



Scuola Superiore Sant'Anna



Memory Management

Giuseppe Lipari



Summary

- Introduction
- Overview of hardware architectures
- Processes
- Processes and Threads
- Concurrency: mutual exclusion and synchronisation
- Deadlock and Starvation
- **Memory Management**
- Scheduling
- I/O Management
- File system
- Distributed Systems



Requirements

- Requirements for a memory management system
 - Relocation
 - Protection
 - Sharing
 - Logical organisation
 - Physical organisation



Relocation

- Many programs can be in memory at the same time
 - We do not know where in memory the program will be loaded
 - Moreover, a program can be swapped out
 - When it is swapped in, we cannot expect it to be loaded in the same place
 - How to deal with absolute addresses?
 - ✓ For example an absolute brach in a program



Protection

- Each process must be protected from the malicious or unwanted interferences from other process
 - This cannot be done at compile time
 - ✓ For example address calculation in an array
 - It is in contrast with relocation!
- We need to do a check at run-time!



Sharing

- Two or more processes should be allowed to share some portion of memory
 - To permit shared memory communication
 - To allow two process to share (part of) the code
- This is in contrast with protection!



Logical organisation

- Physical memory is a linear array of bytes
- However, a program is often organised in modules
 - Modules can be compiled separately and then linked together at the end
 - Some module should be read-only (for example the code) some other should be read/write
 - Modules can be shared among processes



Physical organisation

- To support virtual memory, that is virtually infinite, the mass storage memory is used
 - When there is no more memory space, some process is “swapped out”
 - When the process must execute again, it is “swapped in”
 - This swapping is the responsibility of the memory management system



Memory partitioning

- Before looking at modern virtual memory systems, we should look at simpler systems that are not used anymore
 - Step by step we will complicate the management system to arrive to the modern managers
- Techniques
 - Fixed partitioning
 - Dynamic partitioning
 - Simple paging
 - Virtual memory paging
 - Virtual memory segmentation



Fixed partitions

- The physical memory is divided into fixed partitions
 - Uniform partitions: they all have the same size
 - Non uniform: they can have different sizes
- A program can be loaded into a partition of equal or greater size
 - If the program is too big to be fit into a partition, it cannot be loaded. The programmer should work out a solution, for example by dividing the program into two smaller programs that communicate



Fixed and Uniform Partitions

Process memory requirements

PA

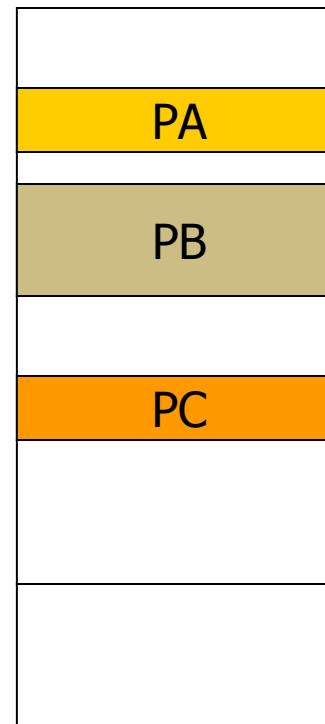
PB

PC

PD

Doesn't
fit!

Fixed and Uniform
Partitions

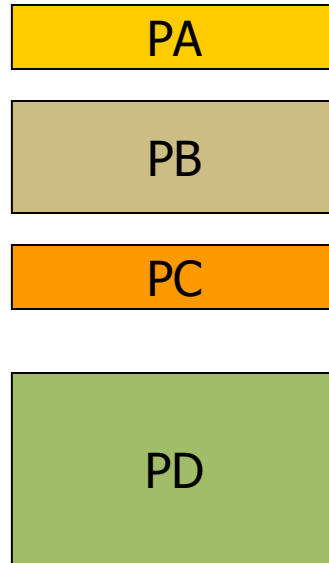


wasted
space

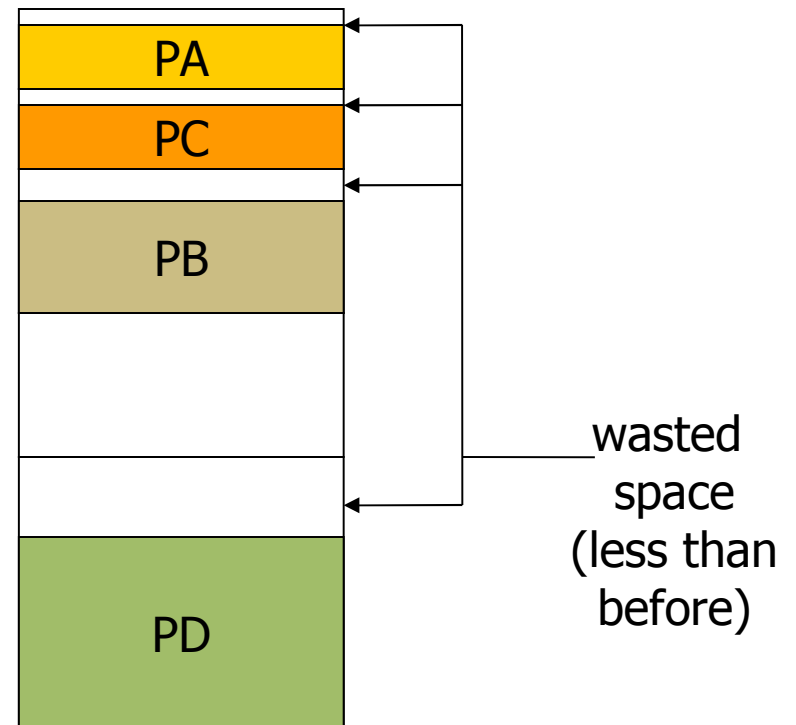


Fixed and non-uniform Partitions

Process memory requirements



Non-Uniform Partitions





Placement algorithm

- Uniform partitions:
 - trivial: every program can be loaded in any partition, it does not matter
 - If all partitions are full, one process is swapped out
- Non-uniform partitions:
 - First algorithm:
 - ✓ Assign each process to the smallest partition that can hold it
 - ✓ If two process can fit in the same partitions, one is swapped out
 - ✓ One queue for every partition
 - Second algorithm:
 - ✓ Assign each process to the smallest free partition that can hold it
 - ✓ If there are no more partitions, swap out the process in the smallest partition that can hold the desired process



Problems with fixed partitions

- Uniform partitions
 - Problems: we may waste a lot of space (small programs will occupy one partition). This problem is called *internal fragmentation*
- Non uniform partitions
 - First algorithm: reduces internal fragmentation, but a lot of swapping in and out. Some part of memory may remain unused
 - Second algorithm: good use of memory, but internal fragmentation



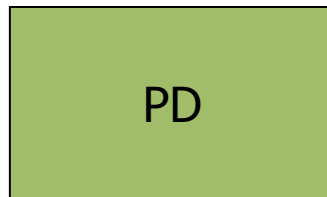
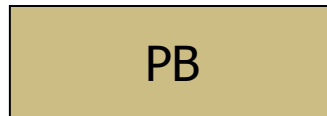
Dynamic partitions

- To overcome the internal fragmentation memory, we can compute the size of each partition dynamically
 - At the system start-up there is only one single partition
 - When a process is loaded, if there is a free partition big enough
 - ✓ The free partition is splitted in two
 - ✓ The first one has exactly the size of the process
 - ✓ The second one has the remaining size

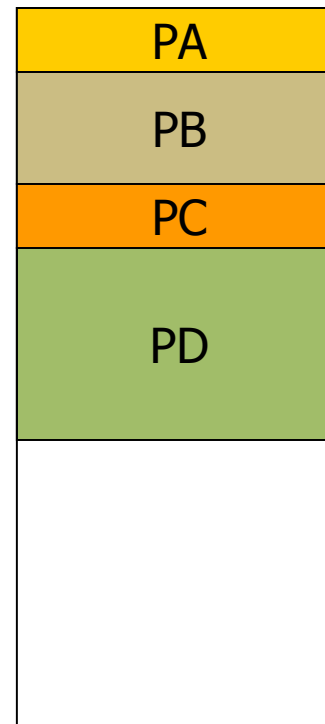


Dynamic Partitions

Process memory
requirements



Dynamic Partitions



No internal
fragmentation!



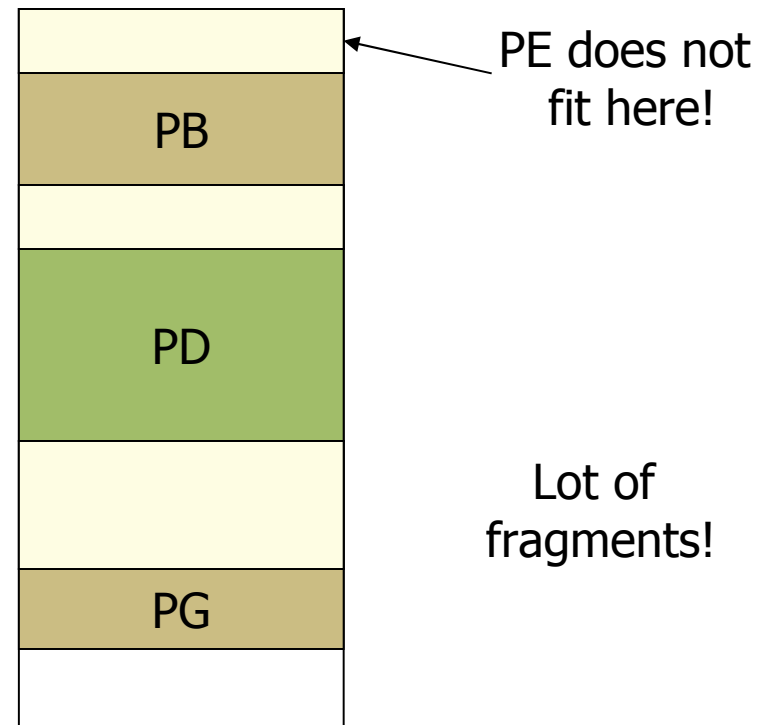
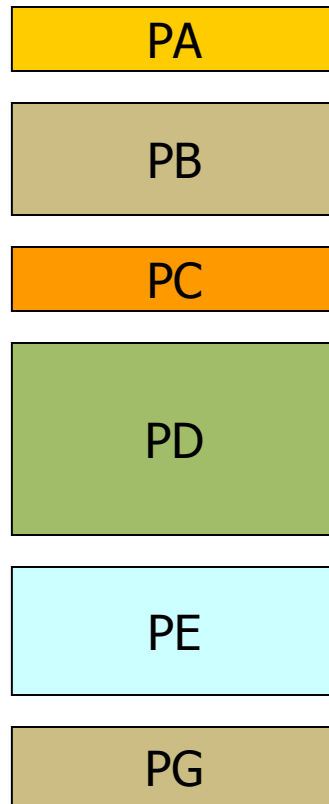
Problems

- As process are created and destroyed, a lot of small partitions may arise
 - These are called “holes”
 - This problem is called “external fragmentation”
 - It may be overcome with *compaction*
 - ✓ But it takes a lot of time...
- Analogy
 - ✓ The Norton Disk de-fragmenter



Fragmentation

Process memory requirements





Placement algorithms

- The placement algorithm is very important to minimise the holes
 - Which free partition to split when a process must be loaded?
 - We can choose the smallest partition which the process fits in
 - ✓ Best-fit
 - ✓ This generates a small hole
 - We can choose the first partition we find
 - ✓ First-fit
 - We can choose the next partition after the last one we have assigned
 - ✓ Next fit
 - Which one is best???
 - ✓ there is not best! depends on the sequence of processes



Buddy system

- It is similar to the dynamic partitioning
 - Partitions can only be of size 2^i
 - Initially there is one single partition
 - When a process of size s arrives
 - ✓ Find the smallest i such that $s < 2^i$: $i = \lceil \ln(s) \rceil$
 - ✓ If there is a block of size 2^i , then it is assigned to the process
 - ✓ Otherwise, find the smallest block with size greater than s , suppose it is of size 2^k
 - ✓ Split the block into two smaller blocks of size 2^{k-1}
 - If $i = k-1$, assign one block to the process
 - Otherwise, go to step 3



Buddy system

- It is a good compromise between speed and need to reduce fragmentation
 - The internal fragmentation is not so big, especially for small processes
 - There are other superior systems
 - However, in some special application it is useful and it is still implemented



Relocation

- We need a way to relocate the process when it is first loaded and when it is swapped in again
 - i.e. the address references in the program must be *translated* into actual addresses in memory
- We distinguish
 - Logical address: a reference in memory made by the program that needs to be translated into actual address in memory
 - Relative address: a reference in memory relative to some know point, for example the beginning of the program
 - Physical address: the real address in the physical memory after the translation



Hardware support for relocation

- All high-end processor have an internal module called MMU (memory management unit)
 - Every reference in memory is first passed to the MMU that actually translates the logical address into a physical address



Paging

- Memory is divided into *frames* of small size
 - Very similar to fixed partitions
 - but these are very small (typically 4Kb)
- The memory of each process is also divided into pages of the same size
 - Each page of the process can be loaded into one frame
 - The pages must not be necessarily consecutive in memory
 - Internal fragmentation is very small
 - ✓ Only the last part of the last page
 - External fragmentation is 0
- Single pages can be moved out or in
 - One process does not need to be entirely on the main memory at the same time!



Pages and Frames

Process PA

1
2
3
4
5

Process PB

1
2
3

Phys.
Memory
(frames)

3
2
3
4
1
1
2
5

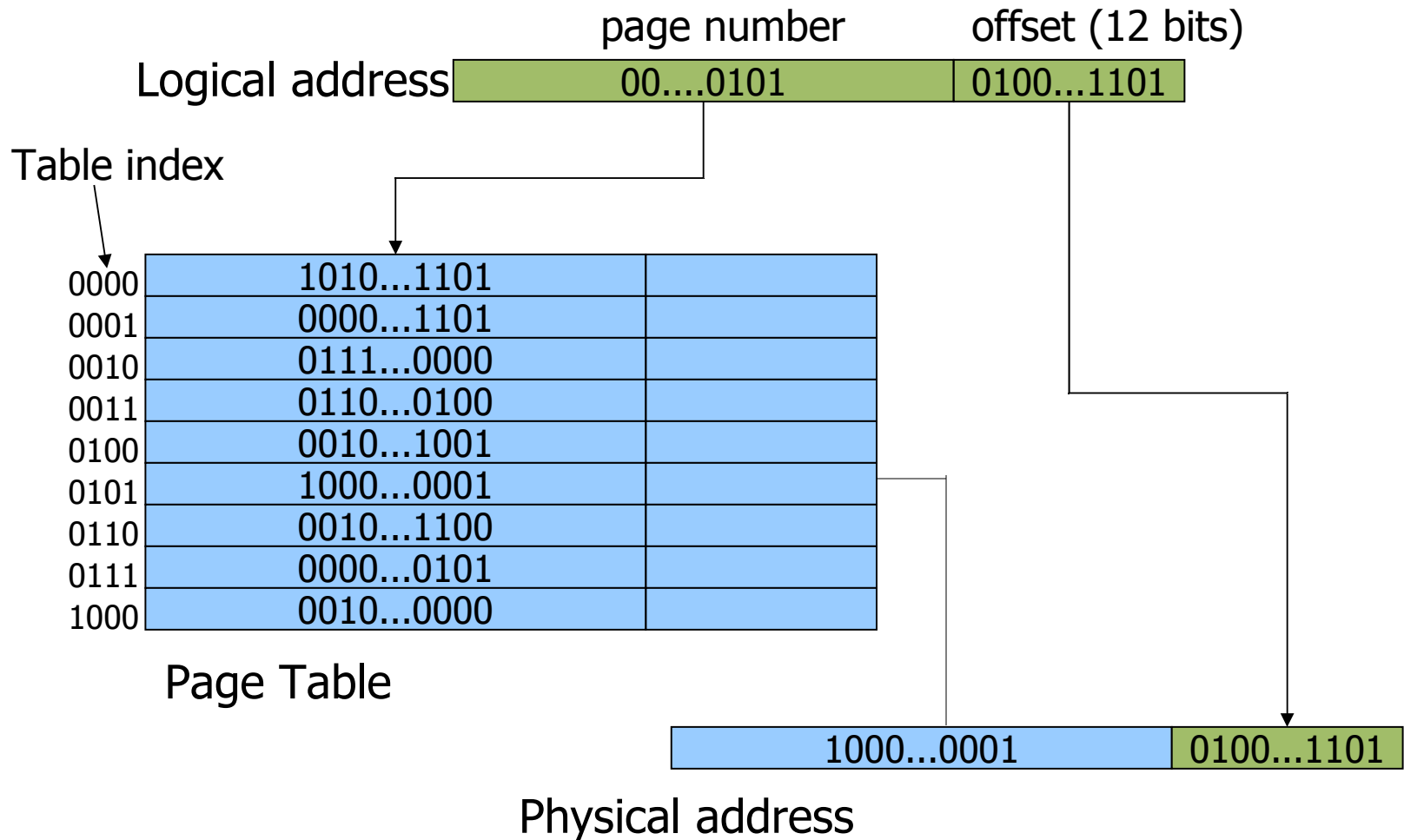


Page table

- Each logical address can be divided into two parts
 - The first part is the page number
 - The second part is the offset relative to the beginning of the page
- Example: pages of 4 Kb, addresses on 32 bits
 - 20 bits are for page numbers
 - 12 bits are offsets
- The OS maintains a page table for each process
 - For each page, it stores the frame address in the real memory
- To translate logical addresses into physical addresses we need
 - Obtain the page number
 - Obtain the frame address
 - Add the offset to the frame address
- Hardware support: see next...



Page table





Segmentation

- Alternative to paging
 - A program is divided into “segments” of different size
 - Each segment is assigned a base address and a bound
 - Each program is associated a “segment table”
 - Very similar to paging but segments are fewer and larger
 - The division in segments is usually done by the compiler and by the linker
 - For example, one or more segments for the code, one or more segments for the data



Virtual memory

- By using paging and segmentation, we are ready to implement a memory management system (virtual memory)
- Usually, virtual memory is based on paging,
- Although all modern virtual memory systems use both paging and segmentation
- Let us start by describing the basic hardware mechanisms necessary to virtual memory



Paging

- Let us start by analysing the paging mechanism in more details
 - In the page table, we need two bits
 - ✓ P: is the page is present in memory. In virtual memory it is possible to swap out only some of the process pages
 - ✓ M: set to 1 when the page is modified. If it is not modified we do not need to write it to disk when we swap out (think of code pages for example)
 - Other bits can be present
 - ✓ S: if the page is shared or not
 - ✓ U: if the page was recently used or not



Hardware structures for paging

- Page table must be loaded in memory
- However, page tables may be huge
 - 2^{20} is the maximum number of elements of a process page table!
- This means that page tables are stored *in virtual memory*
 - Page tables are kernel data structures
 - They stay on the virtual memory of the kernel
 - Part of the page table can be swapped out on disk

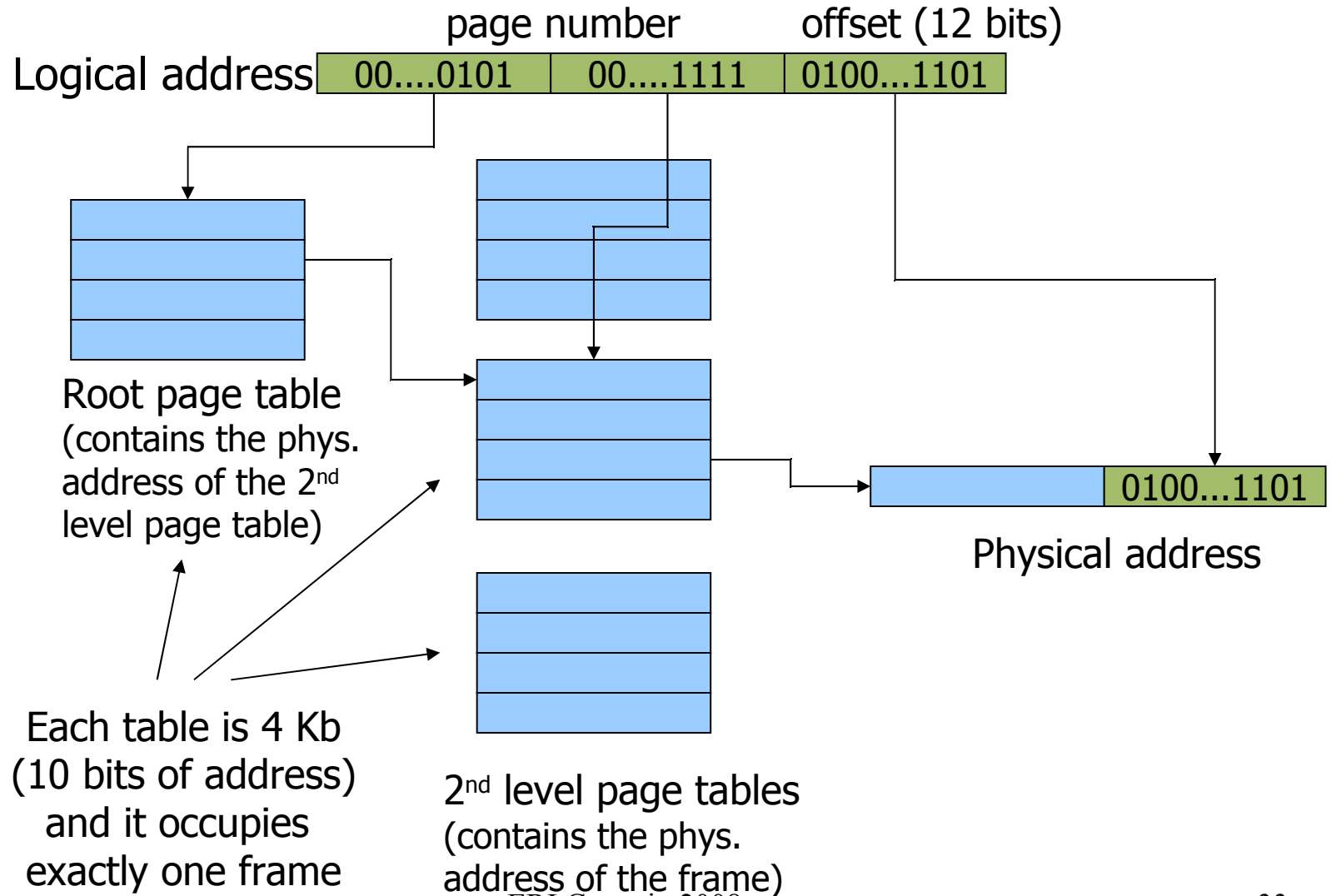


Two-level structure

- Some processor makes use of a two level structure
 - A root page table holds address of segments of the page table
 - The page number is divided into two parts:
 - ✓ 10 bits for the index in the root page table
 - ✓ 10 bits are offsets in the page table
- Another mechanism consist in using an hash table



Two-level structure



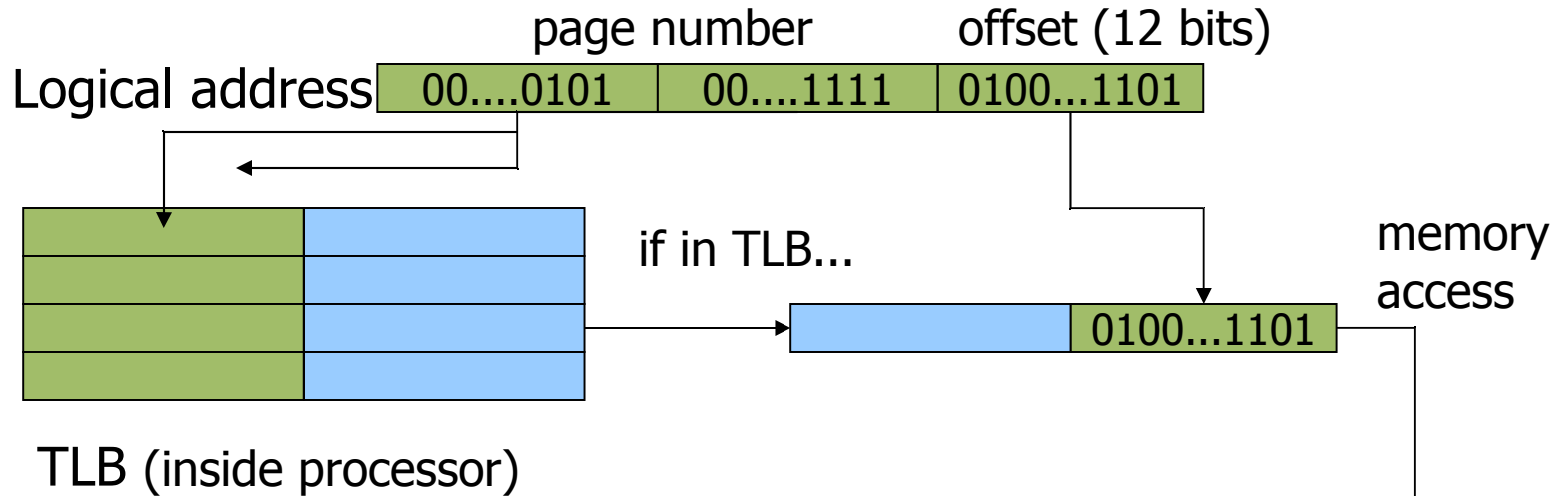


TLB

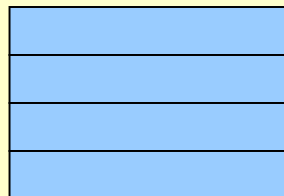
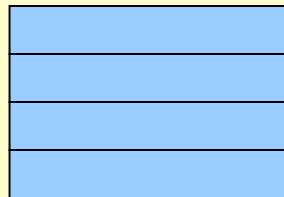
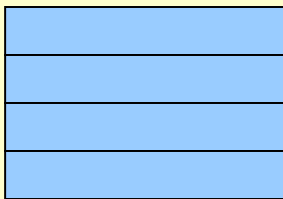
- A translation lookaside buffer is used to speed up the paging mechanism
 - Each access in memory would require two more accesses
 - ✓ One for getting the frame address
 - ✓ The other one for the real access
 - TLB is a cache for frame addresses
 - It is a small array that contains a small subset of elements of the page table
 - It is managed like a cache
 - Following the principle of locality of a program, only rarely we need to access the page table in memory



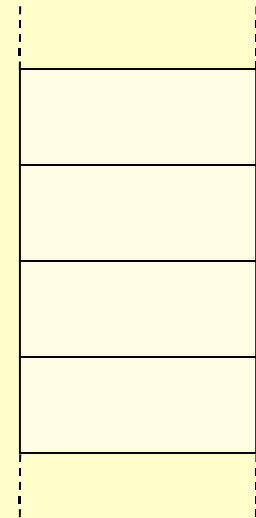
TLB (successful hit)



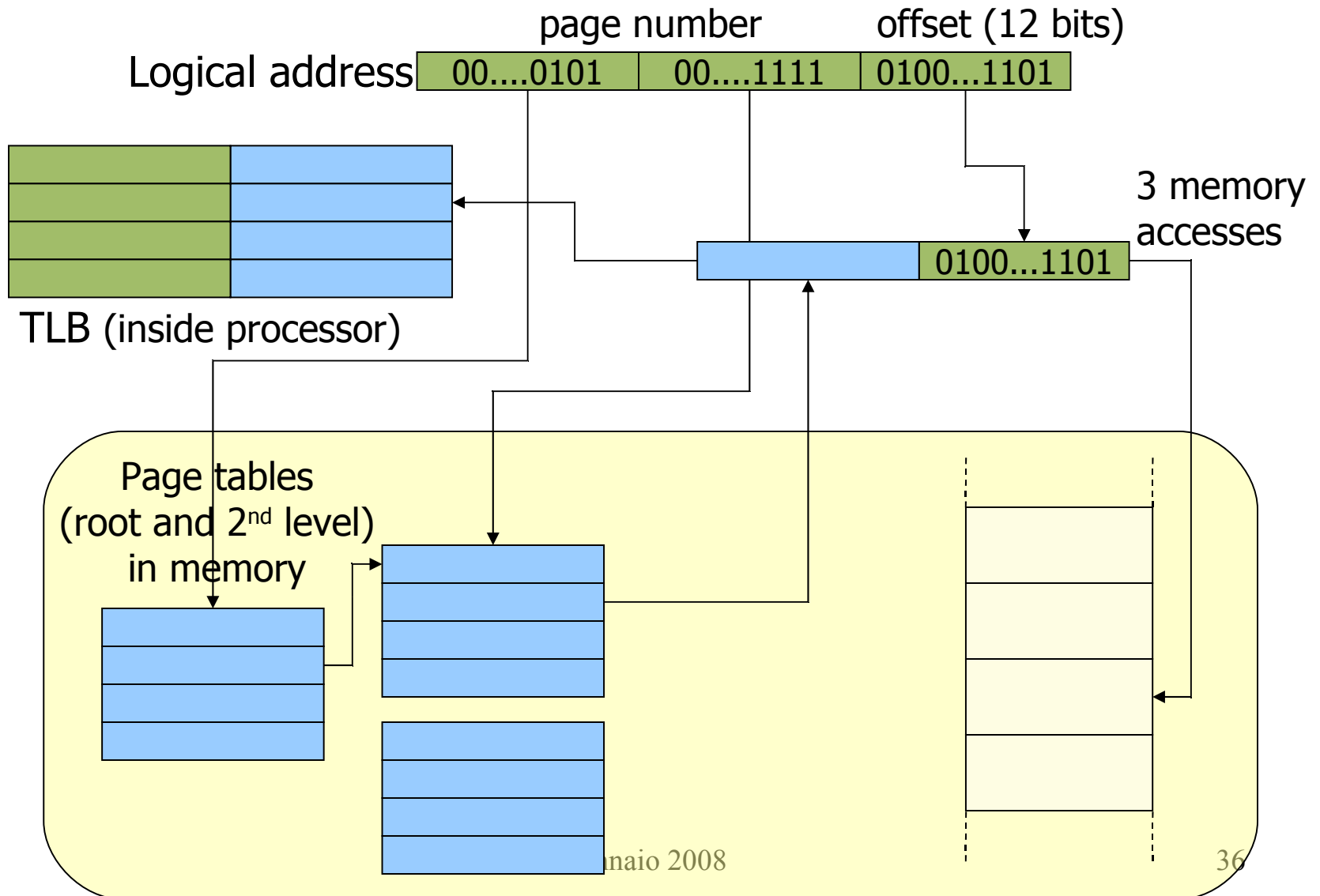
Page tables
(root and 2nd level)
in memory



maio 2008



TLB (miss)





Page fault

- What if a page is not present in main memory?
 - If a process accesses a page with $P=0$, an exception (trap) occurs and an operating system interrupt handler is executed
 - The interrupt is part of the virtual memory subsystem
 - It must decide:
 - ✓ Whether to load just one page or many
 - ✓ Whether to unload some other page



Segments

- In most modern processors, paging and segmentation techniques are combined
 - Paging
 - ✓ avoid external segmentation
 - ✓ Nice way of supporting virtual memory
 - Segments
 - ✓ avoid internal segmentation
 - ✓ Provide protection
 - ✓ Provide sharing

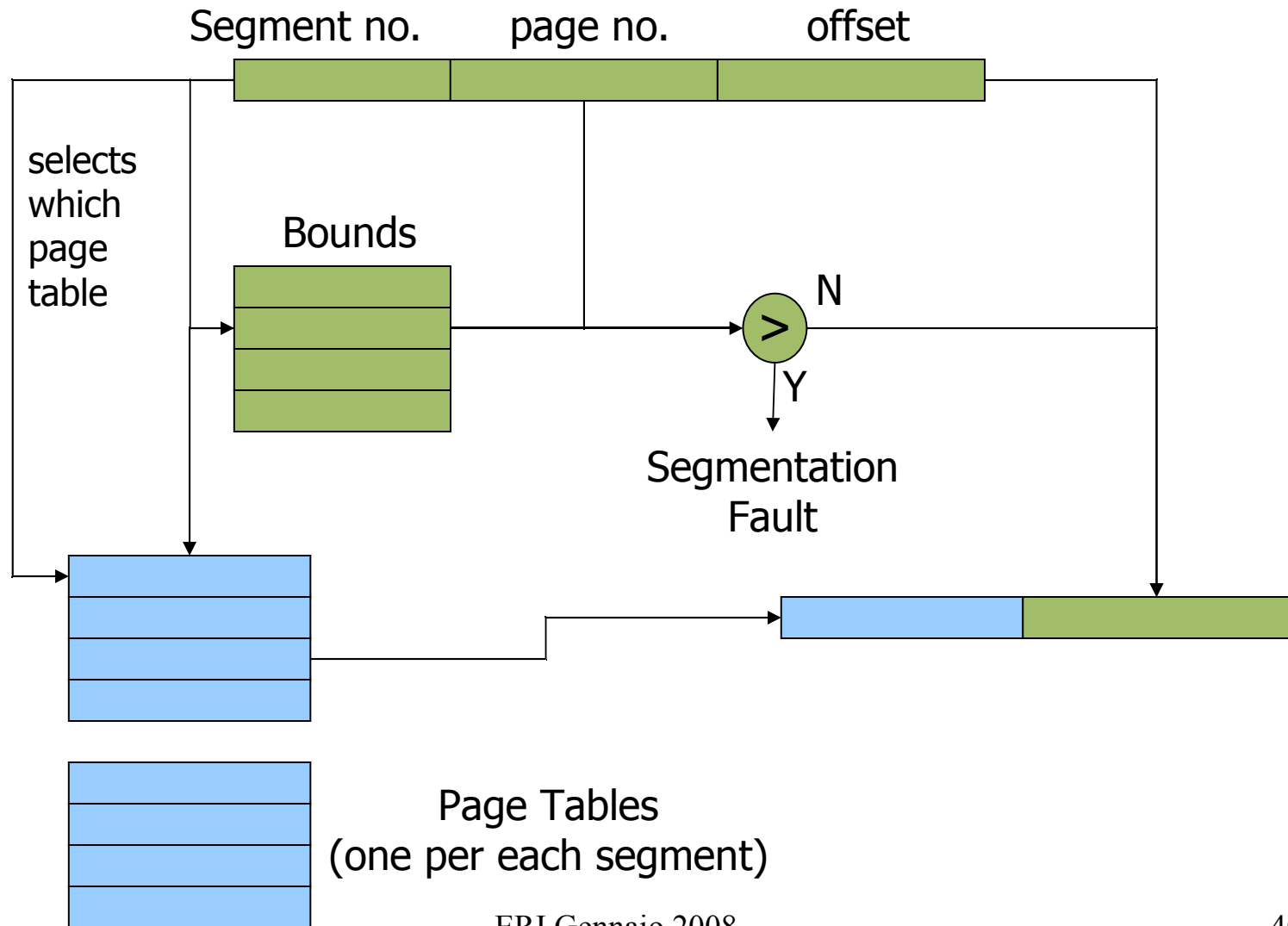


Combining segments and pages

- For each process, a set of segments
- For each segment, a different page table
 - The address is divided into three parts
 - ✓ Segment number
 - ✓ Page number
 - ✓ offset in the page
 - Each segment is associated a base address (that starts with the page table) and a bound address Translation
 - ✓ Get the segment base address by looking in the segment table
 - ✓ Check with the bounds
 - ✓ if ok:
 - add the page number: obtain an address in a page table
 - get the frame address
 - add the offset: obtain the final address



Segments and pages





Exercise

In a computer system, the virtual address space consists of addresses of 16 bits length. The processor uses pages of 1 Kb (i.e. 10 bits). Moreover, the processor has a TLB containing only 2 page table entries. The TLB is managed with a LRU policy. Consider the following page table for process PA:

Page	Frame	P
0000 01	1110 10	1
0000 10	1000 10	1
0000 11	1001 01	1
0001 00	1010 10	1
0001 01	0010 00	0
0001 10	0101 01	0
0001 11	0101 11	1
0100 00	0010 01	1
0100 01	1100 00	1

Initially, the TLB is empty. For each memory reference (in read only):

- (2 *points*) compute the real memory address after the translation, or write PF for a page fault, or IA for an illegal access;
- (3 *points*) compute how many memory accesses are necessary (consider the TLB mechanism)

(memory references in next slide)

Exercise

Page	Frame	P
0000 01	1110 10	1
0000 10	1000 10	1
0000 11	1001 01	1
0001 00	1010 10	1
0001 01	0010 00	0
0001 10	0101 01	0
0001 11	0101 11	1
0100 00	0010 01	1
0100 01	1100 00	1



Virtual address	Physical address	# mem acc.
0001 0011 0101 0101	1010 1011 0101 0101	3
0001 1110 0000 0000	0101 1110 0000 0000	3
0001 0011 0111 1000	1010 1011 0111 1000	1
0001 1110 0000 1001	0101 1110 0000 1001	1
0100 0000 1000 1111	0010 0100 1000 1111	3
0001 0011 1111 1111		
0100 0011 1111 1111		
0100 0111 1111 1111		
0100 0011 1111 0101		

TLB

0100 00	0010 01
0001 11	1010 11



Operating system code

- Many different techniques can be used to implement virtual memory
 - Almost all techniques are based on paging
- Three “things” to be decided
 - Fetch Policy
 - ✓ DEMAND PAGING: Pages can be loaded on demand (i.e. only when a page fault occurs)
 - ✓ PREPAGING: Pages can also be loaded in advance (to take advance of the disk mechanisms)
 - Placement policy
 - ✓ With paging, this is not a problem, since all frames are equivalent
 - Replacement policy
 - ✓ Which page has to be unloaded



Replacement Policy

- Three subproblems
 1. How many pages are to be allocated to each process
 2. Whether to unload pages
 - ✓ Of the same process that made the page fault
 - ✓ Of all the processes
 3. Among the set of pages to be consider, which has to be actually unloaded
- Problems 1 and 2 are referred as the “resident set management” problem
- Problem 3 is referred as the “replacement policy” problem



Replacement

- Let us start from the last problem
 - the objective is to replace the pages so to have the least number of page faults
- Possible algorithms
 - Optimal (cannot be implemented)
 - Least recently used (LRU)
 - First-in first-out (FIFO)
 - Clock
- Optimal policy is used as a reference to see how good is one algorithm
 - Select the page for which the next reference time is the longest
 - This algorithm results in the least number of page faults
 - However, it is not possible to know the future



LRU

- Policy
 - For each page, maintain a time-stamp of the time the page was referenced last
 - Unload the page with the smallest time-stamp
- It takes a lot of overhead
 - It requires a list of pages ordered by timestamp
 - ✓ Every time a page is referenced, update the time-stamp and re-order the list
 - ✓ Re-ordering takes up a lot of time
 - Not feasible in a real OS



FIFO

- Policy
 - The first page loaded in memory is the first to be unloaded
 - We need a circular list of pages
 - Every time a page is loaded, insert it in the head of the list
 - Every time a page must be unloaded, remove from the tail of the list
- Very little overhead
 - But not very good
 - Why?



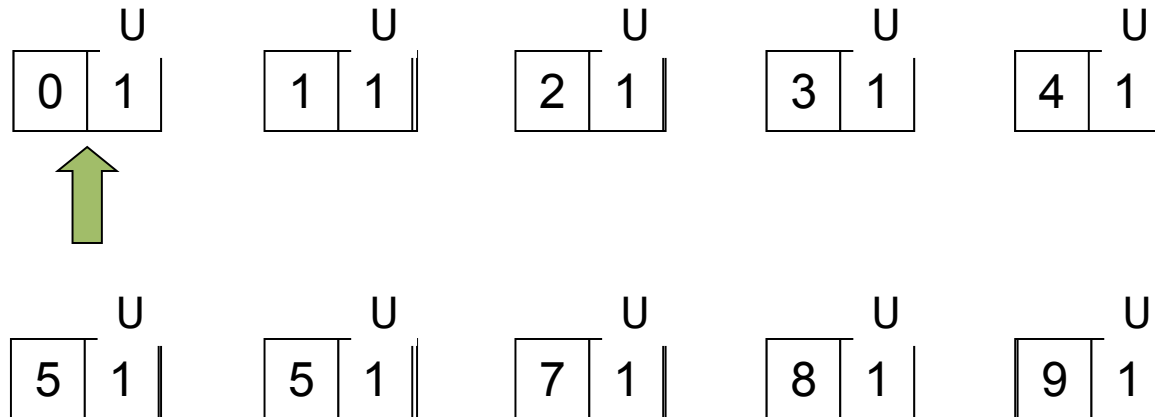
Clock

- It is a combination of both LRU and FIFO
- It has a reasonable overhead
- Policy
 - Every page has a use bit (U)
 - Every time the page is first loaded, $U = 1$
 - Every time the page is accessed $U = 1$
 - When a page has to be unloaded
 - ✓ Use one pointer to a certain page, initially page 0
 - ✓ Analyse all frames in turn, clockwise
 - ✓ If a page has $U=1 \rightarrow U=0$
 - ✓ If a page has $U=0$, unload it



Example of Clock algorithm

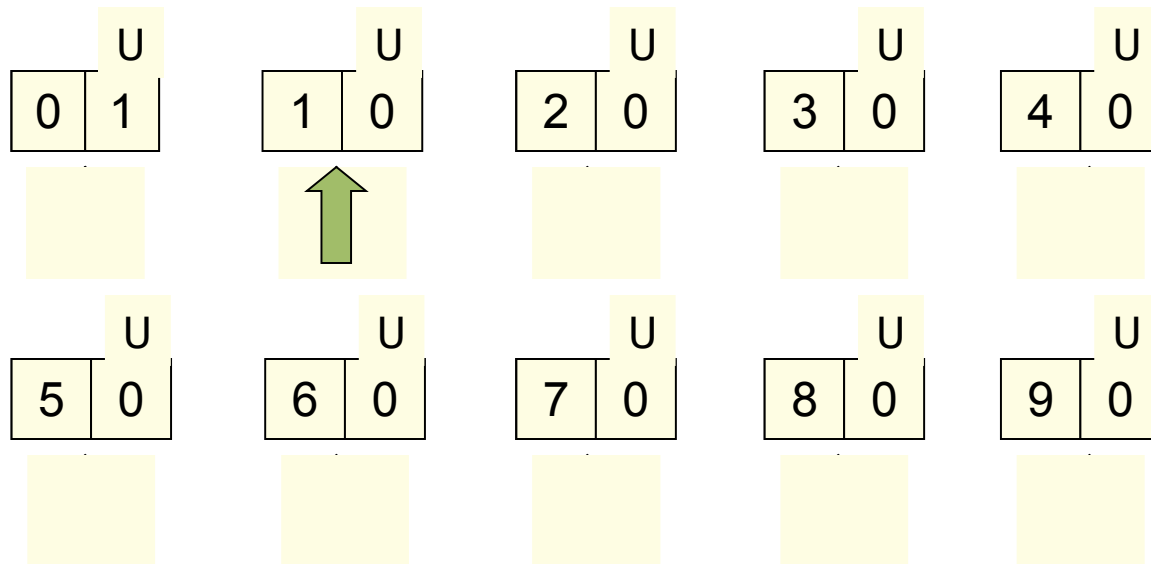
- Initially, all use bit are set to 0
- when the page is loaded, $U = 1$





Example Clock: page fault

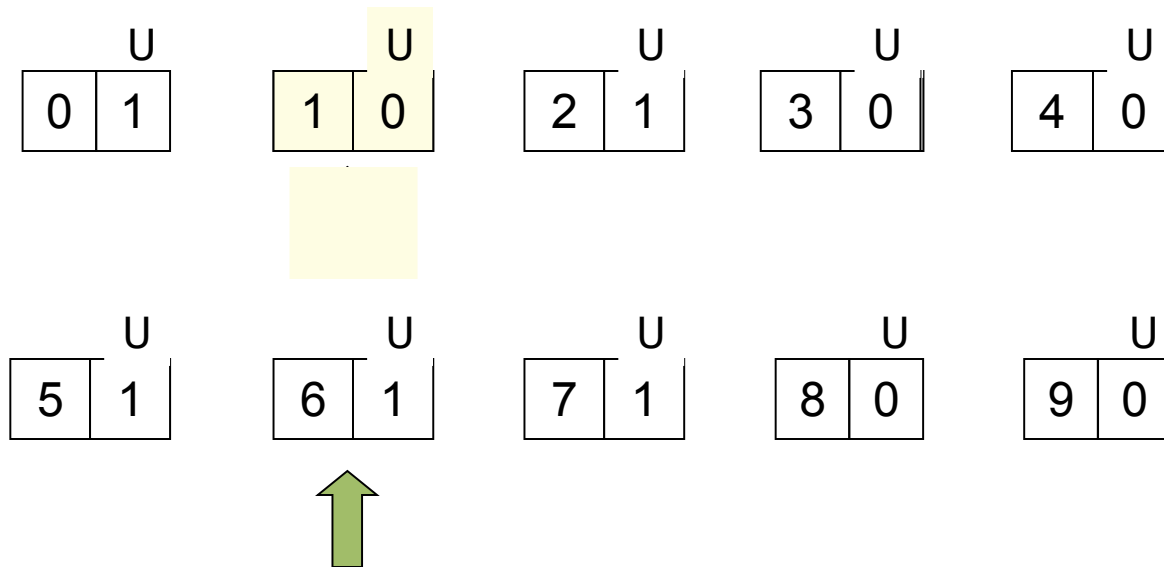
- When a page has to be replaced:
 - The pointer is shifted clockwise until it finds $U=0$;
 - while it passes, if it finds $U=1$ then sets $U=0$
 - The first time, the first page is replaced





Example clock: one more round

- Now suppose that some page is used, and then another page fault
 - the one to be replaced is the least recently used from last clock turn (approximately)





Clock

- It is not LRU
 - But it approximates it
 - Also, overhead is limited
 - Performance is between LRU and FIFO
- Variations
 - Use both the U and M bits



Caching

- Instead of unload a page, we can put it in a temporary buffer in memory
 - If there is the need for more space, take from the buffer, and move an old page into the buffer
 - If a page in the buffer is needed, move that page again in memory
 - In this way, if a page is immediately referenced after it has been moved to the buffer, we can move it again in the main memory at very little cost



Replacement Policy

- Three subproblems
 - How many pages are to be allocated to each process
 - Whether to unload pages
 - ✓ Of the same process that made the page fault
 - ✓ Of all the processes
- 3. Among the set of pages to be consider, which has to be actually unloaded



Resident set management

- The set of pages in memory for a process is called “resident set size”
- How large should be the resident set?
 - Two policies
 - ✓ Fixed set size
 - ✓ Variable set size
 - Fixed set size
 - ✓ Every process is assigned a fixed number of frames
 - ✓ The number of frames process never changes
 - Variable set size
 - ✓ The number of frames assigned to each process may change
 - ✓ It could increase or decrease



Replacement scope

- If a page fault occurs, which page should be unloaded?
- Two policies:
 - Local scope
 - ✓ one page of the same process should be unloaded
 - Global scope
 - ✓ one page from any process can be unloaded



Replacement scope and policy

- The scope and the resident set management are not independent
 1. Fixed allocation and local replacement
 - Number of frames allocated to process is fixed
 - Page to be replaced is chosen from among the frames allocated to the process
 2. Variable allocation and local replacement
 - The number of frames allocated to process is variable
 - Page to be replaced is chosen from among the frames allocated to the process
 - Periodically, enlarge or restrict the resident set
 3. Variable allocation and global replacement
 - The number of frames allocated to process is variable
 - Page to be replaced is chosen from among the frames allocated to all the process



Replacement scope and policy

- Policy 1: fixed allocation and local scope
 - It is not flexible and it is never used
 - Fixing a priori the number of frames is too restrictive
- Policy 3: variable allocation and global scope
 - When a page has to be unloaded, which process should be penalized?
 - ✓ Several possibilities, but a few work
 - The size of the resident set of the process that made a page fault is increased by 1
 - The size of the resident set of the process that has been penalized is decreased by 1



Variable allocation and local scope

- A lot of research has been concentrated on this kind of policy
- When a page fault occurs, a frame from the same process' resident set is unloaded
- Periodically, the resident set size is increased or decreased
 - Intuitively, if the number of page faults is large, it is better to increase the resident set size
 - If the number of page faults is very low, it is better to decrease the resident set size to make space for other processes
 - A simple page fault measurement is not the best thing to do...



Working set

- $W(t, \Delta)$ is the *working set*
- It is the set of pages of a process that has been accessed in the last Δ instants in which the process has executed
- The working set is a non decreasing function of Δ
 - As Δ increases, the working set increases
- The working set changes as the time passes
 - If a process accesses only 1 page for Δ consecutive units of time, the working set has size 1
 - If a process has a high degree of locality, the working set is small
 - If the process has a “jump” and changes abruptly from one set of pages to another set of pages, the working set enlarges rapidly



Working set

- Strategies
 - Monitor the working set
 - Periodically remove from the resident set of a process the pages that are not in the working set
 - A process can execute only if its resident set includes the working set
- Problems
 - It is difficult to keep track of the working set
 - The past does not always predict the future
 - The optimal value of Δ is unknown and it depends on the process structure
- Practical strategies approximate the working set strategy



Page Fault Frequency

- Instead of the keeping track of the working set, we track the page faults
 - Use reference counter for each process, initialized to 0
 - Define a minimum threshold L and a maximum threshold H
 - When a page is accessed
 - ✓ Set the Use bit to 1
 - ✓ Increase the reference counter
 - When a page fault occurs
 - ✓ If the reference counter is smaller than L , increase the resident set size
 - ✓ If the reference counter is greater than H , decrease the resident set size by unloading the pages with use bit = 0
 - ✓ Set the reference counter to 0, and all use bits to 0



PFF

- The basic idea is to “measure” the frequency of page faults
- if page faults occur often,
 - then the reference counter between two faults does not increase much
 - if it is below L , then increase resident set
 - higher frequency \rightarrow need to increase resident set
- If page faults occur rarely
 - then the reference counter between two faults increases a lot
 - if it is greater than M , then decrease resident set
 - lower frequency \rightarrow need to reduce resident set



PFF

- One problem
 - During transient periods, when there is a shift to a new locality, there is a large number of page faults in a short period
 - ✓ reference counter cannot increase much, then always lower than L
 - Suddenly, the size of the resident set could become very large
 - ✓ every page fault there is an increase
 - ✓ the size of the resident set however is not reduced with the same velocity
 - Some other process could suffer from these memory peaks
 - ✓ the process that change location takes all the space in a greedy way
- Need to find a better strategy!



Variable-interval Sampled Working Set

- VSWS
- In this strategy, we define an interval
 - At the beginning of the interval, all use bits are reset to 0
 - When a page is accessed, its use bit = 1
- At the end of the interval, the use bits are scanned
 - All the pages with use bit = 1 are kept in the resident set
 - All the pages with use bit = 0 are discarded
- At interval boundaries, the resident set can only decrease
- During the interval, when a page fault occurs, the resident set is increased by 1



VSWS

- M: minimum duration of the sampling interval
- L: maximum duration of the sampling interval
- Q: maximum number of page faults in the interval
- 4. virtual time (e.g. number of memory references)
- 5. If the virtual time since last sampling instance reaches L, suspend the process and scan the use bits
 - discard all the pages with use bit = 0, set all remaining pages' use bit = 0;
- 6. If, priori to an elapsed virtual time of L, Q page faults occur
 - If the virtual time is less than M, wait until the elapsed time is equal to M
 - If the virtual time is greater than M, suspend the process and scan again the use bits



VSWs

- The interval is not fixed
 - It can vary between M and L , depending on the amount of time in which Q page faults have occurred
 - When the number of page faults is large (for example during transitions) the interval is short (but never shorter than M), so the resident set cannot be too large
 - When the number of page faults is small, the interval is large (but never larger than L), and the resident set does not change much



Load control

- The number of processes that are totally or partially in memory is called *multiprogramming level*
- Problem: Trashing
 - If there are too many processes in memory at the same time, the resident sets are necessarily small
 - A high number of page faults is likely
 - The system spends most of the time loading pages from the disk!
 - The processor utilisation falls down dramatically



Reducing the multiprogramming level

- If the number of page faults is too large, we must reduce the number of processes in memory, by swapping out some process completely
 - L=S criterion
 - ✓ the mean time between faults must be equal to the average time needed to load a page
 - ✓ If the two are different the multiprogramming level is increased or decreased
- Which process to swap out?
 - Many possibilities!!
 - ✓ Lowest priority process
 - ✓ Faulting process
 - ✓ Largest process
 - ✓ ...