

# Chapter 3 - Memory Management

Luis Tarrataca

`luis.tarrataca@gmail.com`

CEFET-RJ

## 1 A Memory Abstraction: Address Spaces

The Notion of an Address Space

Swapping

Managing Free Memory

Memory Management with Bitmaps

Memory Management with Linked Lists

## 2 Virtual Memory

Paging

Page Tables

Structure of a Page Table Entry

Speeding Up Paging

Translation Lookaside Buffer

Page Tables for Large Memories

Multilevel Page Tables

Inverted Page Tables

### 3 Page Replacement Algorithms

Optimal Page Replacement Algorithm

Not Recently Used Page Replacement Algorithm

First-In, First-Out (FIFO) Page Replacement Algorithm

Second-Chance Page Replacement Algorithm

Clock Page Replacement Algorithm

Least Recently Used (LRU) Page Replacement Algorithm

Working Set Page Replacement Algorithm

WSClock Page Replacement Algorithm

Summary of Page Replacement Algorithms

## 4 Design Issues For Paging Systems

Local versus Global Allocation Policies

Load Control

Page Size

Shared Pages

Cleaning Policy

## 5 References

# The Notion of an Address Space

Lets start with some basic questions:

What is an address space? Any ideas?

In the context of multi-programming:

What are the main objectives that we need to guarantee for a program's address space? Any ideas?

In the context of multi-programming:

What are the main objectives that we need to guarantee for a program's address space? Any ideas?

Two problems have to be solved to allow multi-programming:

- **Protection**
- **Relocation**

What does this mean: **protection** and **relocation**? Any ideas?



Two problems have to be solved to allow multi-programming:

- **Protection:**

- How to guarantee that programs do not overwrite each other's memory space?

- **Relocation:**

- How to relocate programs in order to avoid overwriting each other's memory space?

Why is relocation important? Any ideas? (See Slide 20)

Solving these problems required a new abstraction, the **address space**:

- Set of addresses that a process can use to address memory;
- Each process has its own address space:
  - Independent of those belonging to other processes;
  - Except when processes want to share their address spaces

Multiprogramming allows for multiple programs to coexist:

How can we give each program its own address space? Any ideas?

Example:

- Address 28 in one program maps to physical address X
- Address 28 in another program maps to physical address Y

How can we give each program its own address space? Any ideas?

**Answer:** Equip each CPU with two special hardware registers:

- **Base register:** loaded with the physical address where program begins;
- **Limit register:** loaded with the program's length;

Every time a process references memory the CPU:

- Automatically adds **base** value to the address generated by the process:
  - Before sending the address out on the memory bus;
- Checks whether the address offered is  $\leq$  than the **limit** value:
  - In which case a fault is generated and the access is aborted;

If the original program starts in memory address 16384:

- Then the following address requested by the program:

```
JMP 28 // Relative address;
```

- Would be converted to:

```
JMP 16412 // Absolute address (28 + 16384);
```

Have you ever heard of a **segmentation fault**?

What do you think a segmentation fault is? Any ideas?

What do you think a segmentation fault is? Any ideas?

When a program attempts to:

- Access a memory location that it is not allowed to access;
- Attempts to access a memory location in a way that is not allowed:
  - *E.g.*: write to a read-only location;



# Swapping

In a typical computer: total amount of RAM needed by the processes:

- Often more than can fit in memory;

On a typical Windows, OS X, or Linux system:

- 50 - 100 processes may be started up as soon as the computer is booted:
  - network connections, software update, etc...
- All of this before the first user program is loaded;

## Conclusion:

- Keeping all processes in **RAM** would require a huge amount;
- Usually this cannot be done since there is insufficient memory.

What can be done to deal with memory overloading? Any ideas?

What can be done to deal with memory overloading? Any ideas?

Two approaches to dealing with memory overload exist:

- **Swapping:**

- Load entire process into memory, run, then transfer to disk;
- Idle / blocked processes are mostly stored on disk;

- **Virtual memory:**

- Allows programs to run even when they are only partially in main memory;
- Load only pages that are required at a given instant in time;

# Swapping example

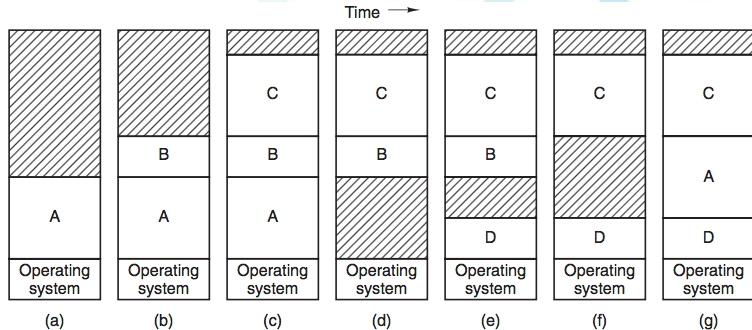


Figure: Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory. (Source: (Tanenbaum and Bos, 2015))

From the previous figure:

- 1 Initially, only process A is in memory;
- 2 Then processes B and C are created or swapped in from disk;
- 3 Process A is then swapped out to disk;
- 4 Then process D comes into memory;
- 5 Process B is removed from memory;
- 6 Finally, process A comes in again:
  - A is at a different location: addresses must be **relocated**;
  - For example, base and limit registers would work fine here;

Can you see any problem with the previous example? Any ideas?

Can you see any problem with the previous example? Any ideas?

Swapping creates multiple **holes** in memory...

Can you see any problem with the previous example? Any ideas?

Swapping creates multiple holes in memory...

What can be done to solve the memory holes introduced by swapping?



Can you see any problem with the previous example? Any ideas?

Swapping creates multiple holes in memory...

What can be done to solve the memory holes introduced by swapping?

Memory compaction...

Can you see any problem with the previous example? Any ideas?

Swapping creates multiple holes in memory...

What can be done to solve the memory holes introduced by swapping?

Memory compaction...

Can you see any problem with memory compaction?

Can you see any problem with the previous example? Any ideas?

Swapping creates multiple holes in memory...

What can be done to solve the memory holes introduced by swapping?

Memory compaction...

Can you see any problem with memory compaction?

Requires a lot of CPU time.... =(

Also, how much memory should be allocated for a process when it is created or swapped in?

If processes are created with a fixed size that never changes:

- Simple: OS allocates exactly what is needed;
- Unrealistic to assume that processes will not change in size;

Why is it unrealistic to assume that a process will never change in size?  
Any ideas?

Why is it unrealistic to assume that a process will never change in size?  
Any ideas?

Several examples:

- Process call stack...
- Data structure changes...
- Etc...

If processes data segments can grow (1/3):

- If a **memory hole** is adjacent:
  - Process can be allowed to grow into the hole;

If processes data segments can grow (2/3):

- If another process is **adjacent**:
  - Growing process will have to be moved to a large enough hole...
  - **Or** one or more processes will have to be swapped out;



If processes data segments can grow (3/3):

- If a process cannot grow in memory and the swap area on the disk is full:
  - Process will have to be suspended until some space is freed up;
  - or growing process can also be killed...

If most processes are expected to grow:

- Probably a good idea to allocate a little extra memory;
- Reduces overhead associated with:
  - Moving or swapping processes that no longer fit in their allocated memory.

Example in which space for growth has been allocated to two processes:

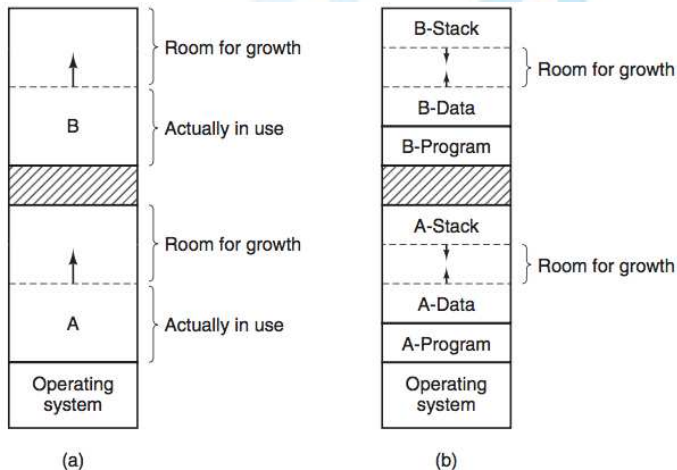


Figure: . (a) Allocating space for a growing data segment. (b) Allocating space for a growing stack and a growing data segment. (Source: (Tanenbaum and Bos, 2015))

In the previous examples why are the stack and data segments growing in opposite directions? Any ideas?

In the previous examples why are the stack and data segments growing in opposite directions? Any ideas?

Best use of the available room for growth:

- Just a single growth segment is need;
- Instead of having:
  - Stack growth segment;
  - Data growth segment,

# Managing Free Memory

When memory is assigned dynamically the OS must manage it:

How can the OS manage memory usage? Any ideas?

# Managing Free Memory

When memory is assigned dynamically the OS must manage it:

How can the OS manage memory usage? Any ideas?

- There are two ways to keep track of memory usage:
  - **Bitmaps**
  - **Free Lists**

Lets have a look at these two methods...

# Memory Management with Bitmaps

Memory is divided into **allocation units**:

- As small as a few words and as large as several kilobytes;
- Corresponding to each allocation unit is a bit in the **bitmap**:

What is a bitmap? Any ideas?



# Memory Management with Bitmaps

Memory is divided into **allocation units**:

- As small as a few words and as large as several kilobytes;
- Corresponding to each allocation unit is a bit in the **bitmap**:
  - With value 0 if the unit is free;
  - With value 1 if the unit is occupied;

# Example

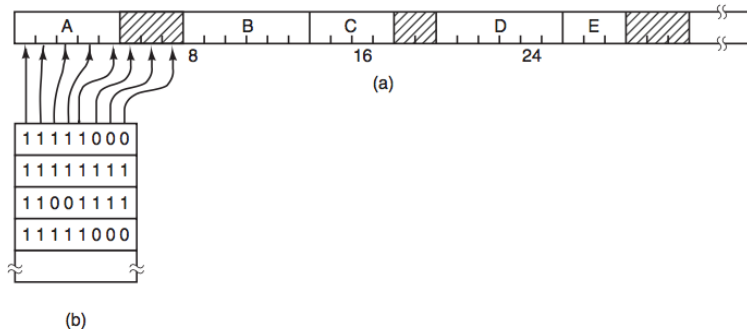


Figure: (a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (Source: (Tanenbaum and Bos, 2015))

From the previous figure:

- 11111000
  - Positions 1 through 5 are occupied;
  - Position 6 through 8 are free;
- 11111111
  - Positions 9 to 16 are occupied;
- 11001111
  - Position 17 and 18 are occupied;
  - Position 19 and 20 are free;
  - Positions 21 through 24 are occupied;
- 11111000
  - Positions 25 through 29 are occupied;
  - Positions 30 through 32 are free;

Can you say anything about the **size** of the allocation unit? Any ideas?

*Example:*

- Allocation unit represents 16 bytes;
- Allocation unit represents 32 bytes;
- Allocation unit represents 64 bytes;
- Etc...

Can you say anything about the **size** of the allocation unit? Any ideas?

**Size** of the allocation unit is an important design issue:

- The **smaller** the allocation unit, the **larger** the bitmap:
  - Multiple allocation units may be required for a process;
- The **larger** the allocation unit, the **smaller** the bitmap:
  - May result in wasted space if process is small. Why?

Can you see any problems with using bitmaps? Any ideas?

Can you see any problems with using bitmaps? Any ideas?

- When a  $k$ -unit process needs to be brought into memory:
  - OS must search for a  $k$ -unit **consecutive** space;
  - Searching a bitmap for a space of a given length is a slow operation;

Can you see any problems with using bitmaps? Any ideas?

- When a  $k$ -unit process needs to be brought into memory:
  - OS must search for a  $k$ -unit **consecutive** space;
  - Searching a bitmap for a space of a given length is a slow operation;

What can be done to solve this problem? Any ideas?



Can you see any problems with using bitmaps? Any ideas?

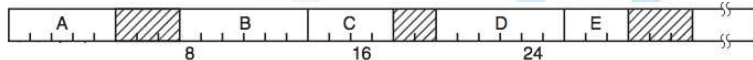
- When a  $k$ -unit process needs to be brought into memory:
  - OS must search for a  $k$ -unit **consecutive** space;
  - Searching a bitmap for a space of a given length is a slow operation;

What can be done to solve this problem? Any ideas?

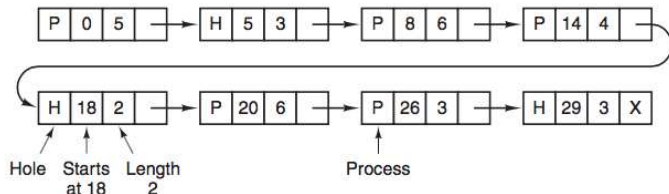
- Memory Management with Linked Lists;

# Memory Management with Linked Lists

Track of memory through a list of allocated/free memory segments:



(a)



(c)

Figure: (a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (c) The same information as a list. (Source: (Tanenbaum and Bos, 2015))

From the previous figure each list **entry** specifies:

- **H** specifies a **hole**;
- **P** specifies a **process**;
- Starting address;
- Length;
- Pointer to next list item;

How should this list be managed when a process terminates or is swapped out? Any ideas?

In the previous example **segment list** is kept **sorted** by address:

- Terminating process normally has two neighbours:
  - Except when it is at the very top or bottom of memory;
  - Neighbours may be either processes or holes, leading to four combinations:

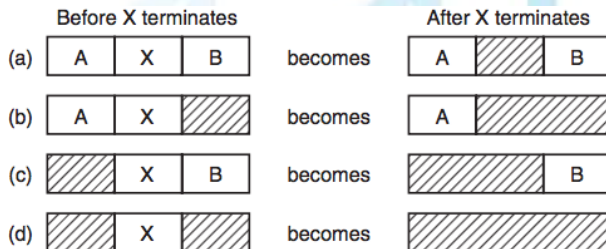


Figure: Four neighbour combinations for the terminating process, X. (Source: (Tanenbaum and Bos, 2015))

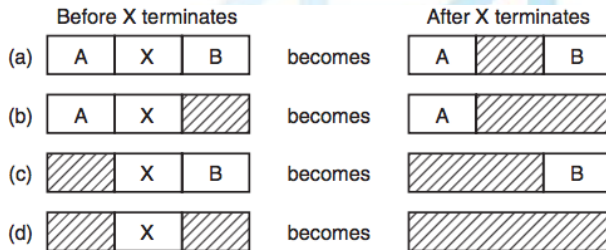


Figure: Four neighbour combinations for the terminating process, X. (Source: (Tanenbaum and Bos, 2015))

- Figure (a) updating the list requires replacing a P by an H;
- Figure (b) and (c) two entries are merged into one;
- Figure (d) three entries are merged;

How can we allocate memory based on the list approach? Any ideas?

When processes and holes are kept on a list **sorted** by address:

- Several algorithms can be used to allocate memory for:
  - Creating a process;
  - Swapping an existing process from the disk;
- Examples of algorithms:
  - **First fit**
  - **Next fit**
  - **Best fit**

Lets have a look at these...



# First fit

- Memory manager scans list until it finds a hole that is big enough;
- Hole is then broken up into two pieces:
  - One for the process;
  - One for the unused memory;
  - Except in the statistically unlikely case of an exact fit

# Next fit

- Works the same way as first fit except that:
  - Keeps track of where it is whenever it finds a suitable hole;
  - Next time it is called to find a hole:
    - Starts search from the place where it left off last time...
    - ....instead of always at the beginning, as first fit does
- Simulations show that next fit gives **slightly worse** performance than first fit.

# Best fit (1/2)

- Best fit searches the entire list, from beginning to end:
  - For the smallest hole that is adequate;
- Rather than breaking up a big hole that might be needed later:
  - tries to find a hole that is close to the actual size needed...
  - ...to best match the request and the available holes.
- Slower than **first fit**: entire list must be searched every time;

# Best fit (2/2)

- **Surprisingly:**
  - Results in more wasted memory than first fit or next fit:

Why do you think this happens? Any ideas?

## Best fit (2/2)

- **Surprisingly:**

- Results in more wasted memory than first fit or next fit:

Why do you think this happens? Any ideas?

- Because it tends to fill up memory with **tiny, useless holes**;
- First fit generates larger holes on the average.

# Virtual Memory

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

# Virtual Memory

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

OS always loads all the memory of a process...

# Virtual Memory

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

OS always loads all the memory of a process...

Does the OS always need to load the entire process into memory? Any ideas?



# Virtual Memory

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

OS always loads all the memory of a process...

Does the OS always need to load the entire process into memory? Any ideas?

Only load those pages that are required at a single moment:

- Due to the Space-time locality principle =>
- Concept of **virtual memory**

# Virtual Memory

**Virtual memory** basic idea:

- Each program's address space is broken into chunks:
  - Called **pages**;
- Each **page** is a **contiguous** range of addresses:
  - Pages are mapped onto physical memory;
  - Not all pages have to be in physical memory at the same time;
  - This is due to the space-time locality principle;

When the program references an address that:

- Is **in** physical memory:
  - MMU maps virtual address to the corresponding physical address;
- Is **not in** physical memory:
  - OS is alerted to get missing piece:
    - This is known as a **page fault**;
  - OS re-executes the failed instruction;

# Paging

Addresses that are specific to a program are called **virtual addresses**

- Addresses only make sense in the context of a specific program;
- Addresses still need to be mapped to a physical address:
  - Physical address can be generated using the **base register**;
  - *E.g.:* Virtual address 28 can be mapped to physical address 666h

When virtual memory is used:

- Virtual addresses do not go directly to the memory bus;
- Instead they go to a **Memory Management Unit**
  - **MMU** maps virtual addresses onto physical addresses;

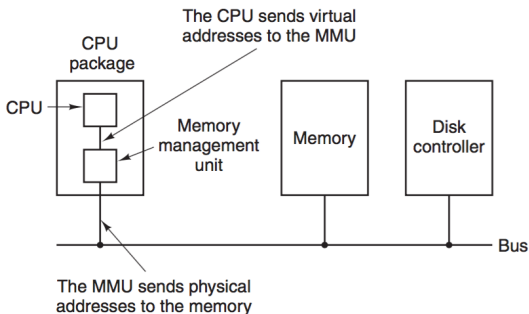
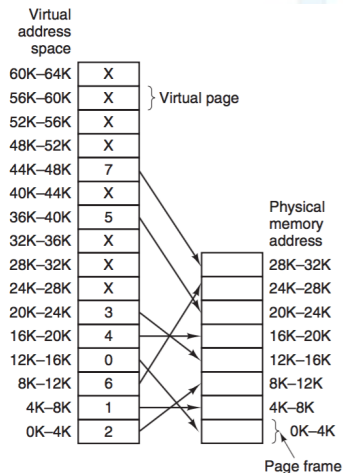


Figure: The position and function of the MMU. Here the MMU is shown as being a part of the CPU chip because it commonly is nowadays. However, logically it could be a separate chip and was years ago. (Source: (Tanenbaum and Bos, 2015))

# Example (1/10)

Consider the following address spaces (virtual + physical):



To what physical address is virtual address 0 mapped to? Any ideas?

To what physical address is virtual address 8192 mapped to? Any ideas?

To what physical address is virtual address 20500 mapped to? Any ideas?

Figure: (Source: (Tanenbaum and Bos, 2015))

## Example (2/10)

- Computer generates 16-bit addresses: from 0 to  $64K - 1$ :
  - These are the **virtual addresses**;
- Computer however only has 32 KB of physical memory;
- 64-KB programs can be written but cannot be loaded entirely into memory;
  - Program's image must be present on the disk so that pieces can be loaded;

## Example (3/10)

- Virtual address space consists of fixed-size units called **pages**:
- Corresponding units in the physical memory are called **page frames**.
  - Pages and page frames are usually the same size;
  - In this example: pages are 4KB:
    - 64 KB of virtual memory implies 16 virtual pages;
    - 32 KB of physical memory implies 8 page frames;
- Transfers between RAM and disk are always in whole pages;



## Example (4/10)

To what physical address is virtual address 0 mapped to? Any ideas?

## Example (4/10)

To what physical address is virtual address 0 mapped to? Any ideas?

- When program accesses virtual address 0:
  - Virtual address is sent to MMU;
  - MMU maps:
    - **page 0** to **page frame 2**;
    - **virtual address 0** to **physical address**  $(0 - 0) + (2 \times 4\text{KB}) = 8192$
  - MMU produces output address 8192 onto the bus;

## Example (5/10)

To what physical address is virtual address 8192 mapped to? Any ideas?

## Example (5/10)

To what physical address is virtual address 8192 mapped to? Any ideas?

- When program accesses virtual address 8192:
  - Virtual address is sent to MMU;
  - MMU maps:
    - **page 2** to **page frame 6**;
    - **virtual address 8192** to **physical address**  $(8192 - 8192) + 6 \times 4\text{KB} = 24576$
  - MMU produces output address 24576 onto the bus;

## Example (6/10)

To what physical address is virtual address 20500 mapped to? Any ideas?

## Example (6/10)

To what physical address is virtual address 20500 mapped to? Any ideas?

- When program accesses virtual address 20500:
  - Virtual address is sent to MMU;
  - MMU maps:
    - **page 5** to **page frame 3**;
    - **virtual address 20500** to **physical address**  $(20500 - 20K) + (3 \times 4KB) = 12308$ ;
  - MMU produces output address 12308 onto the bus;

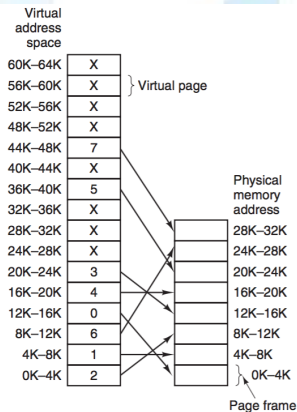
## Example (7/10)

What happens if the program references an unmapped address?

## Example (8/10)

What happens if the program references an unmapped address?

**E.g.:** CPU tries to access address 32780 from the previous example (1/2):





## Example (9/10)

What happens if the program references an unmapped address?

**E.g.:** CPU tries to access address 32780 from the previous example (1/2):

- This is byte 12 within virtual page 8;
- MMU issues a **page fault**;

What can the OS do with this page fault? Any ideas?

## Example (9/10)

What happens if the program references an unmapped address?

**E.g.:** CPU tries to access address 32780 from the previous example (1/2):

- This is byte 12 within virtual page 8;
- MMU: issues a **page fault**;

What can the OS do with this page fault? Any ideas?

- 1 Picks a little-used **page frame**;
- 2 Writes its contents back to the disk;
- 3 Fetches from disk the page that was referenced;
- 4 Changes the map;
- 5 Restarts the trapped instruction;

## Example (10/10)

**E.g.:** CPU tries to access address 32780 from the previous example (2/2):

- *E.g.* OS decides to:
  - Evict page frame 1;
  - Load virtual page 8 at physical address 4096;
  - Makes two changes to the MMU:
    - Marks virtual page 1 as unmarked;
    - Marks virtual page 8 with a 1;

# Memory Management Unit

Now that we have seen virtual memory:

How do you think the MMU works? Any ideas?

Don't forget that the MMU objective is to:

- Map virtual address spaces into physical addresses;

# Memory Management Unit

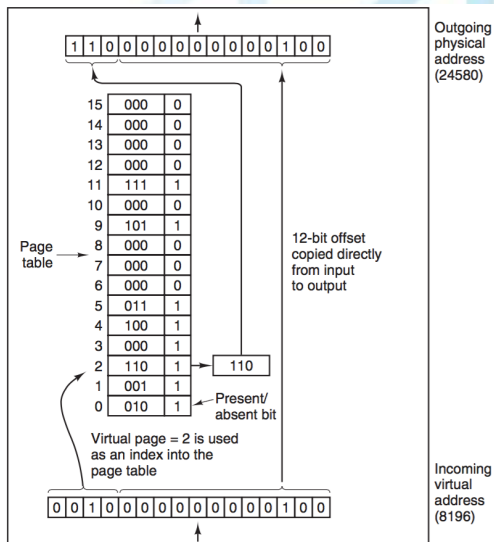


Figure: The internal operation of the MMU with 16 4-KB pages. (Source: (Tanenbaum and Bos, 2015))

From the previous figure (1/2):

- Virtual address  $8196_{10} = 0010000000000100_2$  needs to be mapped to;
- The 16-bit virtual address is split into:
  - 4-bit page number:
    - So that 16 pages of virtual memory can be addressed;
  - 12-bit offset:
    - So that we can address all of the 4096 bytes within a page;

From the previous figure (2/2):

- Page number is used as an index into the page table:
  - Yielding corresponding page frame;
  - If the Present / Absent bit is 0:
    - Trap to OS is caused (**page fault**);
  - If the Present / Absent bit is 1:
    - Page frame number is copied to the high-order 3 bits of the output register;
    - Along with the 12-bit offset which is copied unmodified;
    - Together they form a 15-bit physical address;
    - Output register is then put onto the memory bus;

# Page Tables

Virtual address mapping onto physical addresses can be summarized as:

- Virtual address is split into:
  - Virtual page number (high-order bits);
  - Offset (low-order bits);
- *E.g.*: 16-bit address, 4-KB page size:
  - Upper 4 bits specify one of the 16 virtual pages;
  - Lower 12 bits specify the byte offset;



Virtual page number is used as an index into the page table:

- In order to find the virtual page entry:
  - Allowing to find the page frame number (if any);
- Page frame number is attached to the high-order end of the offset:
  - Replacing virtual page number:
    - Forming a physical address that can be sent to the memory.

# Structure of a Page Table Entry

Ok, now that we have a better understanding of page tables:

What are the details of a single page table entry? Any ideas?

# Structure of a Page Table Entry

Ok, now that we have a better understanding of page tables:

What are the details of a single page table entry? Any ideas?

- Exact layout of an entry in the page table is highly machine dependent;
- **However:** information is roughly the same from machine to machine.

What information do you think should be part of a page table entry?

What information do you think should be part of a page table entry?

Several items are needed (1/3):

- **Page frame number:** most important information:
  - After all, the goal is to map a page into a frame ;)
- **Present / absent bit:**
  - **Value 1:**
    - Valid entry and frame can be used;
  - **Value 0:**
    - Page fault!
    - Page needs to be loaded into memory;

What information do you think should be part of a page table entry?

Several items are needed (2/3):

- **Protection** bits determine access type:
  - read? write? execute?
- **Modified** bit:
  - Set to 1 when page is modified;
- **Referenced** bit:
  - Set to 1 whenever a page is referenced (read or write);
  - Helps OS decide which page to evict when a page fault occurs:
    - Choose pages that are not being used;

What information do you think should be part of a page table entry?

Several items are needed (3/3):

- **Caching** bit:
  - Allows caching to be disabled for the page;
  - Important for pages that map onto devices registers rather than memory;

Sample page table entry:

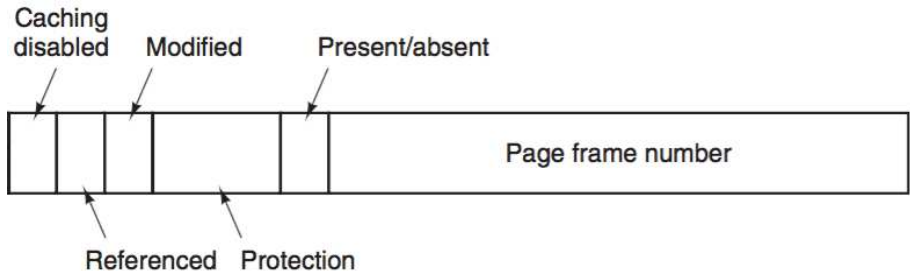


Figure: A typical page table entry. (Source: (Tanenbaum and Bos, 2015))

- Size varies: but 32 bits is a common size;



# Speeding Up Paging

In any paging system, two major issues must be faced:

- 1 Mapping from virtual address to physical address must be fast:
  - Virtual-to-physical mapping must be done on every memory reference;
  - Several page table references per instruction:
    - Fetch instruction;
    - Fetch operands;
    - Store results;

# Speeding Up Paging

In any paging system, two major issues must be faced:

2 If the virtual address space is large, the page table will be large:

- Virtual address space of 32 bits and 4-KB page size:
  - $2^{32}/2^{12} = 2^{20}$  pages;
- Virtual address space of 64 bits and 4-KB page size:
  - $2^{64}/2^{12} = 2^{42}$  pages;

Lets focus on the first point:

- Mapping from virtual address to physical address must be fast:

How can we have fast page mapping? Any ideas?

Lets focus on the first point:

How can we have fast page mapping? Any ideas?

Simplest design is to:

- Have a page table consisting of an array of fast hardware registers;
  - With one entry for each virtual page, indexed by virtual page number;
- **When a process starts:**
  - OS loads registers with the process' page table;

Can you see any **advantages** with the previous scheme? Any ideas?

Can you see any **advantages** with the previous scheme? Any ideas?

- Page table entries are filled out when process is started:
  - No more memory accesses are necessary in order to access the page table;
- Very simple to implement;

Can you see any **disadvantages** with the previous scheme? Any ideas?

Can you see any **disadvantages** with the previous scheme? Any ideas?

- Expensive if the page table is large;
- Having to load the full page at every context switch would kill performance;



So what can we do to solve these issues? Any ideas?

So what can we do to solve these issues? Any ideas?

With a little help of our friend:

- **Translation Lookaside Buffer;**

# Translation Lookaside Buffer

First things first:

Do you have any idea of what a Translation Lookaside Buffer is? Any ideas?

# Translation Lookaside Buffer

First things first:

Do you have any idea of what a Translation Lookaside Buffer is? Any ideas?

- **HINT:** Are there program parts that execute more than others?

# Translation Lookaside Buffer

First things first:

Do you have any idea of what a Translation Lookaside Buffer is? Any ideas?

- **HINT:** Are there program parts that execute more than others?
- **HINT:** Space / Locality principle:
  - If one instruction is accessed:
    - Highly probable that neighbour instructions will be accessed;
    - Highly probable that the instruction will be accessed again;

# Translation Lookaside Buffer

What can we do to take advantage of the space / locality principle?  
Any ideas?

# Translation Lookaside Buffer

What can we do to take advantage of the space / locality principle?  
Any ideas?

- Cache systems;
- The **Translation Lookaside Buffer** is a memory cache;

# Translation Lookaside Buffer

Translation lookaside buffer (**TLB**) (1/2):

- Cache used to reduce the time to map virtual to physical addresses;
- Part of the chip's memory-management unit (**MMU**);
- Stores the recent translations of virtual memory to physical memory;



# Translation Lookaside Buffer

Translation lookaside buffer (**TLB**) (2/2):

- Consists of a small number of entries: rarely more than 256
- Each entry contains information about one page, including:
  - virtual page number;
  - modified bit;
  - protection bits (read/write/execute permissions);
  - physical page frame;
  - valid bit: indicates whether the entry is valid (*i.e.*, in use) or not.

# Example

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Figure: A TLB to speed up paging. (Source: (Tanenbaum and Bos, 2015))

**Example** of a process that might generate previous TLB:

- Loop that spans virtual pages 19, 20, and 21:
  - These TLB entries have protection codes for reading and executing;
- Main data currently being used:
  - *E.g.*: an array being processed;
  - Pages 129 and 130;
  - These TLB entries have protection codes for reading and writing;
- Page 140 contains the indices used in the array calculations;
- Stack is on pages 860 and 861.

With the addition of the TLB:

What is the process for converting a virtual address into a physical one?  
Any ideas?

When a virtual address is presented to the MMU for translation (1/2):

- 1 Hardware checks if virtual page number is present in the TLB;
  - This is done by comparing it to all the entries simultaneously (*i.e.*, in parallel)
  - Doing so requires special hardware, which all MMUs with TLBs have;
- 2 If match is found (**TLB hit**) and protection bits are **not violated**:
  - Page frame is taken directly from the TLB, without going to the page table

When a virtual address is presented to the MMU for translation (2/2):

- 3 If match is found (**TLB hit**) and protection bits are **violated**:
  - Protection fault is generated;
- 4 If match is not found (**TLB miss**) then the MMU:
  - Does an ordinary page table lookup:
  - Evicts one TLB entry and replaces it with the page table entry just looked up:
    - TLB's entry modified bit is copied back into the page table entry;

# Page Tables for Large Memories

Lets focus on the second point:

- If the virtual address space is large, the page table will be large;
  - Virtual address space of 32 bits and 4-KB page size:
    - $2^{32}/2^{12} = 2^{20}$  pages;
  - Virtual address space of 64 bits and 4-KB page size:
    - $2^{64}/2^{12} = 2^{42}$  pages;

How to deal with very large virtual address spaces? Any ideas?

# Multilevel Page Tables

How to deal with very large virtual address spaces? Any ideas?

**Idea:** Avoid keeping all the page tables in memory all the time

- Pages that are not needed should not be kept around;
- This is the idea behind multilevel page tables;

Lets look at this through an example =>



## Example (1/7)

*E.g.:* process that needs 12 MB:

- Bottom 4 MB of memory for program text;
- Next 4 MB for data;
- Top 4 MB for the stack.
- Between data and stack: **hole** that is not used.

## Example (2/7)

Consider the following **multilevel page table** for a 32-bit virtual address:

- 10-bit **PT1** field;
- 10-bit **PT2** field;
- 12-bit **offset** field;

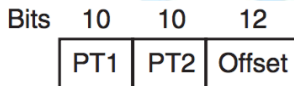
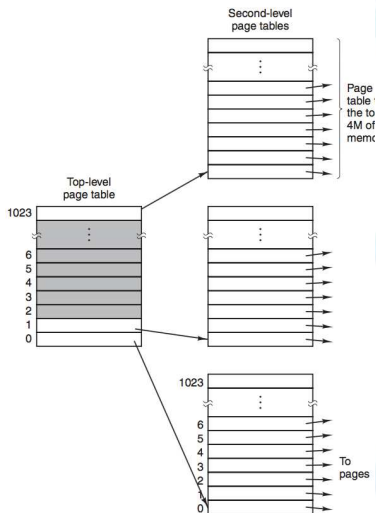


Figure: A 32-bit address with two page table fields. (Source: (Tanenbaum and Bos, 2015))

# Example (3/7)



Top-level page entries:

- Entry 0: program text;
- Entry 1: data;
- Entry 1023: stack;
- Entries 2-1022 (shaded): not used;

Figure: Two-level page tables. (Source: (Tanenbaum and Bos, 2015))

## Example (4/7)

Top-level page (left-most table) with 1024 entries (10-bit PT1 field):

- Each entry represents 4M of addresses ( $2^{32}/2^{10} = 2^{22} = 4M$ );

When a virtual address is presented to the MMU:

- PT1 field is extracted and used to index top-level page:
  - Producing frame number of a second-level **page table**;
  - **Important:** this is frame is also a page table;

## Example (5/7)

Consider 32-bit virtual address 0x00403004 with decomposition:

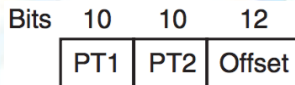


Figure: A 32-bit address with two page table fields. (Source: (Tanenbaum and Bos, 2015))

How can we obtain the multilevel information from this address? Any ideas?

## Example (5/7)

Consider 32-bit virtual address 0x00403004

- $00403004_{(16)} = \underbrace{0000000001}_{\text{PT1 Field}} \underbrace{0000000011}_{\text{PT2 Field}} \underbrace{000000000100}_{\text{Offset}}_{(2)}$
- PT1 field = 1;
- PT2 field = 3;
- Offset field = 4;

## Example (6/7)

1 MMU uses PT1 to index top-level page table:

- Entry 0:  $[0, 4M - 1]$ ;
- Entry 1:  $[4M, 8M - 1]$ ;
- Entry 2:  $[8M, 12M - 1]$ ;
- Entry 3:  $[12M, 16M - 1]$ ;
- ...
- Entry 1023:  $[1023 * 4M, 1023 * 4M + 4M - 1]$ ;

## Example (6/7)

- 1 MMU uses PT1 to index top-level page table:
- Entry 0:  $[0, 4M - 1]$ ;
  - **Entry 1:  $[4M, 8M - 1]$  (PT1 field = 1);**
  - Entry 2:  $[8M, 12M - 1]$ ;
  - Entry 3:  $[12M, 16M - 1]$ ;
  - ...
  - Entry 1023:  $[1023 * 4M, 1023 * 4M + 4M - 1]$ ;



## Example (6/7)

② MMU uses PT2 to index second-level page table:

- Each entry maps to a 4KB page frame ( $2^{22}/2^{10}$ ):
  - Entry 0:  $[0, 4K - 1]$ ;
  - Entry 1:  $[4K, 8K - 1]$ ;
  - Entry 2:  $[8K, 12K - 1]$ ;
  - Entry 3:  $[12K, 16K - 1]$ ;
  - ...
  - Entry 1023:  $[1023 * 4K, 1023 * 4K + 4K - 1]$ ;

## Example (6/7)

② MMU uses PT2 to index second-level page table:

- Each entry maps to a 4KB page frame ( $2^{22}/2^{10}$ ):
  - Entry 0:  $[0, 4K - 1]$ ;
  - Entry 1:  $[4K, 8K - 1]$ ;
  - Entry 2:  $[8K, 12K - 1]$ ;
  - **Entry 3:  $[12K, 16K - 1]$  (PT2 field = 3);**
  - ...
  - Entry 1023:  $[1023 * 4K, 1023 * 4K + 4K - 1]$ ;

## Example (7/7)

This means that virtual address 0x00403004:

- PT1 field = 1;
- PT2 field = 3;
- Offset field = 4;

Maps to the following physical address:

$$\begin{aligned} & PT1 \times 4M + PT2 \times 4K + offset \\ &= 1 \times 4M + 3 \times 4K + 4 \\ &= 4 \times 2^{20} + 3 \times 4 \times 2^{10} + 4 \\ &= 4206596 \end{aligned}$$

Seems confusing?

- It is indeed a little bit confusing;
- Requires practice!

But is it useful to continue adding more levels? Any ideas?

But is it useful to continue adding more levels? Any ideas?

- Additional levels give more flexibility;
- Depends on the processor being used;
- Usually:
  - After a certain level the entropy only increases;
  - *i.e.* no information gains are achieved;

So what can be done besides increasing the number of levels? Any ideas?

So what can be done besides increasing the number of levels? Any ideas?

- Inverted page tables;



# Inverted Page Tables

Alternative to ever-increasing levels:

- One table entry per **page frame** in real memory;
- Rather than one entry per page of virtual address space;
- Entry tracks which (process, virtual page) is located in the page frame;

# Example

Consider a 64-bit virtual addresses, with 4-KB page size and 4-GB of RAM:

- Only  $\frac{2^{32}}{2^{12}} = 2^{20}$  inverted page table **entries** are required;
- Instead of having to have  $\frac{2^{64}}{2^{12}} = 2^{52}$  **entries**;
- If each entry requires 4 bytes (32 bits) difference between storing
  - $2^{20} \times 32 = 32 \text{ MB}$ ;
  - $2^{52} \times 32 = 128 \text{ PB}$ ;

Guess which one is better in terms of space saved? ;)

Can you see any problems with inverted page tables? Any ideas?

Can you see any problems with inverted page tables? Any ideas?

Virtual-to-physical translation becomes much harder;

- Entry keeps track of which (process, virtual page) is located in the frame;
- Hardware can no longer index the table using the virtual page;
- Instead it must search the entire inverted page table for the entry:
  - This must be done on every memory reference...;
  - If table has  $N$  elements then  $O(N)$  time...

What can be done to avoid having to search the inverted page table?  
Any ideas?

What can be done to lessen the effects of having to search the inverted page table? Any ideas?

Make use of TLB:

- **TLB Hit:** translation happens very fast;
- **TLB Miss:** table has to be searched;

However, a **TLB miss** still implies having to search...

What can be done to lessen the effects of having to search the inverted page table? Any ideas?

However, a **TLB miss** still implies having to search...

What can be done to lessen the effects of having to search the inverted page table? Any ideas?

Use a hash table hashing on the virtual address =>



Use a hash table hashing on the virtual address:

- Virtual pages with the same hash value are chained together;
- If the hash table has as many slots as the machine has physical pages:
  - Average chain will be only one entry long;
  - Greatly speeding up the mapping;

Once the page frame number has been found:

- Tuple (virtual, physical) is entered into the TLB;

In conclusion:

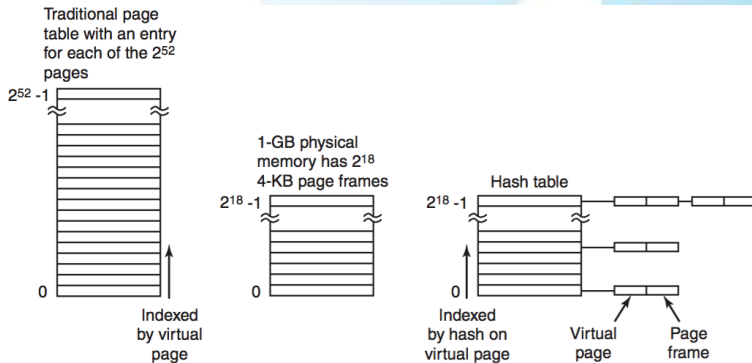


Figure: Comparison of a traditional page table with an inverted page table. (Source: (Tanenbaum and Bos, 2015))

# Page Replacement Algorithms

Lets have a look at another topic:

Why do we need page replacement algorithms? Any ideas?

When a page fault occurs:

- OS has to choose a page to evict to make room for the incoming page;
- If the page to be removed has been **modified** while in memory:
  - Page must be rewritten to disk;
- If, however, the page has not been changed:
  - disk copy is already up to date: no rewrite is needed;
  - page to be read overwrites page being evicted.

What would be good ideas for choosing a page for replacement? Any ideas?

What would be good ideas for choosing a page for replacement? Any ideas?

- **Idea:** Choose a page that is not used a lot ;)
  - Whether from the process causing the fault;
  - Or from some other process...

There are several page replacement algorithms:

- Optimal Page Replacement Algorithm;
- Not Recently Used Page Replacement Algorithm;
- First-In, First-Out (FIFO) Page Replacement Algorithm;
- Second-Chance Page Replacement Algorithm;
- Clock Page Replacement Algorithm;
- Least Recently Used (LRU) Page Replacement Algorithm
- Working Set (WS) Page Replacement Algorithm
- WSClock Page Replacement Algorithm

Lets have a look at these algorithms:

- All of which you should know for the exam =)

# Optimal Page Replacement Algorithm

Lets start by considering an example with two pages:

- One will not be used for 8 million instructions;
- Another will not be used for 6 million instructions

When a **page fault** occurs:

Which would you remove first? Any ideas?



# Optimal Page Replacement Algorithm

Lets start by considering an example with two pages:

- One will not be used for 8 million instructions;
- Another will not be used for 6 million instructions

When a **page fault** occurs:

Which would you remove first? Any ideas?

Remove page that will need longer to be used (first one):

- Computers, like people, try to put off unpleasant events for as long as they can ;)

# Optimal Page Replacement Algorithm

When a **page fault** occurs:

- Page containing the instruction will be referenced next;
- Other pages may not be referenced until 10, 100 or 1000 instructions later:

For each page:

- Remember number of instructions that need to be executed:
  - **Before** page is first referenced.
- Remove page that has highest label:
  - */i.e.:* Page that will be referenced later when compared with other pages;

Can you see any problem with the previous algorithm? Any ideas?

Can you see any problem with the previous algorithm? Any ideas?

Algorithm is **unrealizable**:

- No way of knowing when each of the pages will be referenced next;

# Not Recently Used Page Replacement Algorithm

Use two bits **R** (referenced) and **M** (modified) for each page table entry:

- When a process is started up:
  - Both page bits for all its pages are set to 0 by the OS;
- Periodically the R bit is cleared:
  - Distinguish pages that have not been referenced recently;

On a page fault OS inspects pages and divides them into four categories

- Class 0 - not referenced, not modified ( $\bar{R}\bar{M}$ )
- Class 1 - not referenced, modified ( $\bar{R}M$ )
- Class 2 - referenced, not modified ( $R\bar{M}$ )
- Class 3 - referenced, modified ( $RM$ )

**NRU** (Not Recently Used) algorithm removes:

- Random page from the lowest-numbered nonempty class;
- **Idea:**
  - Better to remove a modified page that has not been referenced lately...
  - ...than a clean page that is in heavy use.

NRU algorithm is:

- Easy to understand;
- Performance: not optimal but may be adequate.



# First-In, First-Out (FIFO) Page Replacement Algorithm

OS maintains a list of all pages currently in memory:

- Most recent arrival at the tail;
- Least recent arrival at the head

On a **page fault**:

- Page at the head is removed;
- New page added to the tail of the list;

Can you see any problem with the FIFO approach? Any ideas?

Can you see any problem with the FIFO approach? Any ideas?

- Oldest page may still be useful...
- For this reason, FIFO in its pure form is **rarely** used.

# Second-Chance Page Replacement Algorithm

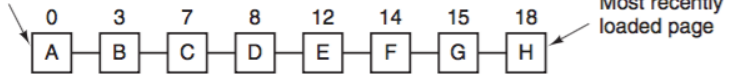
Can you think of a method to improve the FIFO approach? Any ideas?

Can you think of a method to improve the FIFO approach? Any ideas?

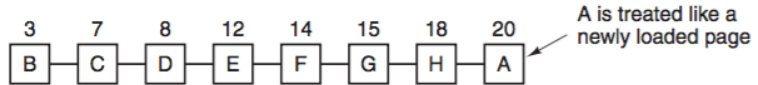
Modify FIFO to avoid throwing out a heavily used page:

- Inspect the R bit of the oldest page:
  - If it is  $R = 0$ :
    - Page is both old and unused: replace immediately
  - If the  $R = 1$ :
    - Bit is cleared and page is put onto the end of the list of pages;
    - Search continues;

Page loaded first



(a)



(b)

Figure: Operation of second chance. (a) Pages sorted in FIFO order. (b) Page list if a page fault occurs at time 20 and A has its R bit set. The numbers above the pages are their load times. (Source: (Tanenbaum and Bos, 2015))

- **Load times:** time pages arrived in memory;

From the previous figure:

- Suppose that a page fault occurs at time 20:
  - Oldest page is A, which arrived at time 0, when the process started
  - If A has  $R=0$ : evict from memory;
  - If the  $R = 1$ :
    - A is put onto the end of the list;
    - A's "load time" is reset to the current time (20);
    - A's R bit is also cleared
    - Search for a suitable page continues with B.

But what happens if all pages have  $R = 1$ ? Any ideas?



But what happens if all pages have  $R = 1$ ? Any ideas?

If all the pages have been referenced:

- One by one OS moves the pages to the end of the list:
  - clearing the respective  $R$  bits;
- Eventually all pages will have  $R$  cleared ( $R=0$ ):
  - And the list will be in the exact same initial order;
  - Degenerating into pure FIFO

# Clock Page Replacement Algorithm

Can you see any problems with the 2<sup>nd</sup> chance algorithm? Any ideas?

Can you see any problems with the 2<sup>nd</sup> chance algorithm? Any ideas?

Constantly moving pages around the list is inefficient...

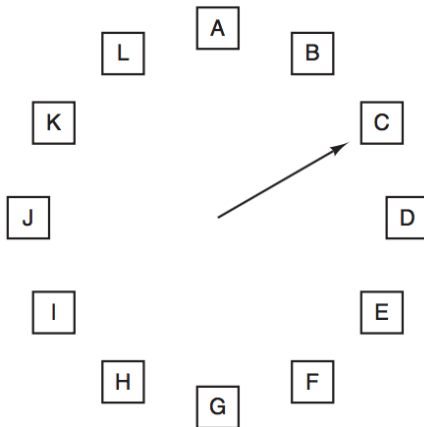
What can be done to avoid always moving pages? Any ideas?

Can you see any problems with the 2<sup>nd</sup> chance algorithm? Any ideas?

Constantly moving pages around the list is inefficient...

What can be done to avoid always moving pages? Any ideas?

Better approach: keep all page frames on a circular list (clock)



When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:

R = 0: Evict the page

R = 1: Clear R and advance hand

Figure: The clock page replacement algorithm. (Source: (Tanenbaum and Bos, 2015))

From the previous figure:

- On a page fault: page being pointed to by the hand is inspected:
  - If  $R=0$ :
    - Page is evicted, the new page is inserted into the clock in its place
    - Hand is advanced one position
  - If  $R=1$ :
    - $R$  is cleared ( $R=0$ );
    - Hand is advanced to the next page
  - Process is repeated until a page is found with  $R = 0$ .

# Least Recently Used (LRU) Page Replacement Algorithm

## Observation:

- Pages **heavily used** will probably be heavily used again;
- Pages that have **not been heavily used** will probably remain unused;

## Idea:

- On a page fault:
  - Throw out the page that has been unused for the longest time
- Strategy is called **LRU** (Least Recently Used) paging.

Can you see any problems with the LRU approach? Any ideas?



Can you see any problems with the LRU approach? Any ideas?

Not cheap to implement:

- Necessary to maintain a linked list of all pages in memory:
  - Most recently used page at the front;
  - Least recently used page at the rear;
- List must be **updated** on every memory reference:
  - Main difference to the FIFO method;
- Time consuming operations:
  - Finding a page in the list;
  - Deleting a page in the list;
  - Moving a page to the front of the list;

# Working Set Page Replacement Algorithm

## Idea:

- Keep track of each process working set (**WS**) of pages:
  - Obviously over time this set will change...
- Make sure that set is in memory before letting the process run.

WS algorithm:

- OS keeps track of pages in the WS;
- On a page fault:
  - Find a page not in the process WS and evict it;
- WS can have several definitions:
  - **Definition 1:**  
Set of pages used in the  $k$  last memory references
  - **Definition 2:**  
Set of pages used during the past  $X$  msec of execution time  
(**Preferred Definition**)

Basic idea:

- Find a page that **is not** in the WS and evict it;
- Each entry contains at least two items:
  - Time the page was last used;
  - R (Referenced) bit.

# Example

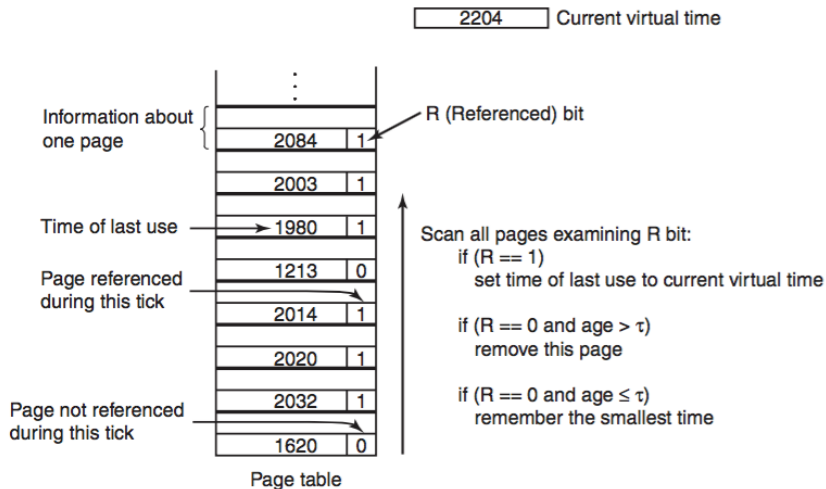


Figure: The WS algorithm. (Source: (Tanenbaum and Bos, 2015))

The algorithm works as follows (1/2):

- Hardware is assumed to set the R and M bits
- Clock interrupt clears the R bit periodically;
- On every **page fault**:
  - Page table is scanned to look for a suitable page to evict;
  - Process each entry and examine R bit:

The algorithm works as follows (2/2):

- As each entry is processed, the R bit is examined:
  - If  $R = 1$ :
    - Set time of last use to current time...
    - ...Indicating page was in use at the time the fault occurred.
    - ...thus the page is in WS and is not suitable for removal;
  - If  $R = 0$ :
    - Page not referenced recently and may be a candidate for removal;
    - Pages that are older than threshold  $\tau$  can be removed;

# WSClock Page Replacement Algorithm

Can you see any problems with the WS approach?



Can you see any problems with the WS approach?

Requires scanning entire page table at each page fault:

- Until a suitable candidate is located...

Can you see any problems with the WS approach?

Entire page table has to be scanned at each page fault until a suitable candidate is located...

Can you think of a better method? Any ideas?

Can you see any problems with the WS approach?

Entire page table has to be scanned at each page fault until a suitable candidate is located...

Can you think of a better method? Any ideas?

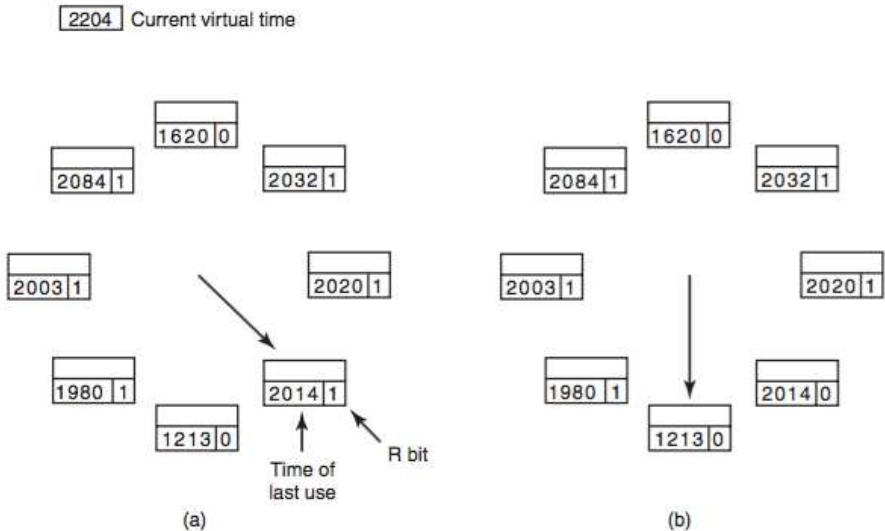
Use a a circular list of page frames (clock)

Data structure needed is a circular list of page frames:

- Initially: list is empty:
  - When the first page is loaded, it is added to the list.
  - As more pages are added: ring is formed;
- Each list entry contains:
  - Time of last use field;
  - R and M bits;

As with the clock algorithm (1/2):

- 1 Each **page fault**: page pointed to by the hand is examined first;
- 2 If  $R = 1$ : not ideal candidate to remove since it was used recently;
  - R bit is then set to 0: hand advances to next page;
  - Search continues until an  $R = 0$  is found;



(a)

(b)

Figure: Operation of the WSClock algorithm. (a) and (b) give an example of what happens when  $R = 1$ . (Source: (Tanenbaum and Bos, 2015))

As with the clock algorithm (2/2):

3 If  $R=0$

- If the age  $\geq \tau$  and the page is not modified:
  - Page is not in working set;
  - Page can be removed;
- If the age  $\geq \tau$  and the page is modified:
  - Page can be removed after being written to disk;

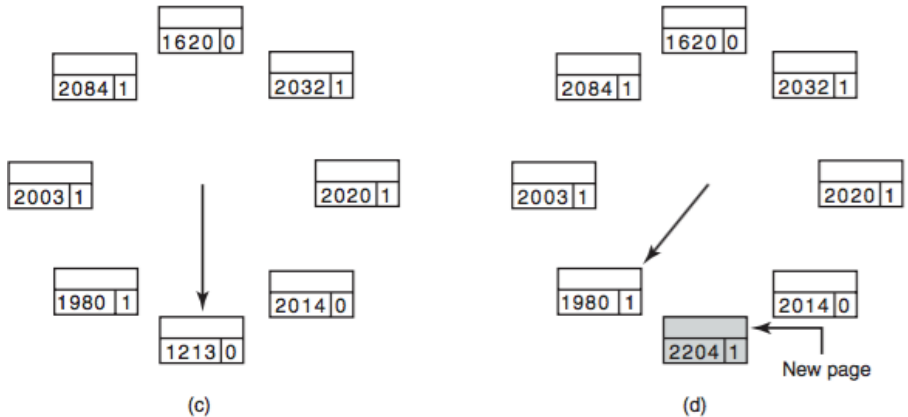


Figure: c) and (d) give an example of  $R = 0$ . Current virtual time = 2204 ( (Source: (Tanenbaum and Bos, 2015))



# Summary of Page Replacement Algorithms

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude approximation of LRU
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

Figure: Page replacement algorithms discussed in the text. (Source: (Tanenbaum and Bos, 2015))

# Design Issues For Paging Systems

The following slides discuss issues that impact the paging system:

- Local versus Global Allocation Policies
- Load Control
- Page Size
- Shared Pages
- Cleaning Policy

# Local versus Global Allocation Policies

How should memory be allocated among the competing runnable processes? Any ideas?

# Example

	Age
A0	10
A1	7
A2	5
A3	4
A4	6
A5	3
B0	9
B1	4
B2	6
B3	2
B4	5
B5	6
B6	12
C1	3
C2	5
C3	6

How memory should be allocated among the competing runnable processes? Any ideas?

Figure: Local versus global page replacement.  
(Source: (Tanenbaum and Bos, 2015))

# Example

	Age
A0	10
A1	7
A2	5
A3	4
A4	6
A5	3
B0	9
B1	4
B2	6
B3	2
B4	5
B5	6
B6	12
C1	3
C2	5
C3	6

From the figure on the left:

- Three processes - A, B, C;
- Suppose A gets a page fault;
- Which pages should be considered?
  - All pages from A, B and C?
  - Only pages from A?

Figure: Local versus global page replacement.  
(Source: (Tanenbaum and Bos, 2015))

# Example

If we employ an LRU strategy **only to A's** pages:

- then A5 is replaced by A6:

	Age
A0	10
A1	7
A2	5
A3	4
A4	6
A5	3
B0	9
B1	4
B2	6
B3	2
B4	5
B5	6
B6	12
C1	3
C2	5
C3	6

Figure: Original configuration (Source: (Tanenbaum and Bos, 2015))

A0
A1
A2
A3
A4
A6
B0
B1
B2
B3
B4
B5
B6
C1
C2
C3

Figure: Local page replacement. (Source: (Tanenbaum and Bos, 2015))

# Example

If we employ an LRU strategy to **all** pages:

- then B3 is replaced by A6:

	Age
A0	10
A1	7
A2	5
A3	4
A4	6
A5	3
B0	9
B1	4
B2	6
B3	2
B4	5
B5	6
B6	12
C1	3
C2	5
C3	6

Figure: Original configuration (Source: (Tanenbaum and Bos, 2015))

A0
A1
A2
A3
A4
A5
B0
B1
B2
A6
B4
B5
B6
C1
C2
C3

Figure: Local page replacement. (Source: (Tanenbaum and Bos, 2015))

Decision between **local** and **global** pages:

- **Local:** only considers pages from the process's address space;
- **Global:** considers pages from all address spaces;

In general: **global algorithms** work better. Why? Any ideas?



In general: **global algorithms** work better. Why? Any ideas?

If a **local algorithm** is used and **working set grows**:

- Pages will be replaced even if there are free page frames;
- Trashing will occur!

If a **local algorithm** is used and **working set shrinks**:

- Pages from original set will be replaced:
  - Instead of considering that the working set is shrinking...
  - Resulting in wasted memory!

If a **global algorithm** is used (1/2):

- OS must decide how many page frames to assign to each process:
  - Bigger processes should get higher number of page frames;
  - Smaller processes won't have a big demand for page frames;
- Start each process with a number of pages proportional to the process size;

If a **global algorithm** is used (2/2):

- If the **page fault frequency** starts **increasing**:
  - Signals to increase a process' page allocation
- If the **page fault frequency** starts **decreasing**:
  - Signals to decrease a process' page allocation

# Load Control

What happens whenever the combined working sets of all processes exceed the capacity of memory?

**Trashing** can be expected:

- System will spend a lot of time copying data from disk to memory;
- If the wrong pages are evicted from memory:
  - They will have to be loaded again very soon;
- This results in a lot of wasteful copying... A.k.a. as **trashing**

What can be done to minimize the chances of trashing? Any ideas?

What can be done to minimize the chances of trashing? Any ideas?

Possible strategy:

- Swap some of the processes to disk:
  - Freeing up all the pages they are holding;
  - Free page frames can be attributed to trashing processes;
- Continue swapping processes until trashing stops;

# Page Size

OS chooses page size:

How should the OS select a page size? Any ideas?

# Page Size

OS chooses page size:

How should the OS select a page size? Any ideas?

Determining best size requires balancing several competing factors...

- Choose a page size too big:
  - Possibility for wasted memory;
- Choose a page size too small:
  - OS needs to maintain larger page table;
  - Using valuable space in the **TLB**;
- **Conclusion:** There is no overall algorithm...



# Shared Pages

Sometimes it is useful to share pages among processes:

- *E.g.:* user may be running several programs that use the same library;
  - Efficient to share pages: avoid multiples copies of the same page;
- Problem is that not all pages are sharable:
  - Pages that are read-only, such as program text, can be shared;
  - Sharing data pages is more complicated...

When two or more processes share some code;

- Special care needs to be employed with shared pages;
- When a process is removed from memory:
  - Shared pages need to stay in memory:
    - Otherwise: other processes will start generating page faults;
  - Therefore: essential to discover shared pages still in use:
    - Searching all page tables is expensive;
    - Special data structures are needed;

# Cleaning Policy

Paging works best when:

- There is an abundant supply of free page frames;
  - Such frames can be claimed as page faults occur;

If every page frame is full and modified:

- Before a new page can be brought in:
  - Old page must first be written to disk

Periodically a **paging daemon** process is executed:

- Responsible for inspecting memory:
  - And freeing frames if too few page frames are free;
  - Pages are evicted according to some page replacement algorithm;
  - If pages have been modified: they are written to disk.

# References I



Tanenbaum, A. and Bos, H. (2015).

*Modern Operating Systems.*

Pearson Education Limited.