

# Chapter 8 - Virtual Memory

Luis Tarrataca

`luis.tarrataca@gmail.com`

CEFET-RJ

## 1 Motivation

## 2 Operating System Functions

## 3 OS a resource Manager

### Scheduling

Process

Process States

Process Control Block

### Scheduling Techniques

## 4 Memory Management

Memory Swapping

Memory Partitioning

Paging

Virtual Memory

- Demand paging

- Page Table Structure

Translation Lookaside Buffer

Translation Lookaside Buffer



5 Where to focus your study

6 References

# Motivation

In computer architecture we have a series of components:

- CPU
- Memory
- Bus
- Pipeline
- I/O module
  - USB;
  - SCSI;
  - SATA.
  - etc...

These components interact with each other at the hardware level.

Who is responsible for managing these resources? Any ideas?

Who is responsible for managing these resources? Any ideas?

**Operating System** is a program that:

- Manages the computer's resources;
- Schedules the execution of other programs;
- Acts as an interface between applications and the computer hardware;

What are the main functions of an OS?

## What are the main functions of an OS?

- An application is expressed in a programming language;
- But it would be difficult to have to worry about all computer components.
- To ease this task, the OS makes available a set of systems programs:
  - Frequently used functions that assist in program creation;
  - The management of files;
  - And the control of I/O devices.



# Operating System Functions (1/2)

Operating system:

- Masks the details of the hardware from the programmer;
- Provides the programmer with a convenient interface for using the system;
- Acts as mediator:
  - Easier for the programmer/application to use resources.

# Operating System Functions (2/2)

OS typically provides services in the following areas:

- **Program execution:**

- Instructions and data must be loaded into main memory;
- I/O devices and files must be initialized;
- and other resources must be prepared.

# OS as a resource Manager (1/2)

Computer is a set of resources for:

- Moving, storing and processing data;
- And for the control of these functions;

OS is responsible for managing these resources:

- OS is in control of the computer's basic functions;
- But this control is exercised in a curious way...

# OS as a resource Manager (1/2)

OS acts as an unusual control mechanism, *i.e.*:

- OS provides instructions for the processor;
  - OS is also a computer program that is executed by the processor;
- Key difference is in the intent of the program:
  - OS directs the processor in the use of system resources;
  - OS directs the timing of processor execution of other programs;
    - 1 OS must cease executing;
    - 2 OS must allow for other programs to execute;
    - 3 Eventually OS will regain control of the processor;

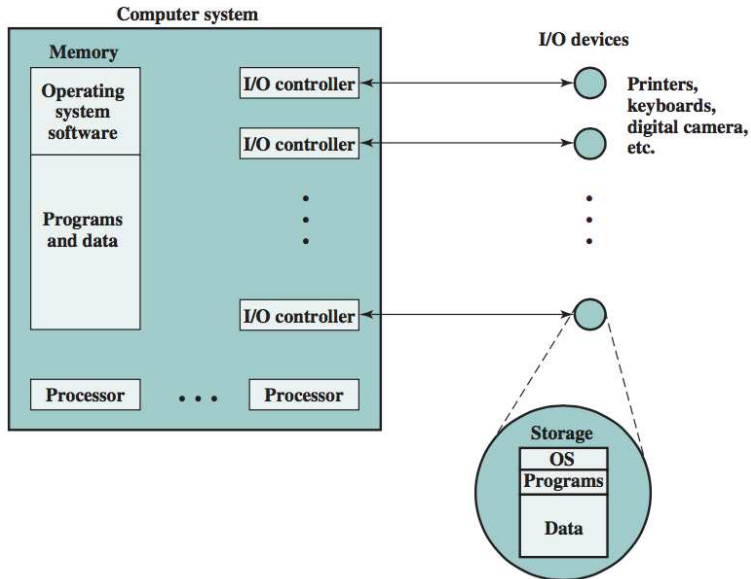


Figure: The operating system as a resource manager (Source: (Stallings, 2015))

Main memory contains:

- A portion of the OS:
  - **Kernel:** most frequently used functions in the OS;
  - But also other OS portions that may be in use;
- User programs and data;
- Main memory is controlled jointly by:
  - OS and memory-management hardware in the processor;

OS is responsible for determining:

- When an I/O device can be used by a program;
- Access to and use of files.
- How processor time is split for program execution;

# Scheduling

Lets take a look at another dimension of the OS:

- Imagine we want to execute a single program;
- but this program sometimes will have to wait for I/Os;

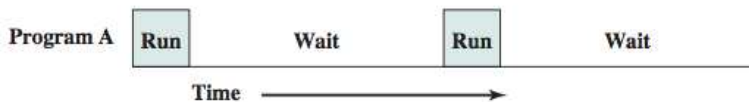


Figure: Executing a single program (Source: (Stallings, 2015))

Is this an efficient use of the processor?



Most of the time the processor is idle not doing anything.

- Processor executes orders of magnitude faster than I/O...
- Consider the following example:

Read one record from file	15 $\mu$ s
Execute 100 instructions	1 $\mu$ s
Write one record to file	<u>15 <math>\mu</math>s</u>
TOTAL	31 $\mu$ s

$$\text{Percent CPU utilization} = \frac{1}{31} = 0.032 = 3.2\%$$

Figure: System utilisation Example (Source: (Stallings, 2015))

Instead of idling the system we could be running another program...

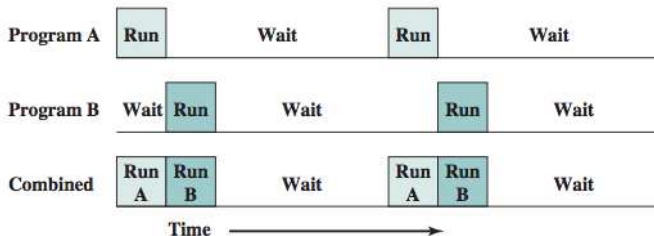


Figure: Executing two programs (Source: (Stallings, 2015))

But this second program may eventually also ask for I/Os...

We can even add a third program...

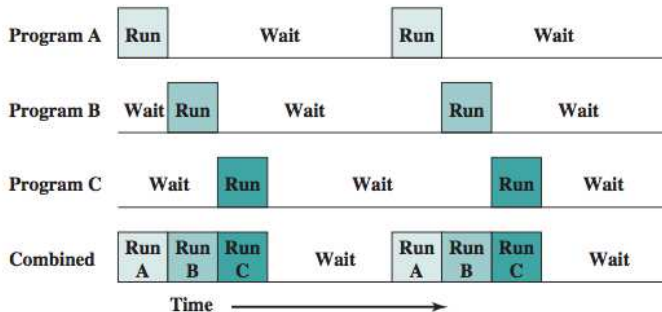


Figure: Executing three programs (Source: (Stallings, 2015))

This way the processor idle times are diminished...

The component responsible for program switching is the **scheduler**:

- In reality the scheduler switches between processes;
- What is a **process**?
  - Executable code;
  - Memory
    - variables, data, etc;
  - Call stack
    - to keep track of active subroutines and/or other events;
  - Operating system descriptors of resources
    - files, sockets, etc;
  - Processor context;

# Process States (1/2)

Process state will change over its lifetime:

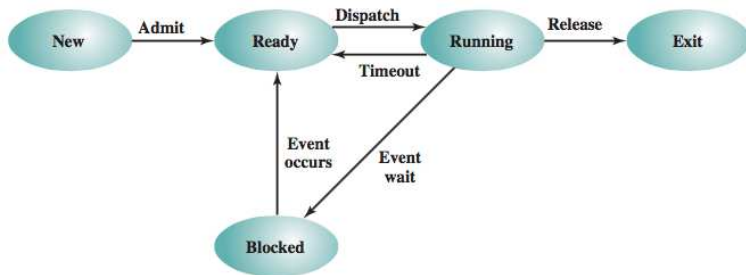


Figure: Five state process model (Source: (Stallings, 2015))

## Process States (2/2)

Process state will change over its lifetime:

- **New:** Process is created but not yet ready to execute.
- **Ready:** Process is ready to execute, awaiting processor availability;
- **Running:** Process is being executed by the processor;
- **Waiting:** Process is suspended from execution waiting a system resource;
- **Halted:** Process has terminated and will be destroyed by the OS.

# Process Control Block

OS represents each process by a control block:

- **Identifier:** Unique process identifier;
- **State:** Current process state;
- **Priority:** Process priority level.;
- **Program counter:** Next instruction;
- **Memory pointers:** Process starting and ending memory locations;
- **Context data:** Processor state registers;
- **I/O status:** I/O requests and I/O devices;
- **Accounting Info:** *E.g.* processor time, clock time, time limits,...

Identifier
State
Priority
Program counter
Memory pointers
Context data
I/O status information
Accounting information
⋮

Figure: Process Control Block (Source: (Stallings, 2015))

# Scheduling Techniques

Consider the following scenario:

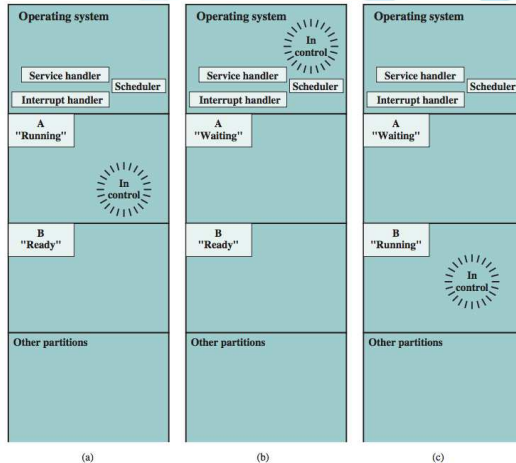


Figure: Scheduling Example (Source: (Stallings, 2015))



Initially process **A** is running and:

- 1 The processor is executing instructions from process **A**;
- 2 The processor then:
  - Ceases to execute **A**;
  - Begins executing OS instructions.
- 3 This will happen for one of three reasons:
  - 1 Process A issues a service call (e.g., an I/O request) to the OS.
    - Execution of A is suspended until this call is satisfied by the OS.
  - 2 Process A causes an interrupt signal:
    - When this signal is detected, the processor ceases to execute A;
    - OS processes the interrupt signal;
  - 3 An event unrelated to process A causes an interrupt.
    - *E.g.* is the completion of an I/O operation.

Process **A** therefore is going to block and control is passed to the OS:

- 1 The OS saves:
  - Current processor context (registers);
  - PC;
- 2 The OS:
  - 1 changes the state of **A** to **blocked**;
  - 2 decides which process should be executed next;
  - 3 instructs the processor to restore B's context data;
  - 4 proceeds with the execution of B where it left off.

## Interruption examples:

- Memory error:
  - process attempts to access unauthorised memory location;
- Instruction error:
  - process attempts to execute a privileged instructions;
- Timeout:
  - each process is granted the processor for a short period at a time.

# Memory Management

Now that we have seen processes consider the following:

- Few processes → high processor idle times → not efficient;
- Many processes → little processor idle times → efficient;
- Objective:

Memory needs to be allocated efficiently to pack as many processes into memory as possible.

How does the OS handle the memory from each program? Any ideas?

# Memory Swapping

Consider the following scenario:

- There are multiple processes to execute;
- It is possible to perform process switching;
- Processor is much faster than I/O:
- Even with multiprogramming:
  - Processor could be idle most of the time.

What can we do to pack into main memory as many processes as possible? Any ideas?

What can we do to pack into main memory as many processes as possible?

- Increase main memory (RAM);
  - Expensive...;
  - Any ideas?



What can we do to pack into main memory as many processes as possible?

- Increase main memory (RAM);
  - Expensive...;
- Can we use any other type of memory?

What can we do to pack into main memory as many processes as possible?

- Increase main memory (RAM);
  - Expensive...;
- Can we use any other type of memory?
  - Well, the hard drive is a type of memory ;)

**Idea:** Use the hard drive as memory for processes:

- Eventually: main memory will be full;
- Rather than the processor remain idle the OS:
  - Swaps one of the blocked processes back out to disk;
  - Selects one of the processes stored in disk to go to main memory;
  - Execution then continues with the new process.
- This procedure is called **memory swapping**.

Can you see any potential problem with memory swapping?

Can you see any potential problem with memory swapping?

- Well we are trying to minimize processor idle times;
- These usually happen when I/O operations occur;
  - Accessing the hard disk is an I/O operation...
  - But because disk I/O is generally the fastest I/O on a system:
    - swapping will usually enhance performance.

# Memory Partitioning (1/2)

How should the OS partition the memory?

- Should every process have the same amount of memory?
- But what if we need less/more space?

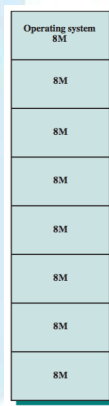
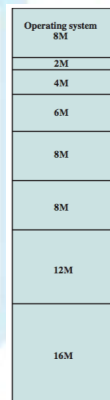


Figure: Equal size partitions (Source: (Stallings, 2015))

# Memory Partitioning (2/2)

How should the OS partition the memory?

- Or should different processes have different amounts of memory?
- When a process is brought into memory, it is placed in the smallest available partition that will hold it.



Can you see any problem with this type of partitioning?



Can you see any problem with this type of partitioning?

- Wasted memory: even with the use of unequal fixed-size partitions;
- In most cases:
  - A process will not require as much memory as provided by the partition;
  - *E.g.* a process that requires 3M bytes of memory would be placed in the 4M partition, wasting 1M that could be used by another process...

Can you think of an alternative method for partitioning memory?

Can you think of an alternative method for partitioning memory?

- What about **variable-size partitions**:
  - When a process is brought into memory:
    - Allocate exactly as much memory as it requires and no more.

- What about **variable-size partitions**:
  - When a process is brought into memory:
    - Allocate exactly as much memory as it requires and no more.

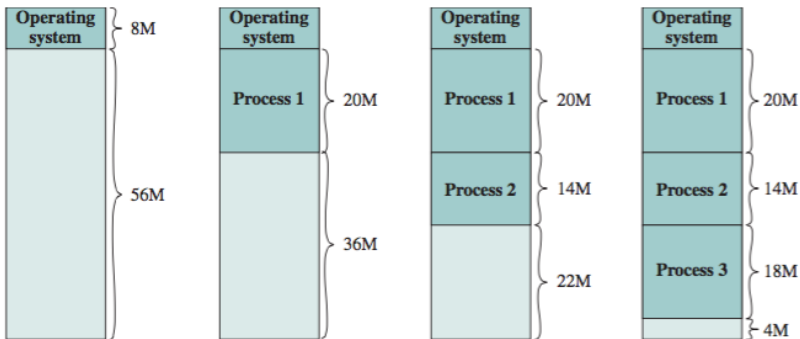


Figure: Variable-size partitions (Source: (Stallings, 2015))

Can you see any problems with this type of partitioning scheme?

Can you see any problems with this type of partitioning scheme?

- This method starts out well:
  - However, eventually the memory will be full of holes.;
  - This happens because the processes either:
    - Terminate;
    - are removed from main to secondary memory (HD, SSD, etc..).
  - From time to time:
    - OS **compacts** the processes in memory;
    - This results in all the free memory being placed together in one block;
    - **This is a time-consuming procedure, wasteful of processor time.**

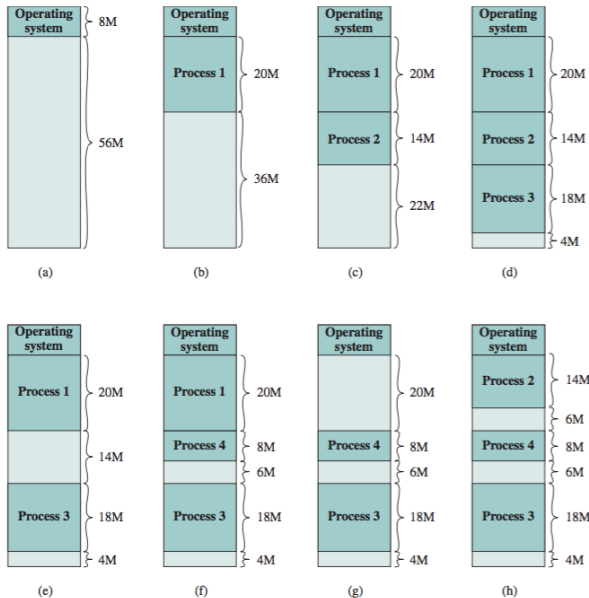


Figure: The effects of dynamic partitioning (Source: (Stallings, 2015))

## Overall conclusion:

- Fixed-size and variable-size partitions are inefficient in the use of memory.

Can we do any better than these types of partitioning schemes?



## Overall conclusion:

- Fixed-size and variable-size partitions are inefficient in the use of memory.

Can we do any better than these types of partitioning schemes?

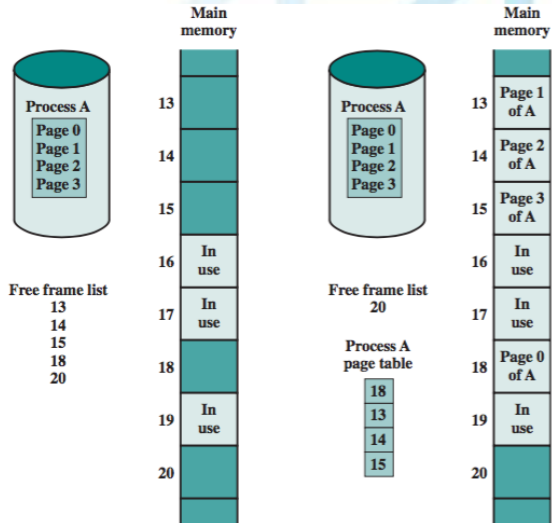
- Yes, we can through a mechanism called **paging**

# Paging

Consider an alternative partitioning scheme:

- Allow memory to be partitioned into equal fixed-size small chunks:
  - known as **page frames**
- Each process is also divided into small fixed-size chunks of some size:
  - known as **pages**
- Each **page** can be assigned to a **page frame**, then:
  - At most, wasted space for a process will be a fraction of the last page.

# Example



(a) Before

(b) After

At a given point in time:

- some of the frames in memory are in use and some are free;
- the **list of free frames** is maintained by the OS;
- process **A**, stored on disk, consists of four pages.
- When it comes time to load this process the OS:
  - finds four free frames;
  - loads the four pages of the process A into the four frames.

Do the frames need to be contiguous (1/2)?

- No they do not. We can use the concept of **logical address**.
- OS maintains a page table for each process:
  - Showing the frame location for each page of the process;
- Within the process each logical address consists of:
  - a page number and a relative address within the page;
- Logical- to-physical address translation is done by processor.

Do the frames need to be contiguous (2/2)?

- Processor must know how to access the process's page table:
  - Input is a logical address:
    - **(page number, relative address)**
  - Output is a physical address obtained through the process page table:
    - **(frame number, relative address)**

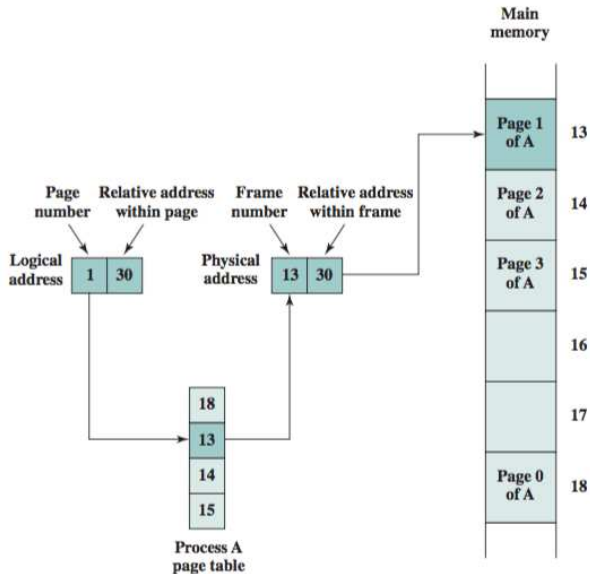


Figure: Logical and physical addresses (Source: (Stallings, 2015))

Can you see any other improvement that can be done to memory management?



Can you see any other improvement that can be done to memory management?

- With the previous scheme:
  - OS always loads all the memory of a process;

Can you see any other improvement that can be done to memory management?

- With the previous scheme:
  - OS always loads all the memory of a process;

Do we really need to always load an entire program?

Do we really need to always load an entire program?

- Space-Time locality principle...

Do we really need to always load an entire program?

- Space-Time locality principle...
- **Idea:**
  - What if we only load those pages that are required at a single moment?
  - This is the concept of **virtual memory**

# Virtual Memory

Each process page is brought in only when it is needed (1/2):

- 1 Procedure is known as **demand paging**;
- 2 Locality principle:
  - Same values, or related storage locations, are frequently accessed.
  - Why then would we need to load every page? Wasteful...

Each process page is brought in only when it is needed (2/2):

- 3 We can make better use of memory by loading in just a few pages
- 4 If the program attempts to access a page not in main memory:
  - a **page fault** is triggered: OS brings in the desired page;
  - These pages reside in secondary memory;
- 5 **Virtual Memory** refers to this much larger memory usable by the program.

Can you see any implication of using virtual memory? Any ideas?

Can you see any implication of using virtual memory? Any ideas?

At any one time, only a few pages of a process are in memory:

- Therefore more processes can be maintained in memory.
- Time is saved because:
  - Unused pages are not swapped in and out of memory;
  - Less RAM/HD accesses;
- **Consequence:**
  - Possible for a process to be larger than all of main memory.



## OS must be clever about how it manages this scheme: (1/2)

- When it brings one page in, it must throw another page out;
  - this is known as page replacement.

## OS must be clever about how it manages this scheme: (2/2)

- OS might throw out a page just before it is about to be used:
  - OS will just have to get that page again almost immediately;
  - Too much of this leads to a condition known as **thrashing**:
    - Processor spends most of its time swapping pages...
    - ...rather than executing instructions
    - extremely slowwww computerrrr...

Do you have any idea how to solve this problem? Any ideas?

Do you have any idea how to solve this problem? Any ideas?

- OS needs to guess which pages are least likely to be used:
  - *E.g.* based on recent history.
  - We have seen some when we studied cache systems:
    - FIFO;
    - LFU;
    - LRU;

## Example (1/2)

VAX architecture:

- Each process can have  $2^{31}$  bytes of virtual memory;
- Each process has pages of size  $2^9$  bytes
- Therefore we need to index  $\frac{2^{31}}{2^9} = 2^{22}$  pages per process...
  - Amount of memory devoted to page tables would be very high;
  - How can we solve this problem?

## Example (2/2)

VAX architecture:

- Each process can have  $2^{31}$  bytes of virtual memory;
- Each process has pages of size  $2^9$  bytes
- Therefore we need to index  $\frac{2^{31}}{2^9} = 2^{22}$  pages...
  - Amount of memory devoted to page tables is very high;
  - How can we solve this problem?
    - We use virtual memory for processes;
    - **We can also use virtual memory for process page tables;**
    - This way we only load entries on a on-demand basis.

# Translation Lookaside Buffer

Every virtual memory reference can cause two physical memory accesses:

- One to fetch the appropriate page table entry;
- And one to fetch the desired data.

Can you see any problem with this?

# Translation Lookaside Buffer

Every virtual memory reference can cause two physical memory accesses:

- one to fetch the appropriate page table entry;
- and one to fetch the desired data.

Can you see any problem with this?

- We are doubling the number of memory accesses;
- Therefore we are also doubling the memory access time...



What do we always do when we need to reduce memory accesses?

What do we always do when we need to reduce memory accesses?

- Use cache for page table **entries** ;)
- Translation lookaside buffer (**TLB**)
  - Functions in the same way as a memory cache;
  - Contains those page table entries that have been most recently used.
  - Physical piece of hardware

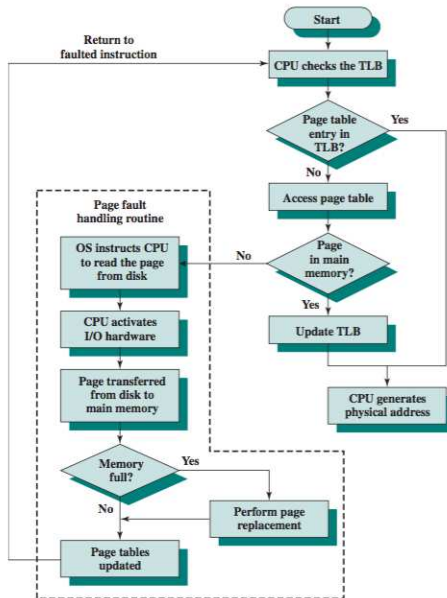


Figure: Operation of paging and translation look aside buffer (Source: (Stallings, 2015))

A virtual address will have the form (page number, offset).

- 1 Memory consults TLB to see if the matching page table entry is present:
  - **If it is:** physical address is generated;
  - **If not:** entry is accessed from a page table.
- 2 Once the real address is generated:
  - Cache is consulted to see if the block containing that word is present:
    - Do not confuse this cache with the TLB, different “caches”;
    - If the word is present, it is returned to the processor;
    - If not, the word is retrieved from main memory.

Virtual memory must interact with system cache (1/2):

1 TLB is checked to see if page table entry is present:

- If yes: physical address is generated;
- If not: entry is accessed from a page table;

2 Once physical address is generated:

- Cache is checked to see if the block containing that word is present:
  - If yes: word is return to processor;
  - If not: word is fetched from main memory;

Virtual memory must interact with system cache (2/2):

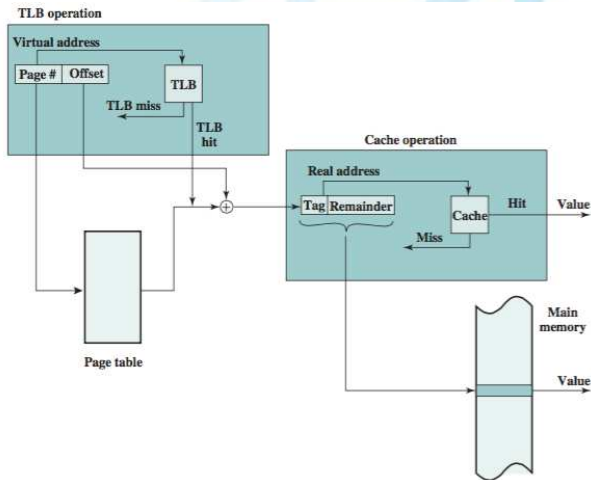


Figure: Translation lookaside buffer and cache operation (Source: (Stallings, 2015))

Notice the **complexity** of a single memory reference:

- Virtual address needs to be translated to physical address:
  - This involves reference to a page table which may be:
    - in the TLB, in main memory, or on disk.
- Once the physical address of the word is obtained:
  - word may be in cache, in main memory, or on disk

# Where to focus your study

After this class you should be able to:

- Summarize, at a top level, the key functions of OS;
- Explain the concept of scheduling;
- Understand the reason for memory partitioning ;
- Explain the various techniques for memory partitioning;
- Define virtual memory;
- Assess the relative advantages of paging.



Less important to know how these solutions were implemented:

- details of specific memory management units;

Your focus should always be on the building blocks for developing a solution  
=>

# References I



Stallings, W. (2015).

*Computer Organization and Architecture.*

Pearson Education.