded into noncontiguous memory. The OS has to maintain the beginning address of each segment. More often, however, segments have a fixed size, say, 64KB.

Paging



- fixed size small pages - 4KB
- easier to implement
- no external fragmentation
  (space on the last page may only be wasted)
- more oriented toward hardware
- starting addresses of pages have to be maintained

Prog 14KB
2 KB wasted due to internal fragmentation

Segmentation and Paging

- programs are divided into fixed size segments, and segments are further divided into fixed size pages

DYNAMIC ADDRESS TRANSLATION

In dynamic address translation for segmentation, paging, or segmentation and paging, virtual addresses are translated into real (physical) addresses. The mechanism of translation is implemented mainly in special-purpose hardware, called Memory Management Unit (MMU)

DYNAMIC ADDRESS TRANSLATION FOR SEGMENTATION

The program is chopped into small segments. How does the OS know where to load segments? The OS maintains the **table of free space.**

Ex. Program A → seg#0=10KB, seg#1=8KB, seg#2=50KB

Note that 1KB=1024 bytes.

Table of free space **before** loading Program A

Start address	length (size)	status of the fragment	
100K	10	0 (free)	← seg#0
110K	10	1 (busy)	
120K	15	0	← seg#1
135K	15	0	← too small
150K	60	0	← seg#2
210K			

Table of free space **after** loading Program A

Start address (in KB and bytes)	length (size in KB)	status of the fragment
100K = 102400	**10**	**1 (seg#0)**
110K = 112640	10	1
120K = 122880	**15**	**1 (seg#1)**
135K = 138240	15	0
150K = 153600	**60**	**1 (seg#2)**
210K = 215040		

Class activity #6

The OS must keep track of each segment entry point. Instructions are "displaced" by a certain amount from the beginning of each segment.

After program compilation and link-editing, the **virtual address** (the operand) of the instruction appears in the **two-part form**:

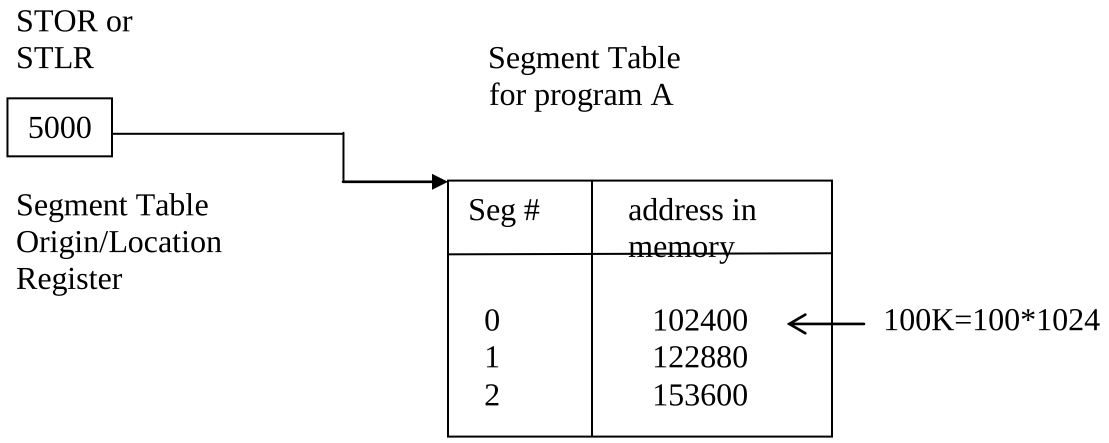
- segment #
- displacement/offset within a segment (distance from the beginning of the segment).

Two-part address structure

seg #	displacement

Segments, say, 0, 1, and 2 may have memory location (address) = 100. Assume now that you want to reference memory location 100. You need to tell in which segment: 0, 1, or 2. All three segments may have memory location 100.

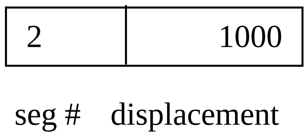
After the 3 segments have been loaded into memory, the OS creates and maintains the **segment table**, ordered by the segment #. The segment table contains the segment #s and segment's entry points (addresses).


STOR or STLR
5000
Segment Table Origin/Location Register
Segment Table for program A
100K=100*1024

Seg #	address in memory
0	102400
1	122880
2	153600

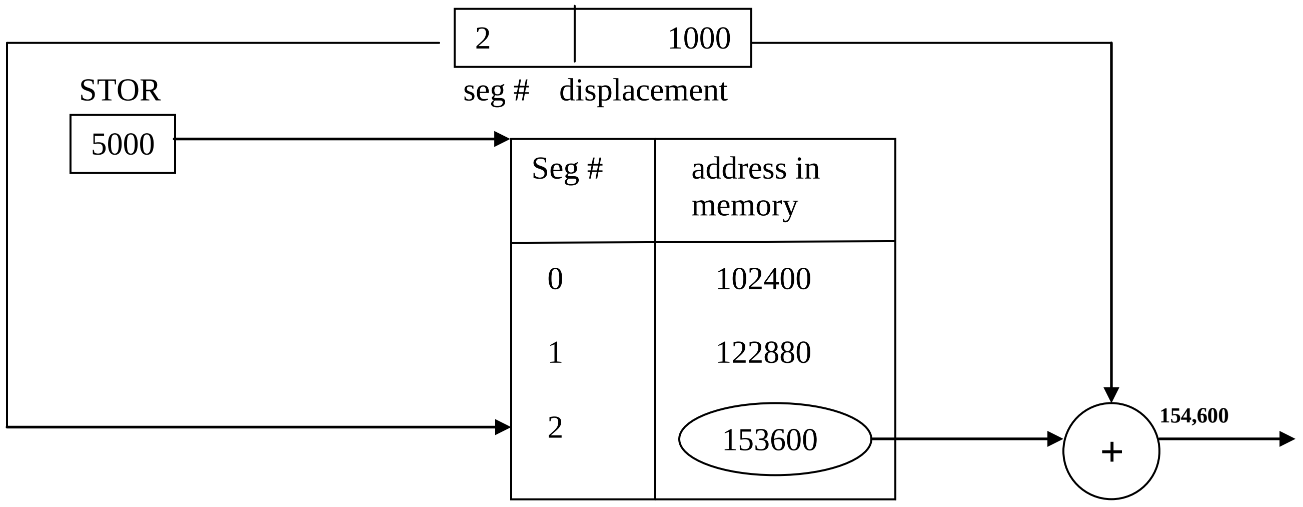
Each program has its own segment table.

Find the physical address of this virtual address



1. Find segment table at 5000, for instance
2. Do table look-up, seg #2 in virtual address points to the 3rd entry in the table, location 153600
3. Add 153600 to displacement $\rightarrow$ 153600+1000=154600 (physical address)

Pictorial presentation



Features

- length of segments vary leaving some unused space
- # of bits in the displacement portion of the address does set up an upper limit on segment size
- under segmentation alone, a program logic is a key to segmenting it

DYNAMIC ADDRESS TRANSLATION FOR <u>PAGING</u>

- program is broken into fixed length **pages**, size 4KB each, for example
- memory is divided into fixed size **page frames** (page size = page frame size)
- internal fragmentation no longer becomes a problem
- memory management is easier
- pages may be loaded into not adjacent memory locations

The OS maintains the **page frame table** (**page map table**) to track the status of each main memory frame.

Ex. Program A=14KB will be broken into 4 pages (Page 0, 1, 2, and 3): 4K+4K+4K+2K (2K is unused on page 3)

Sizes of pages:	128B	256B	512B
	1024B (1KB)	2048B (2KB)	**<u>4096B (4KB)</u>**

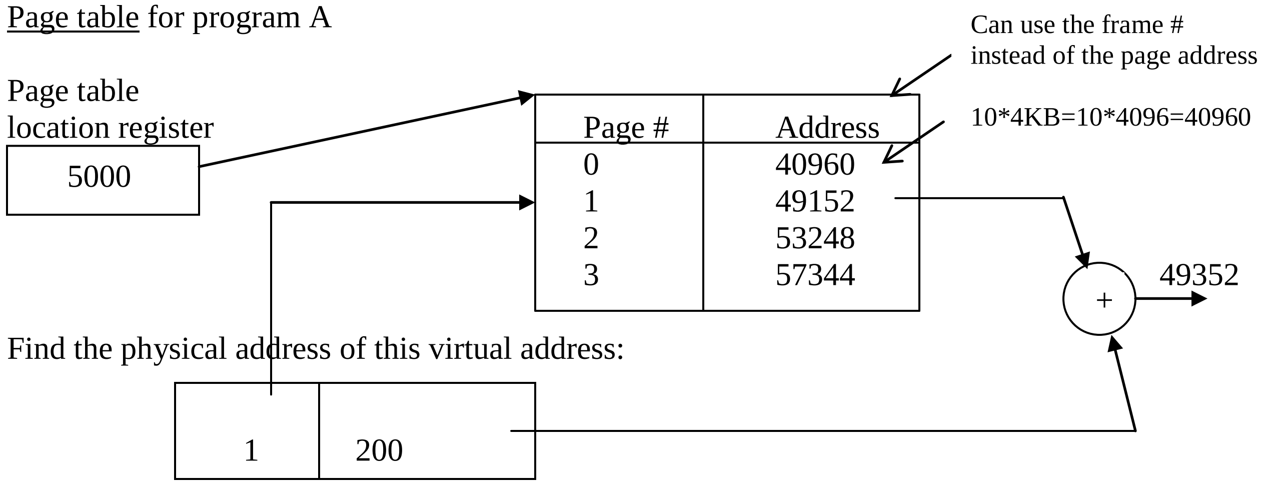
Page frame table **before** loading Program A

Page frame #	Program	Page #	Status of the frame
0-7	OS	0-7	1
8	B	0	1
9	C	0	1
10			0
11	C	1	1
12			0
13			0
14			0
15			0

Page frame table **after** loading Program A

Page frame #	Program	Page #	Status of the frame	
0-7	OS	0-7	1	
8	B	0	1	First fit strategy
9	C	0	1	
10	**A**	**0**	**1**	**(A-page 0)**
11	C	1	1	
12	**A**	**1**	**1**	**(A-page 1)**
13	**A**	**2**	**1**	**(A-page 2)**
14	**A**	**3**	**1**	**(A-page 3)**
15			0	

Page table for program A



Page #	Address
0	40960
1	49152
2	53248
3	57344

Page table location register
5000
Can use the frame # instead of the page address
10*4KB=10*4096=40960
+
49352
1
200

Find the physical address of this virtual address:

A page table is stored in memory. However, part of the page table or entire page table may be stored in associative memory called the **Translation Lookaside Buffer (TLB)** for very fast reference. It speeds up address translation. If the page address (page frame #) is recorded in TLB, address translation can proceed immediately - it is a hit. If the page address is not recorded in the TLB, the address translation is slower - it is a miss. The OS has to refer to the page table stored in main memory and place this address in TLB. The algorithms for the operation of TLB vary, but typically TLB stores the addresses (frame #s) of the most recently referenced pages.

Each process has its own page table. TLB typically contains addresses of pages belonging to different processes. In Windows XP, the size of TLB is 64KB.

DYNAMIC ADDRESS TRANSLATION FOR SEGMENTATION AND PAGING

(See page 1 in the document titled **“**CIS-350 (Real Memory Management and Virtual Memory Organization and Management) - Additional Figures.pdf” posted on Blackboard.)

A typical 24-bit address structure might be:

2	2	2000
segment #	page #	displacement

The “segment #” identifies the particular program segment in which the location is to be found; the “page #” identifies the page within the segment; and the “displacement” is the distance from the beginning of the page.

For each process, a segmentation and paging system requires a single segment table and more than one page table. In fact a page table will be used for each process segment.

Given the STOR (segment table origin register), segment table and page tables, a three-part virtual address shown above is translated as follows:

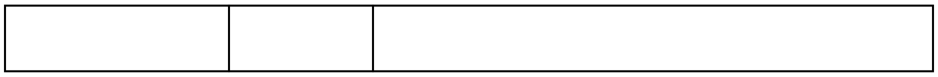
1. The STOR points to the origin of the process segment table - 6000.
2. Segment #2 points to the 2nd entry in the segment table. This gives the location of the page table for segment #2 as 80K.
3. Page #2 points to the 2nd entry in the page table - 208,000. This is the physical address of page #2 in memory.
4. Add: 208,000 + displacement ==> physical address
   208,000 + 2,000   ==> 210,000

(See page 1 in the document titled “CIS-350 (Real Memory Management and Virtual Memory Organization and Management) - Additional Figures.pdf” posted on Blackboard.)

VIRTUAL STORAGE

Program's address space may be structured into segments and pages.

For example, we might have a 24-bit virtual address structure:



8 bits seg #	4 bits page #	12 bits displacement
2^8=256 segments	2^4=16 pages within segment	2^{12}=4096 # of displacements or size of a page

Address space has at most 2^{24}=16,777,216 (more than 16 millions of possible locations)

Real storage is structured into page frames. The size of page frames is equal to the size of pages (for example, 4K).

Virtual memory is the address space, not a physical location for programs. It cannot be seen because it's a model (concept) of storage. Virtual memory does not exist anywhere. It does not store programs and data.

Programs and data are stored physically in real storage or auxiliary storage (external page device - disk) in a swap file.

Let’s assume that the size of the virtual address space is 32GB=2^{35} bytes and the size of the real storage is only 1GB=2^{30} bytes. One can see that the virtual address space is 32 times larger than the physical memory address space ($2^{35}/2^{30}=2^5$=32). In other words, the number of addresses |V| in the virtual memory is 32 times larger than the number of addresses |R| in the real/physical memory. Therefore, not all virtual address space are associated with physical/real memory addresses. Physical/real memory addresses and virtual addresses are not equivalent.

Analogy

Battleships - two players try to sink each other's ships by exchanging coded addresses.

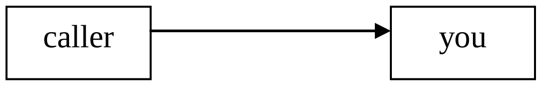
	1	2	3	4	5	6
A						
B			X			
C						
D						
E						
F						

Do these addresses actually exist? Or are they just a convenient way of mapping of imaginary body of water?

Call forwarding feature on a telephone

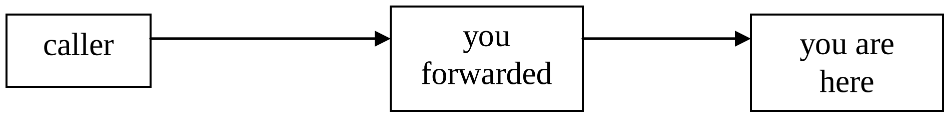
Does the caller have to know the number of the phone that actually reaches you?

No call forwarding feature:



He/she dials 245-2835 — reaches you at 245-2835

Call forwarding feature:



He/she dials 245-2835	245-2835 to 245-5509	245-5509
Process uses virtual address	translation (mapping)	physical/real/absolute address

Does a program need to know the physical/real address? NO!

Virtual addresses are relative and sequential. Physical locations will not be associated with these addresses until execution time.

If a particular data or instruction is referenced by an executing process, it has to be in real/physical storage.

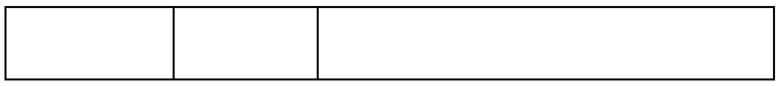
The virtual address is examined and the OS determines if the page on which the instruction is found is in real/physical memory/storage.

If it is, the virtual address is translated into physical address and the execution proceeds.

If it is not found, an appropriate page slot has to be identified and the page must be transferred to real/physical memory.

Assume that a process has been “loaded” into virtual storage and is executing.

A three-part virtual address of the instruction might have the following structure (like in the attached example – see the pdf file posted on Blackboard):



segment # page # displacement

DYNAMIC ADDRESS TRANSLATION FOR VIRTUAL ADDRESSES [PAGE IS OR IS NOT IN REAL (PHYSICAL) MEMORY]

(See page 2 in the document titled "CIS-350- (Real Memory Management and Virtual Memory Organization and Management) - Additional Figures.pdf" posted on Blackboard.)

Let's look how virtual addresses are interpreted and translated.

1. STOR (Segment Table Origin Register) points to the location of the segment table.

2. Seg #2 in virtual address points to the 2nd entry in the segment table, which in turns points to the location of the proper page table.

3. Page #4 (seg #2) gives the physical page frame location in real memory

4. If invalid bit=0, page frame location address + say, 1000, gives an absolute address of page #4 (seg #2)

5. Additional column entry "invalid bit"

   Not every page is actually present in a physical/real memory location at the time of reference

   - Invalid bit=0 (page is in real storage). We can translate the virtual address into physical address and execute the instruction.

   - Invalid bit=1 (page is not in real storage). It has to be brought in. We cannot immediately translate the virtual address into physical address and execute the instruction until the page is swapped in to the physical memory.

We maintain the invalid bit column because the virtual address space is larger than the physical/real address space. It indicates if the page is in physical/real memory or not.

Suppose invalid bit=1. We made a reference to a page which does not exist yet in physical/real memory (**page fault**). An interrupt occurs and page-in operation has to take place.

1. Control is passed to the supervisor which searches the page frame table to find an available page frame (with 0).

2. Location of page in external storage is identified.

3. Page is paged-in to physical/real storage.

4. The page table entry 4 invalid bit is set to 0.

5. the page frame table entry is updated from 0 to 1. So, the virtual address is translated into physical/real address and the execution resumes.

What if all bits in the page frame table are set to 1? It means that no available page frame is found. The page which is needed must replace one which is currently in the main storage. This procedure is called **swapping**.

In order to choose the page to be replaced, the system will use, for example, the **Least Recently Used** (**LRU**) rule. In other words, the pages which have not been referenced recently will be paged-out.

Again, like for paging, the **Translation Lookaside Buffer** (sometimes called an associative array register) keeps addresses of the most recently used pages, so address translation is speeded up very much for each instruction.

Bringing pages to memory only when they are referenced is called **demand paging** (demand fetch).

Prepaging (anticipatory paging/fetch) is bringing a new page into memory before it is actually needed. It is predicting a demand for a new page. This can speed up the execution of a process.

If a process is executing instructions from page #1, it is likely that next instructions will be on page #2, so page #2 will be brought before it is needed.

In the case we have just discussed, we have demand paging.

Excessive swapping-in and out may result in so called **thrashing** which may substantially decrease the system performance.

How many page frames should be assigned to a new process just entering the system to avoid an excessive number of page faults?

The more page frames assigned to a process, the less page faults would occur. But, how about other processes are treated? If the OS assigns many page frames to one process there will be fewer frames available for other processes.

Experimentation performed showed that during execution processes stay within small areas of memory during any given period of time. The areas themselves, however, change over time. This property is called the **concept of locality**.

It is possible to establish the minimum number of page frames, called a **working set**, that a process should be assigned to satisfy the concept of locality. Page faults would only occur when the local area used by the process moves.

Advantages of virtual storage

- fragmentation of real storage is minimized
- applications may be processed which require much more storage space than is available in the computer's physical/real storage
- more processes can be processed since the entire processes do not have to be in memory, only portions of them
- increases the CPU and overall system performance; the idle time is minimized

Disadvantages:

- may decrease performance if used carelessly (too small working set, thrashing)

- overhead (address translation) – implemented in special-purpose hardware by Memory Management Unit (MMU)
- an individual process executes slower, but throughput of the entire system is increased

(Also see pages 4 and 5 in the document titled “CIS-350 (Real Memory Management and Virtual Memory Organization and Management) - Additional Figures.pdf” posted on Blackboard.)