Operating Systems
“Virtual Memory”

Mathieu Delalandre
University of Tours, Tours city, France
mathieu.delalandre@univ-tours.fr

Lecture available at http://mathieu.delalandre.free.fr/teachings/operating2.html

Operating Systems
“Virtual Memory”

1. Introduction
2. Demand paging
3. Performance issues
4. Page table for large memories
5. Allocation of frames
6. Page replacement algorithms

Introduction (1)

Memory management, performances and limitations: with standard memory-management algorithms the instructions being executed must be in physical memory. This requirement is unfortunate for different reasons.

- **Useless memory allocation**
  - **Exception code:** programs often have code to handle unusual error conditions, that is almost never executed.
  - **Dynamic allocation:** arrays, lists, and tables are often allocated with more memory than they actually need.
  - **Expert functions:** certain options and features of a program may be used rarely.
  - **Asynchronous execution:** even where the entire program is needed, it may not all be needed at the same time.

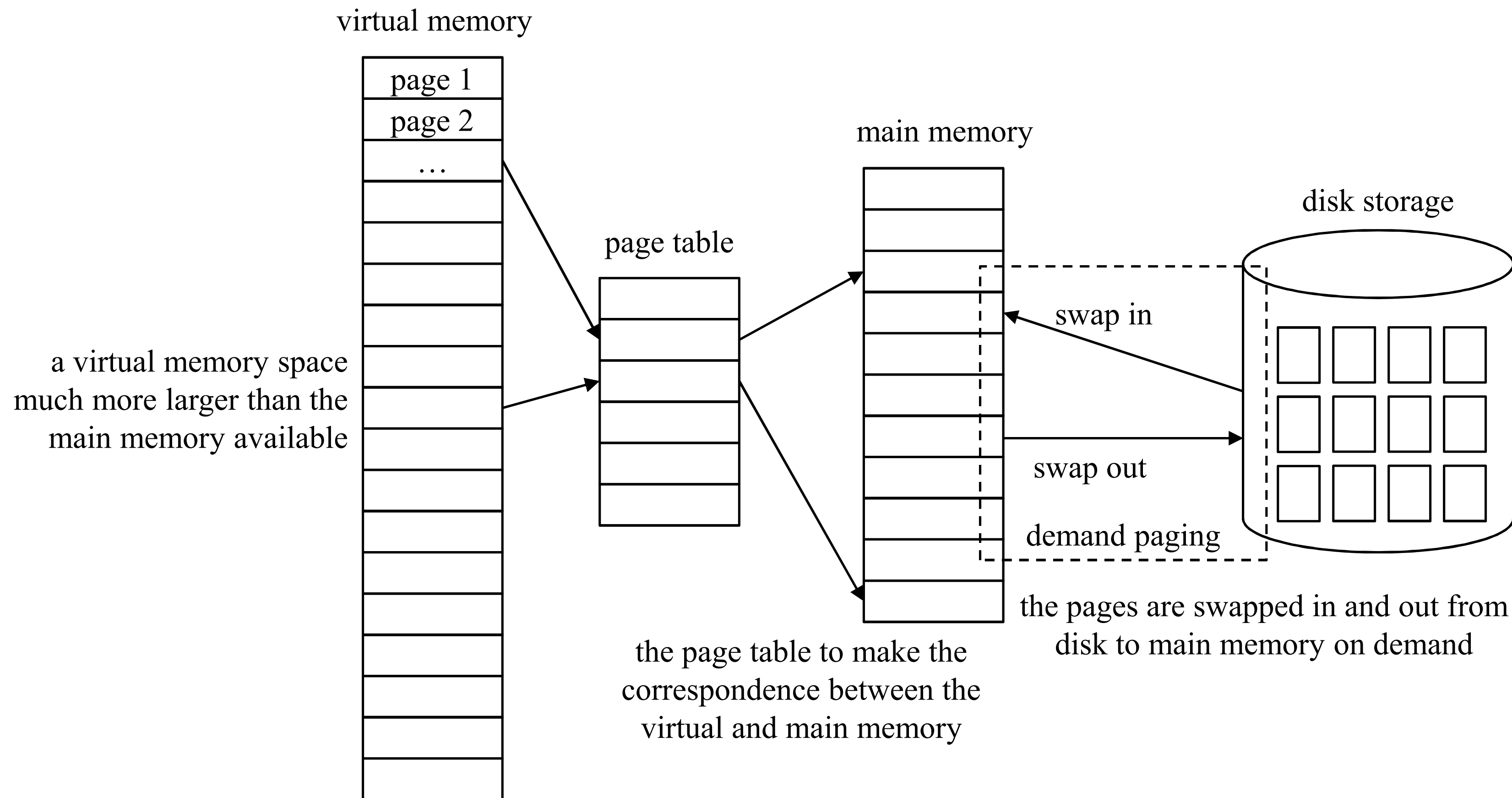
- **System performances**
  - **Parallelism:** without complete loading, more programs could run at the same that increases the CPU utilization and throughput.
  - **I/O:** without complete loading, less I/O would be needed to load or swap user programs into memory, so each user program would run faster.

- **Programming task:** a program would no longer be constrained by the amount of physical memory. Users would be able to write programs for an extremely large address space, said “virtual”, simplifying the programming task.

- **Software frameworks:** some frameworks cannot be loaded entirely into main memory (database and multimedia applications, scientific computing, etc.).

Introduction (2)

Virtual memory: allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available. The two key features are (i) a separation of logical memory as perceived by users from physical memory (ii) to run a program that is not entirely in memory. The virtual memory extends the paging system with demand paging.



Operating Systems
“Virtual Memory”

1. Introduction
2. Demand paging
3. Performance issues
4. Page table for large memories
5. Allocation of frames
6. Page replacement algorithms

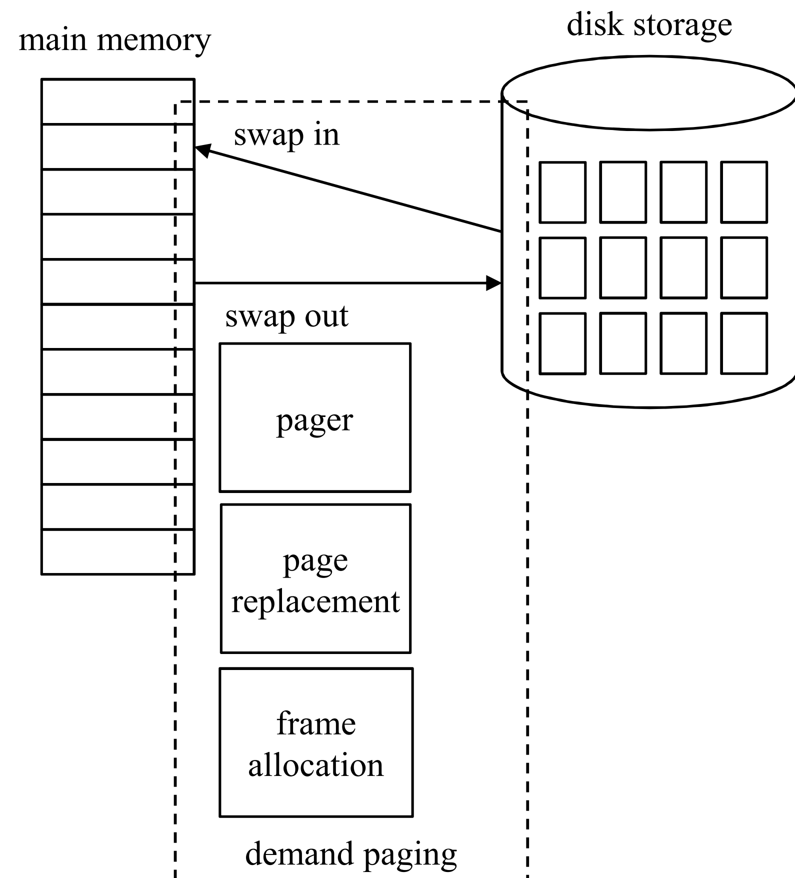
Demand paging (1)

Demand paging system is similar to a paging system with swapping, where processes reside in disk memory.

Pager: we use a lazy swapper that swaps a page into memory unless that page will be needed. The term swapper refers to a process handler and is technically incorrect, we can employ the term pager.

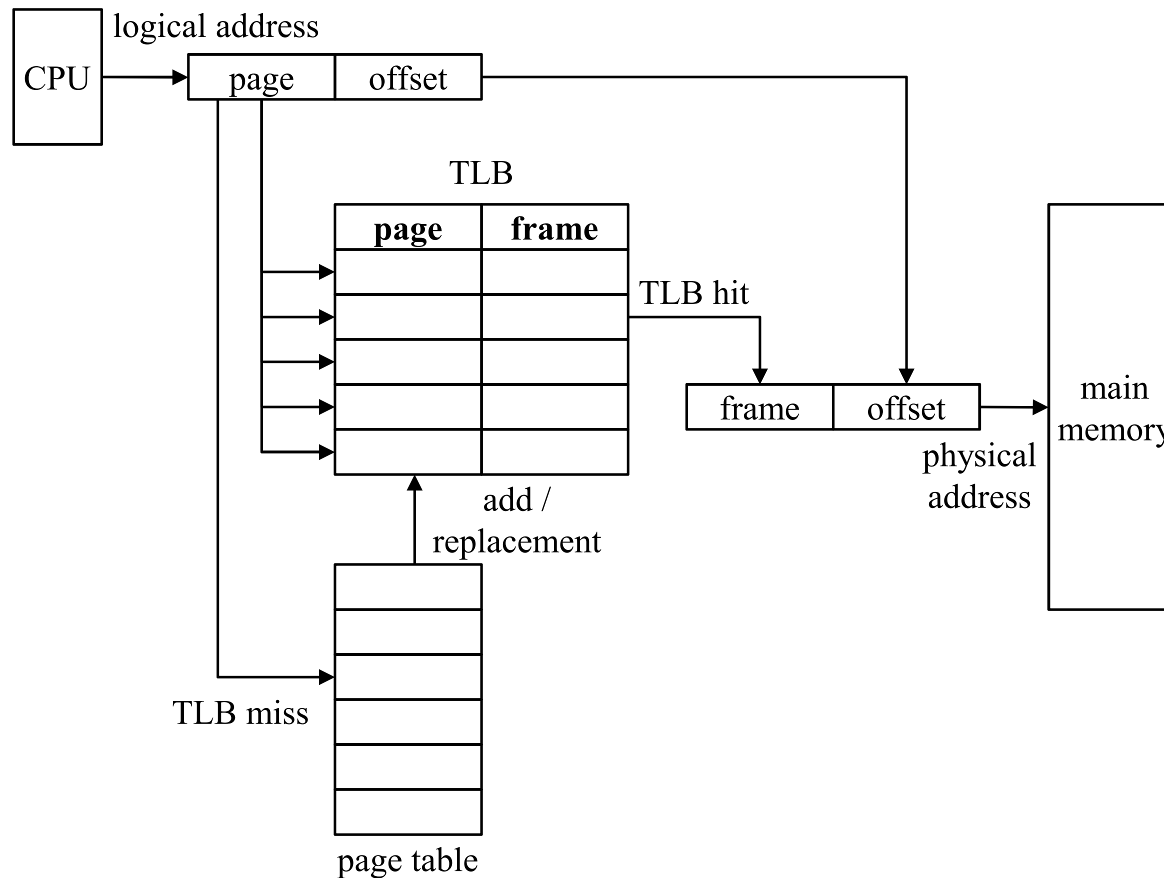
Page replacement: when a page fault occurs (a page needed not in memory), the OS has to choose a page to remove. The system performance is much better if a page that is not heavily used is chosen, this is the charge of page replacement algorithms.

Frame-allocation: if we have multiple processes in memory, we must decide how many frames to allocate to each process. This is the frame-allocation problem.



Demand paging (2)

Transaction look-aside buffer (TLB): in principle every binding from logical address space to physical address space using paging causes two physical memory accesses (1) to fetch the appropriate page table entry (2) to fetch the desired data. This will have the effect of doubling the memory access time. To overcome this problem, we use a special high speed cache for page table entries called Transaction look-aside buffer (TLB).



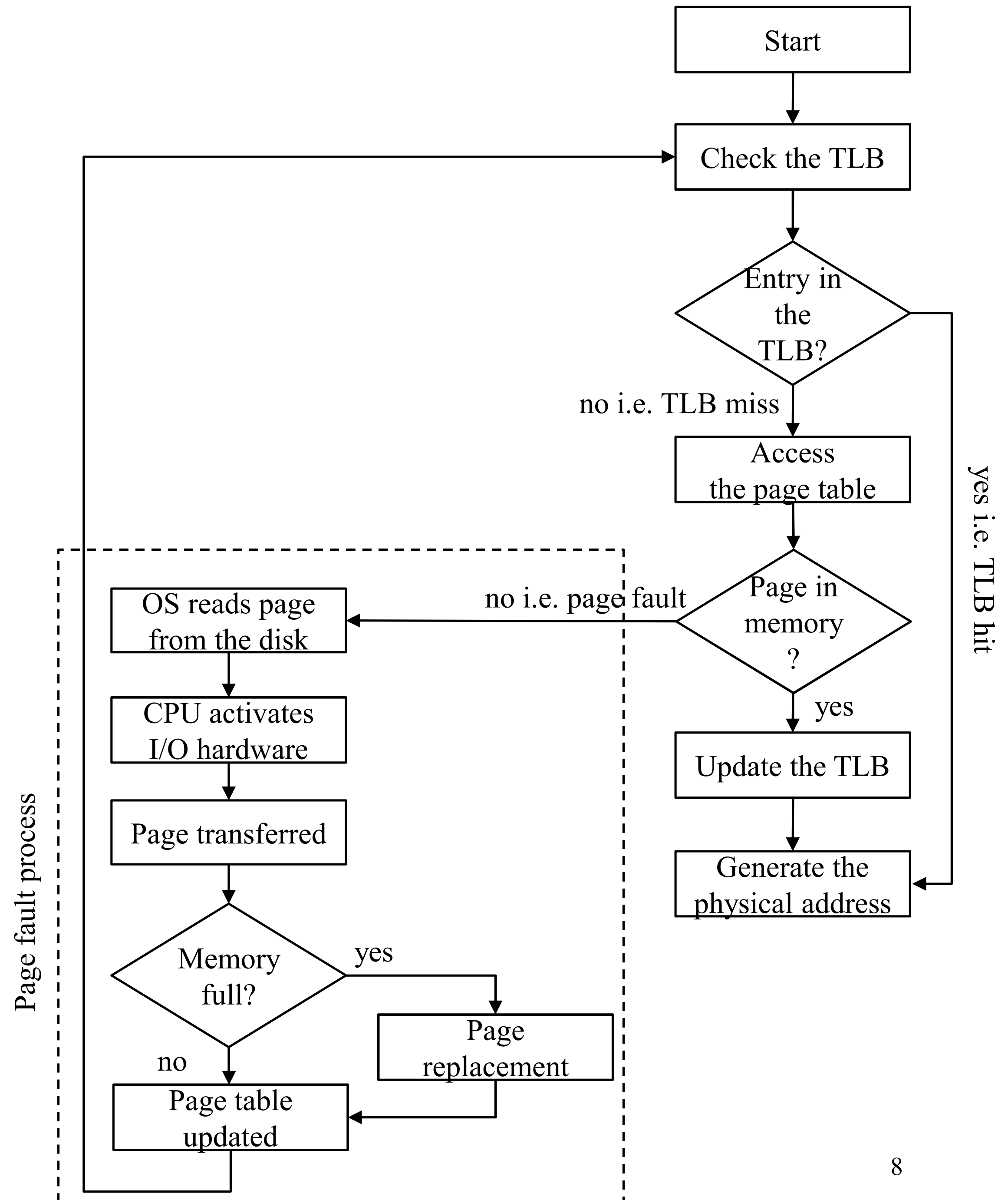
TLB hit: it occurs when the page number is in the TLB.

TLB miss: if the page number is not in the TLB, we add the page number and frame so that they will be found quickly on the next reference. If the TLB is already full of entries, OS must select one entry for replacement (LRU or FIFO policy, etc).

wired down: some TLBs allow entries to be wired down, meaning that they cannot be removed from the TLB.

Demand paging (3)

Page fault: when the desired page is not in main memory a page fault is issued. We leave the realm of hardware and invoke the OS, which loads the needed page and updates the page table.



Demand paging (4)

Structure of a page table entry: the exact layout of an entry in the page table is highly machine dependent, but the kind of information present is roughly the same from machine to machine.

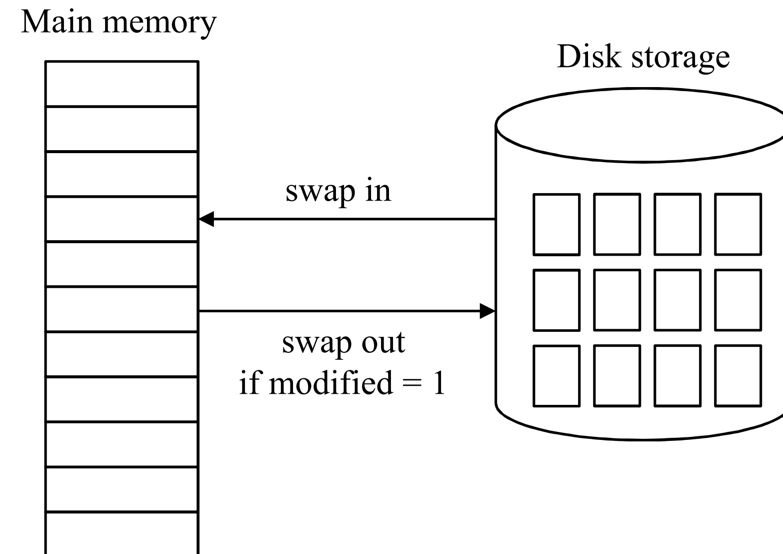
	cashing disabled	use	modified	protection	present/absent	page frame number

The page frame number (f)	the most important …
The present/absent bit	if this bit is 1, the entry is valid and can be used. If it is 0, the virtual page to which the entry belongs is not currently in memory. Accessing a page table entry with this bit set to 0 causes a page fault.
The protection bits	tell what kinds of access are permitted. In the simplest form, this field contains 1 bit, with 0 for read/write and 1 for read only.
The modified bit (m)	when a page is written to, the hardware automatically sets the modified bit (called too dirty bit).
The use or referenced bit (u)	is set whenever a page is referenced, either for reading or for writing.
The cashing disabled bit	this feature is important for pages that map into device registers rather than memory.

Demand paging (5)

Structure of a page table entry: the exact layout of an entry in the page table is highly machine dependent, but the kind of information present is roughly the same from machine to machine.

The modified bit for swaping: if the modify bit is not set, we don't need to write the memory page to the disk. This scheme reduces the I/O time by one-half if the page has not been modified.



Operating Systems
"Virtual Memory"

1. Introduction
2. Demand paging
3. Performance issues
4. Page table for large memories
5. Allocation of frames
6. Page replacement algorithms

Performance issues
“Introduction”

Virtual memory and performances: the experience with numerous OS has demonstrated that virtual memory does work. Accordingly, virtual memory has become an essential component of contemporary OS.

Performance issues
“Trashing”

Trashing: we consider the case where all of the main memory will be occupied with process pieces. If the OS throws out a piece just before it is used, then it will just have to get that piece again almost immediately. This situation is known as thrashing, the system spends most of its time swapping pieces rather than executing instructions.

Consider the following scenario.

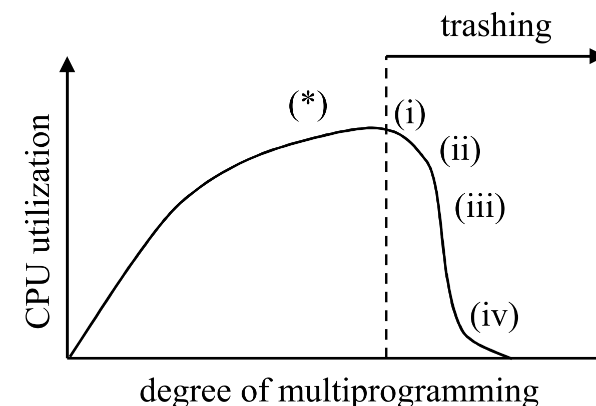
(*) **Low CPU utilization with page replacement:** the OS monitors CPU utilization. If the CPU utilization is too low, we increase the degree of multiprogramming by introducing new processes to the system. A global page-replacement algorithm is used; it replaces pages without regard to the process to which they belong.

(i) **Faulting with cascade:** a process enters a new phase in its execution and needs more frames. It starts faulting and taking frames away from other processes. These processes also fault, taking frames from other processes.

(ii) **CPU utilization decreases:** as the processes queue up for the paging device, the ready queue empties. As processes wait for the paging device, CPU utilization decreases.

(iii) **Parallelism degree increases:** the CPU scheduler sees the decreasing CPU utilization and increases the degree of multiprogramming. The new process tries to get started by taking frames from running processes, causing more page faults.

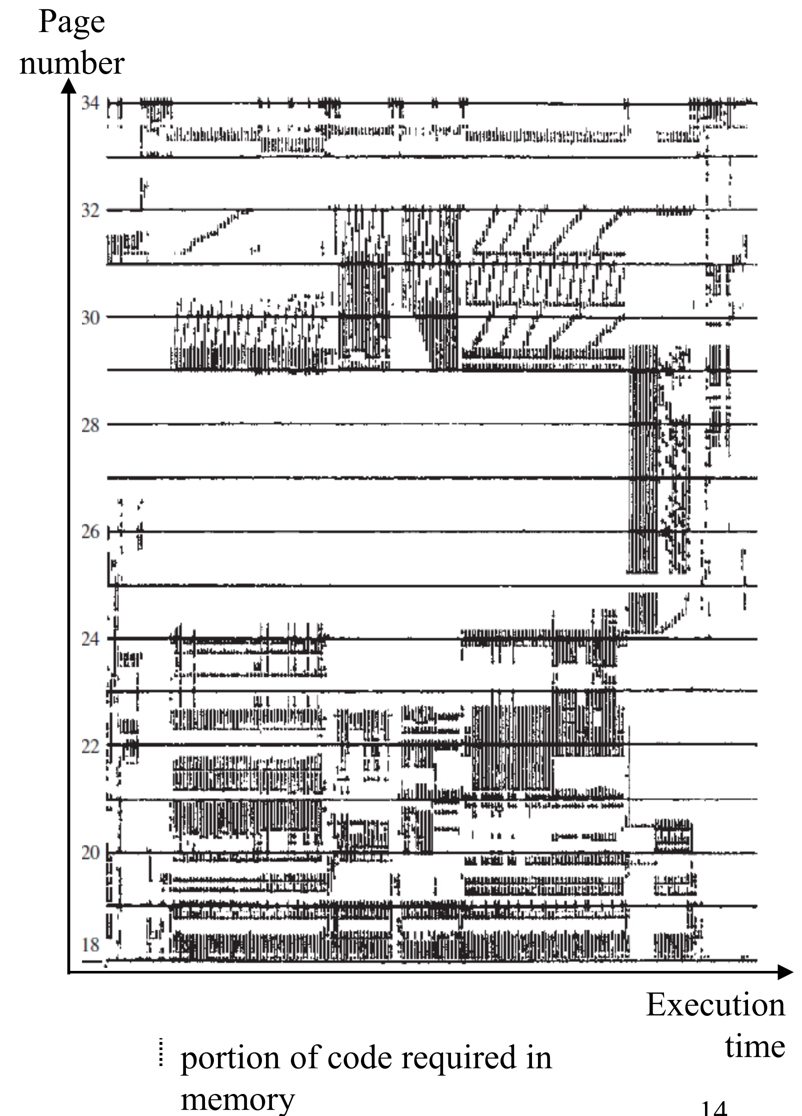
(iv) **Trashing:** the CPU utilization drops even further, and the CPU scheduler tries to increase the degree of multiprogramming. Thrashing has occurred and system throughput plunges.



Performance issues
“Locality and Working-Set” (1)

Principle of locality: the avoidance of thrashing is a major issue. To deal with, the OS tries to guess, based on recent history, which pieces are least likely to be used in the near future. This reasoning is based on belief in the principle of locality, that states that program and data references within a process tend to cluster.

See the famous figure [Hatfield72].



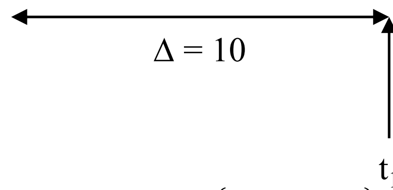
Performance issues
“Locality and Working-Set” (2)

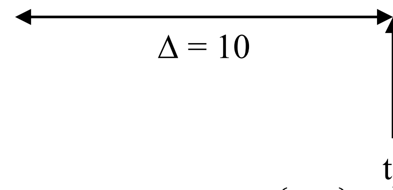
Working-Set model is based on the assumption of locality. This model uses a parameter Δ to define the working-set window. The idea is to examine the most recent Δ page references. The set of pages in Δ is the working set *WS*.

e.g.

Page address stream

… 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 …





$WS(t_1, \Delta) = \{1,5,7,6,2\}$

$WS(t_2, \Delta) = \{4,3\}$

Performance issues
“Locality and Working-Set” (3)

Working-Set model is based on the assumption of locality. This model uses a parameter Δ to define the working-set window. The idea is to examine the most recent Δ page references. The set of pages in Δ is the working set *WS*.

Δ / ***WS*** **correlation:** the larger the window size, the larger is the working set.

$$WS(t, \Delta + 1) \supseteq WS(t, \Delta)$$

Lower bound / upper bound: a working set can use only a single page but can also grow as large as the number of pages *N*. It cannot exceed Δ.

$$1 \leq WS(t, \Delta) \leq \min(\Delta, N)$$

Working-Set Size (WSS): if we compute the working-set size WSS_i for each process in the system, we can then consider that

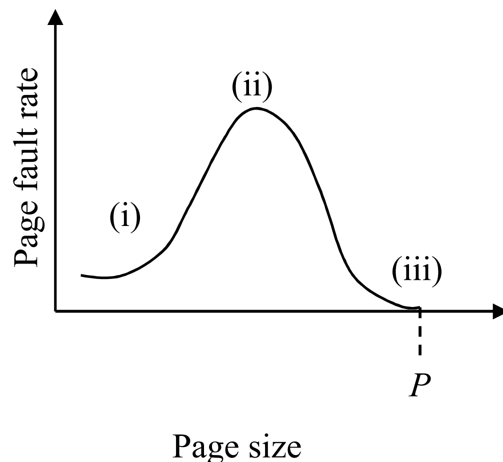
$$D = \sum_{\forall i} WSS_i$$

where *D* is the total demand for frames. If the total demand is greater than the total number of available frames said *m*, then $D > m$ and thrashing will occur.

Performance issues
"Page fault rate" (1)

Page fault rate characterizes the performance of a page replacement algorithm. Measuring the page fault rate is straightforward: just count the number of faults per second, possibly taking a running mean over past seconds as well.

There is an effect of page size on the fault rate at which page faults occurs.



(i) If the page size is small, a large number of pages will be available in main memory. After a time, the pages in memory will all contain portions of the process near recent references and the page fault rate should be low.

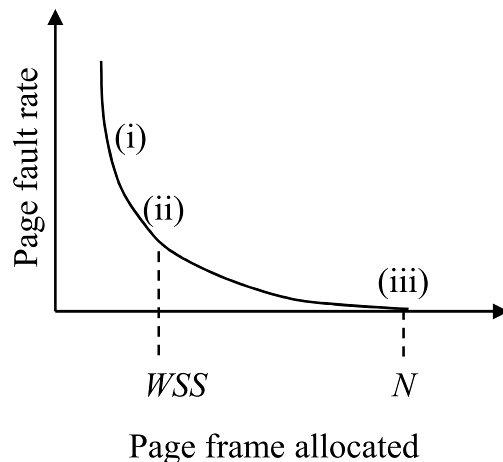
(ii) As the size increases, each individual page will contain locations further and further from any particular recent reference. Thus the effect of the principle of locality is weakened and the page fault rate begins to rise.

(iii) The page fault rate will begin to fall as the size of a page approaches the size of the entire process *P*.

Performance issues
"Page fault rate" (2)

Page fault rate characterizes the performance of a page replacement algorithm. Measuring the page fault rate is straightforward: just count the number of faults per second, possibly taking a running mean over past seconds as well.

The page fault rate is also determined by the number of frames allocated to a process.



(i) For a fixed page size, the fault rate drops as the number of pages required in main memory grows.

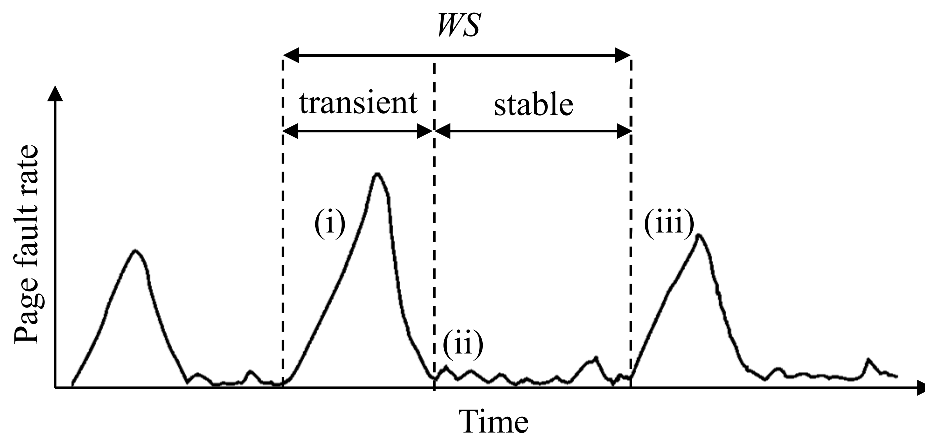
(ii) The parameter *WSS* represents the working set size and characterizes the performance behavior of the process memory image.

(iii) When the page frame allocated reaches the total number of pages in process *N*, the page fault rate will be zero off course.

Performance issues
"Page fault rate" (3)

Page fault rate characterizes the performance of a page replacement algorithm. Measuring the page fault rate is straightforward: just count the number of faults per second, possibly taking a running mean over past seconds as well.

There is a direct relationship between the *WS* of a process and its page-fault rate. Assuming there is sufficient memory to store the *WS*, the page-fault rate of the process will transit between transient and stable periods over the time.



(i) The transient periods reflect a shift of the program to a new locality. During the transition phase, some of the pages from the old locality remain within the window Δ causing a surge in the size of *WS*.

(ii) Once the working set of this new locality is in memory, the page-fault rate falls.

(iii) When the process moves to a new locality, the working set increases and the page-fault rate rises toward a peak once again.

Operating Systems
“Virtual Memory”

1. Introduction
2. Demand paging
3. Performance issues
4. Page table for large memories
5. Allocation of frames
6. Page replacement algorithms

Page table for large memories
"Introduction"

Page table structure: most of the modern computer systems support a large logical address space (e.g. 2^{64}), and for performances the paging system targets small page size (i.e. typically the page number (p) >> page offset (d)). In such an environment, the page table itself becomes excessively large.

e.g. a system with a 64 bits logical address space and a page size of 2^{12} = 4 KB, the page table may consists of up 4500 billions of entry 2^{52}. Assuming that each entry consists of 8 bytes, it will require $2^{52} \times 8$ bytes = 32 PB for storage.

Clearly, we would not want to allocate the page table contiguously in main memory. Different approaches can be handled to solve the problem.

Page table for large memories
"Hierarchical paging" (1)

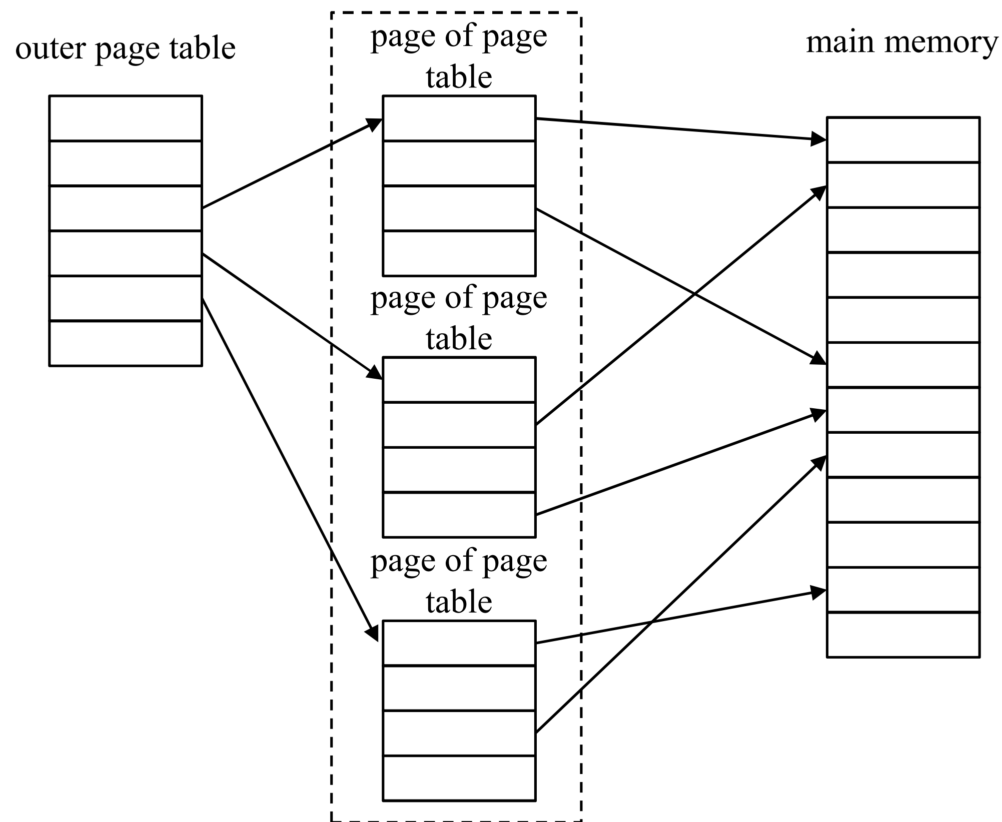
Hierarchical paging: one way is to use a two-level paging algorithm, in which the page table itself is also paged. The page number is further dived into an **index of the outer page (p_1)**, the **displacement within the page of outer page table (p_2)**, and the **offset (d)**. Typically, the maximum length of either a page table is restricted to be equal to one page (i.e. $p_1 \approx p_2 \approx d$).

index of the outer page (p_1)	displacement within the page of outer page table (p_2)	page offset (d)
p_1	p_2	d
m-n		*n*
m		

m	number of bits to encode the logical address
n	number of bit to encode the page offset
m-n	number of bits to encode the page number (index of outer page p_1 plus displacement p_2)

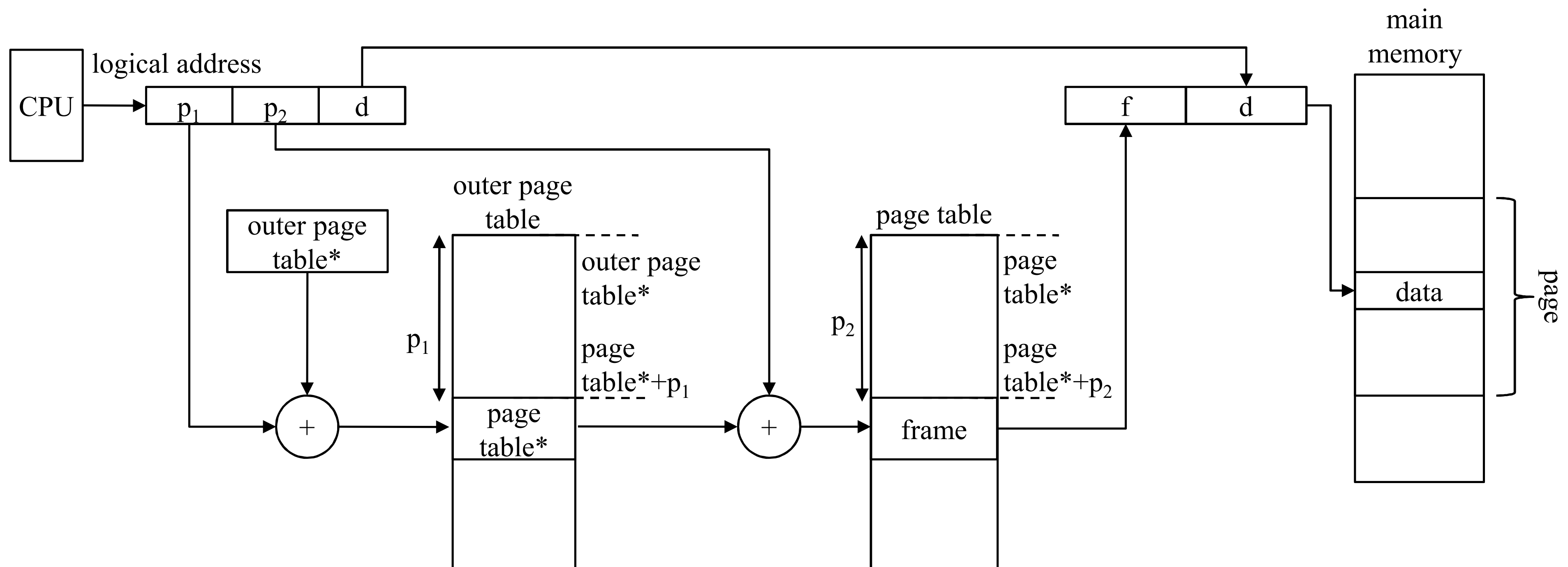
Page table for large memories "Hierarchical paging" (2)

Hierarchical paging: one way is to use a two-level paging algorithm, in which the page table itself is also paged. The page number is further dived into an **index of the outer page (p_1)**, the **displacement within the page of outer page table (p_2)**, and the **offset (d)**. Typically, the maximum length of either a page table is restricted to be equal to one page (i.e. $p_1 \approx p_2 \approx d$).



Page table for large memories
"Hierarchical paging" (3)

Hierarchical paging: one way is to use a two-level paging algorithm, in which the page table itself is also paged. The page number is further dived into an **index of the outer page (p_1)**, the **displacement within the page of outer page table (p_2)**, and the **offset (d)**. Typically, the maximum length of either a page table is restricted to be equal to one page (i.e. $p_1 \approx p_2 \approx d$).



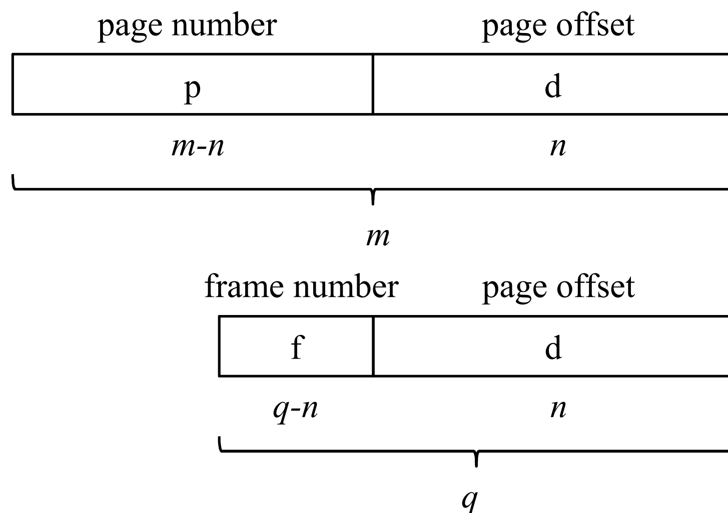
Page table for large memories
“Inverted page table” (1)

Inverted page table: with hierarchical paging the page table size is proportional to the logical address space. An alternative is to use an inverted page table. The page table structure is called inverted because its indexes page table entries by frame numbers rather by page number.

• Each logical address in the system consists of a triple < *process-id, page number, offset* >.

• There is only one inverted page table in the system, and it has only one entry for each page of the physical memory. For a physical memory of 2^m frames, the inverted page table contains 2^m entries.

• The inverted page table is sorted by physical address. Each inverted page table entry is a pair < *process-id, frame number* > where the *process-id* assumes the role of the address space identifier.



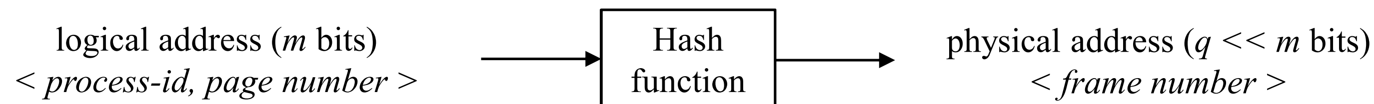
m	number of bits to encode the logical address
n	number of bits to encode the page offset
$m-n$	number of bits to encode the page number
q	number of bits to encode the physical address with $q << m$
$q-n$	number of bits to encode the frame number

Page table for large memories
“Inverted page table” (2)

Inverted page table: with hierarchical paging the page table size is proportional to the logical address space. An alternative is to use an inverted page table. The page table structure is called inverted because its indexes page table entries by frame numbers rather by page number.

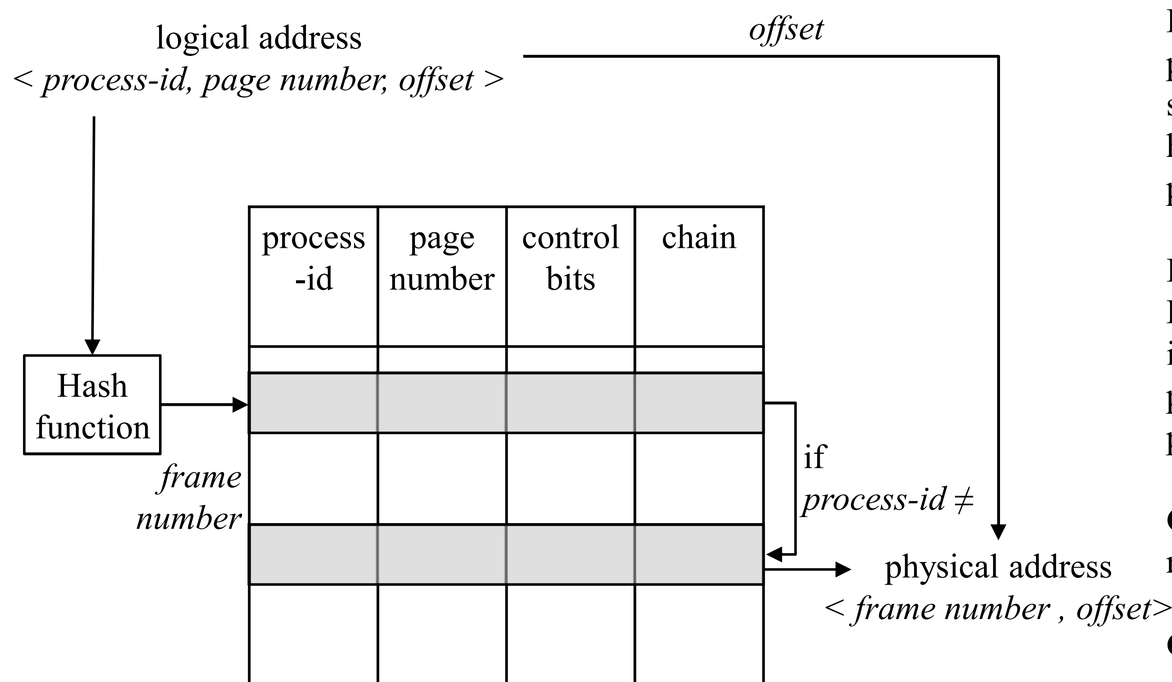
• When a memory reference occurs, the inverted page table is then searched for a match. This search would take far too long, to alleviate this problem a hash function is used.

• Because more than one logical address may map into the same hash table entry, a chaining technique is used for managing the overflow. The hashing technique results in chains that are typically short (e.g. one to two entries).



Page table for large memories
“Inverted page table” (3)

Inverted page table: with hierarchical paging the page table size is proportional to the logical address space. An alternative is to use an inverted page table. The page table structure is called inverted because its indexes page table entries by frame numbers rather by page number.



Hash function: the inverted page table is sorted by physical address, the whole table might need to be searched for a match. To alleviate this problem a hash function is used to return an hash value that points to an invert page table entry.

Process-id refers to the process that owns this page. Because more than one logical address may map into the same hash table entry, the combination of page number with process identifier identifies a page within the logical address space.

Control bits: this field includes flags such as valid, referenced and modified, protection and locking.

Chain pointer: is null is there are no chained entries. Otherwise, it contains a pointer to the next entry in the table to deal with the hash function overflow.

Operating Systems
“Virtual Memory”

1. Introduction
2. Demand paging
3. Performance issues
4. Page table for large memories
5. Allocation of frames
6. Page replacement algorithms

Allocation of frames (1)

Frame-allocation: if we have multiple processes in memory, we must decide how many frames to allocate to each process. This is the frame-allocation problem.

Max / min frame-allocation: considering the allocated frame number a_i to a process i we have:

$$\min < a_i < \max = m$$

- We cannot allocate more than the total number of available frames m in memory, off course.

- We must also allocate at least a minimum number of frames:

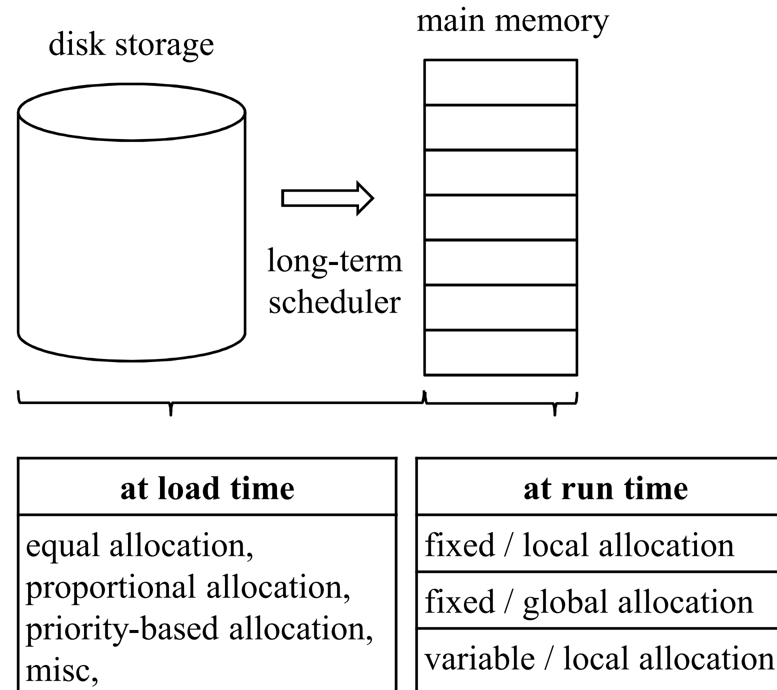
  - for performance, when the page frame allocated reaches the working set we are close from optimal performances, the min is fixed as heuristic in a system.

  - due to computer architecture e.g. the move PDP-11 instruction refers two frames, the MVC IBM 370 instruction would require references on eight frames, etc.

Allocation of frames (2)

Frame-allocation: if we have multiple processes in memory, we must decide how many frames to allocate to each process. This is the frame-allocation problem.

Allocation algorithms: whereas the minimum / maximum numbers of frames per process depend of the computer architecture and performances, in between we are still left with significant choice in frame allocation. This is managed by the frame allocation algorithms that can be driven at the load and run times.



Allocation of frames (3)

at load time	at run time
equal allocation, proportional allocation, priority-based allocation, misc,	fixed / local allocation
	fixed / global allocation
	variable / local allocation

Frame-allocation at load time can be fixed from memory requests or other criteria.

Equal allocation: the easiest way to split m frames among n processes is to give everyone an equal share, m/n frames.

Proportional allocation: let the size of the virtual memory for process p_i be s_i , and define

$$S = \sum_{\forall i} s_i$$

where S is total virtual memory size, we allocate a_i frames to process p_i, where a_i is approximately

$$a_i = \frac{s_i}{S} \times m$$

Of course, we must adjust a_i to be an integer and to fit in the [min, max] range.

Priority-based allocation: with either equal or proportional allocation, a high-priority process is treated the same as a low-priority process. A solution is to use a proportional allocation scheme wherein the ratio of frames depends on the priorities or on a combination of size and priority.

Misc: a number of algorithms and criterion can be applied for frame-allocation.

Allocation of frames (4)

at load time	at run time
equal allocation, proportional allocation, priority-based allocation, misc,	fixed / local allocation fixed / global allocation variable / local allocation

Frame-allocation at run time depends of the fixed/variable policies under the local/global scopes.

Fixed-allocation policy gives a process a fixed number of frames in main memory within which to execute.

Variable-allocation policy allows the number of frames allocated to a process to be varied over the lifetime of the process.

Local replacement requires that each process selects from only its own set of allocated frames. Local replacement might hinder a process, however, by not making available to it other, less used pages of memory.

Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process. That is, one process can take a frame from another.

These approaches can be designed.

	Local replacement	**Global replacement**
Fixed-allocation	worst performance	not possible
Variable-allocation	the best strategy, the key elements are the criteria to determine resident set size and the timing of changes	processes can suffer the reduction

Allocation of frames (5)

at load time	at run time
equal allocation, proportional allocation, priority-based allocation, misc,	fixed / local allocation fixed / global allocation variable / local allocation

Frame-allocation at run time depends of the fixed/variable policies under the local/global scopes.

Fixed-allocation / local scope: we have a process that is running in main memory with a fixed number of frames. When a page fault occurs, the OS must choose which page from among the currently resident pages for this process is to be replaced.

The drawbacks of this approach are listed below.

Small allocations: if allocations tend to be too small, there will be a high page fault rate, causing the entire multiprogramming system to run slowly.

Large allocations: if allocations tend to be unnecessarily large, there will be too few programs in main memory and there will be either considerable processor idle time or considerable time spent in swapping.

Allocation of frames (6)

at load time	at run time
equal allocation, proportional allocation, priority-based allocation, misc,	fixed / local allocation fixed / global allocation variable / local allocation

Frame-allocation at run time depends of the fixed/variable policies under the local/global scopes.

Variable allocation / global scope: there are a number of processes in main memory, each with a certain number of frames allocated to it. The OS also maintains a list of free frames. When a page fault occurs, a free frame is added to the resident set of a process and the page is brought in. Thus, a process experiencing page faults will gradually grow in size.

The drawback of this approach is given here.

Page replacement: when there are no free frames available, the OS must choose a page currently in memory to replace. The page selected for replacement can belong to any of the resident processes; there is no discipline to determine which process should lose a page from its resident set. Therefore, the process that suffers the reduction in resident set size may not be optimum.

Allocation of frames (7)

at load time	at run time
equal allocation, proportional allocation, priority-based allocation, misc,	fixed / local allocation fixed / global allocation variable / local allocation

Frame-allocation at run time depends of the fixed/variable policies under the local/global scopes.

Variable allocation / local scope attempts to overcome the problems with a global-scope strategy. When a page fault occurs, we select the page to replace from among the resident set of the process that suffers the fault. From time to time, it reevaluates the allocation provided to the process, and increases or decreases it to improve overall performance. The key elements are the criteria used to determine resident set size and the timing of changes. The strategy is twofold.

Working set strategy monitors the working set of each process. Periodically, it removes from the resident set of a process those pages that are not in its working set. A process may execute only if its working set is in main memory.

The drawbacks of this approach are listed below.

***WS* prediction:** the past does not always predict the future. Both the size and the membership of the working set will change over time.

***WS* measurement:** a true measurement of working set for each process is impractical. It would be necessary to time-stamp every page reference for every process using the virtual time of that process and then maintain a time-ordered queue of pages for each process.

Optimal Δ value: the optimal value of Δ is unknown and in any case would vary from process to process.

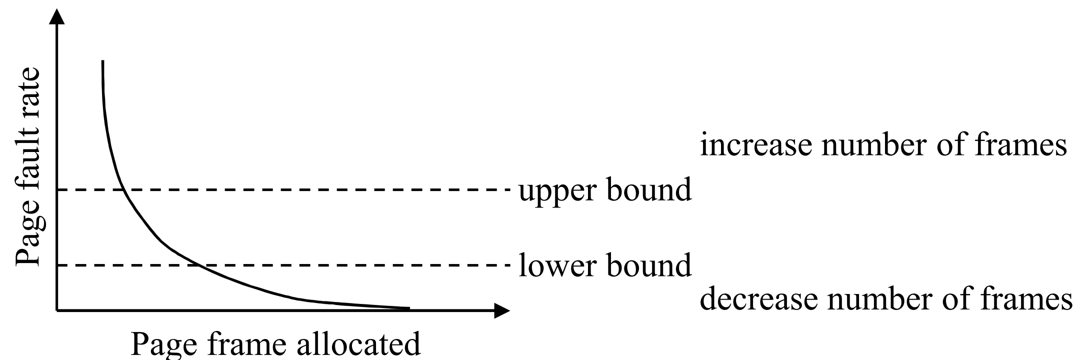
Allocation of frames (8)

at load time	at run time
equal allocation, proportional allocation, priority-based allocation, misc,	fixed / local allocation
	fixed / global allocation
	variable / local allocation

Frame-allocation at run time depends of the fixed/variable policies under the local/global scopes.

Variable allocation / local scope attempts to overcome the problems with a global-scope strategy. When a page fault occurs, we select the page to replace from among the resident set of the process that suffers the fault. From time to time, it reevaluates the allocation provided to the process, and increases or decreases it to improve overall performance. The key elements are the criteria used to determine resident set size and the timing of changes. The strategy is twofold.

Page Fault Frequency (PFF) wants to control the page-fault rate. If the actual page-fault rate exceeds the upper limit, we allocate the process another frame; if the page-fault rate falls below the lower limit, we remove a frame from the process. When a page fault occurs, the OS notes the virtual time since the last page fault for that process for PFF update.



Allocation of frames (9)

at load time
equal allocation, proportional allocation, priority-based allocation, misc,

at run time
fixed / local allocation
fixed / global allocation
variable / local allocation

Frame-allocation at run time depends of the fixed/variable policies under the local/global scopes.

Variable allocation / local scope attempts to overcome the problems with a global-scope strategy. When a page fault occurs, we select the page to replace from among the resident set of the process that suffers the fault. From time to time, it reevaluates the allocation provided to the process, and increases or decreases it to improve overall performance. The key elements are the criteria used to determine resident set size and the timing of changes. The strategy is twofold.

Page Fault Frequency (PFF) wants to control the page-fault rate. If the actual page-fault rate exceeds the upper limit, we allocate the process another frame; if the page-fault rate falls below the lower limit, we remove a frame from the process. When a page fault occurs, the OS notes the virtual time since the last page fault for that process for PFF update.

The drawback to this approach is given here.

Transient period: during inter-locality transitions, the rapid succession of page faults causes the resident set of a process to swell before the pages of the old locality are expelled; the sudden peaks of memory demand may produce unnecessary process deactivations and reactivations, with the corresponding undesirable switching and swapping overheads.

Operating Systems
"Virtual Memory"

1. Introduction
2. Demand paging
3. Performance issues
4. Page table for large memories
5. Allocation of frames
6. Page replacement algorithms

Page replacement algorithms
“Introduction”

Page replacement: when a page fault occurs, the OS has to choose a page to remove to make room for the incoming page. While it would be possible to pickup a random page to evict at each page fault, system performance is much better if a page that is not heavily used is chosen. This is the problem of page replacement and a number of algorithm has been proposed.

Algorithm	**Performances**	**Hardware support**	**Comment**
Optimal	+++	Na	not implementable, but useful as a benchmark
LRU (Least Recently Used)	++	high	excellent, but difficult to implement exactly
FIFO (First-In, First-Out)	-	no	might throw out important pages
Clock Page Replacement	+	intermediate	good improvement over FIFO

Algorithm	**Performances**	**Hardware support**	**Comment**
Optimal	+++	Na	not implementable, but useful as a benchmark
LRU (Least Recently Used)	++	high	excellent, but difficult to implement exactly
FIFO (First-In, First-Out)	-	no	might throw out important pages
Clock Page Replacement	+	intermediate	good improvement over FIFO

Page replacement algorithms
“Optimal page replacement (OPT)”

Optimal page replacement (OPT) selects for replacement a page for which the time to the next reference is the longest. It can be shown that this policy results in the fewest number of page faults. Clearly, this policy is impossible to implement, because it would require the OS to have perfect knowledge of future events. However, it does serve as a standard against which to judge real-world algorithms.

time t	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Page address stream	2	3	2	1	5	2	4	5	3	2	5	2
Page frames	2	2	2	2	2	2	4	4	4	2	2	2
		3	3	3	3	3	3	3	3	3	3	3
				1	5	5	5	5	5	5	5	5
Page fault	×	×		×	F		F			F		

(a) the page 1 is replaced with 5, it is the longest reference neither requited then t=∞

(b) the page 2 is replaced with 4, it is the longest reference requited at t=9

(c) the pages 3 and 4 are the longest references neither required t=∞, the system selects one of them to be replaced with the page 2

Algorithm	Performances	Hardware support	Comment
Optimal	+++	Na	not implementable, but useful as a benchmark
LRU (Least Recently Used)	++	high	excellent, but difficult to implement exactly
FIFO (First-In, First-Out)	-	no	might throw out important pages
Clock Page Replacement	+	intermediate	good improvement over FIFO

Page replacement algorithms
"Least Recently Used (LRU)" (1)

Least Recently Used (LRU) policy replaces the page in memory that has not been referenced for the longest time. By the principle of locality, this should be the page least likely to be referenced in the near future. And, in fact, the LRU policy does nearly as well as the optimal policy.

time t	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Page address stream	2	3	2	1	5	2	4	5	3	2	5	2
	2	2	2	2	2	2	2	2	3	3	3	3
Page frames		3	3	3	5	5	5	5	5	5	5	5
				1	1	1	4	4	4	2	2	2
Page fault	×	×		×	F		F		F	F		

(a) the page 3 is replaced with 5, it has been referenced at t=1 vs. 2 at t=2 and 1 at t=3

(b) the page 1 is replaced with 4, it has been referenced at t=3

(c) the page 2 is replaced with 3, it has been referenced at t=5

(d) the page 4 is replaced with 2, it has been referenced at t=6

Rq. if S is a page address stream and S^R its reverse, then the page fault rate PFR is PFR(OPT, S) = PFR(LRU, S^R).

Page replacement algorithms
"Least Recently Used (LRU)" (2)

Least Recently Used (LRU) policy replaces the page in memory that has not been referenced for the longest time. By the principle of locality, this should be the page least likely to be referenced in the near future. And, in fact, the LRU policy does nearly as well as the optimal policy.

Implementation of LRU: an LRU page-replacement algorithm requires substantial hardware assistance to determine an order for the frames defined by the time of last use. Two implementations are feasible.

Counter We associate with each page-table entry a time-of-use field and a clock is incremented for every memory reference. Whenever a reference to a page is made, the clock register is copied to the time-of-use field.

We replace the page with the smallest time value, this requires a search of the page table to find the LRU page with the lowest clock value.

This scheme requires to record the time-of-use field for each memory access.

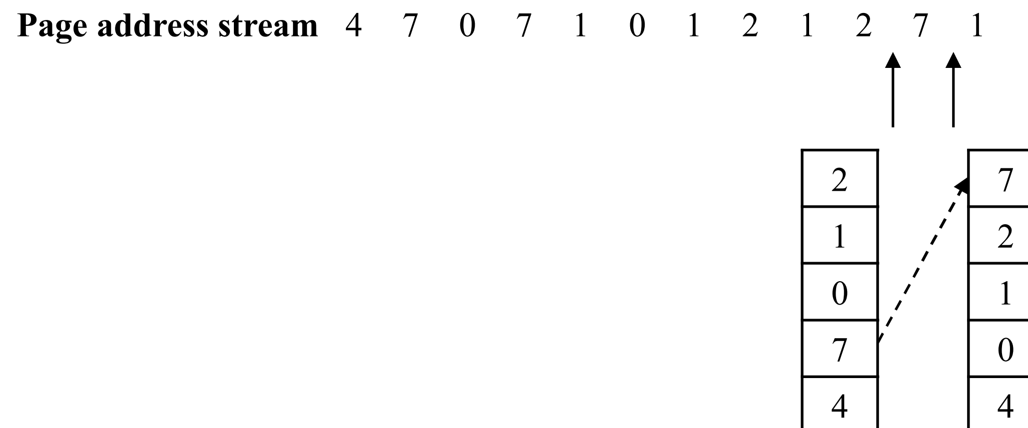
Page replacement algorithms
“Least Recently Used (LRU)” (3)

Least Recently Used (LRU) policy replaces the page in memory that has not been referenced for the longest time. By the principle of locality, this should be the page least likely to be referenced in the near future. And, in fact, the LRU policy does nearly as well as the optimal policy.

Implementation of LRU: an LRU page-replacement algorithm requires substantial hardware assistance to determine an order for the frames defined by the time of last use. Two implementations are feasible.

Stack Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the most recently used page is always at the top of the stack and the least recently used page is always at the bottom.

Because the entries could be removed at any location in the stack, it is best to use a double linked list with a head pointer and a tail pointer. Updating a page requires changing six pointers. Each update is a little more expensive, but there is no search for a replacement as the tail pointer points the LRU page.



Algorithm	Performances	Hardware support	Comment
Optimal	+++	Na	not implementable, but useful as a benchmark
LRU (Least Recently Used)	++	high	excellent, but difficult to implement exactly
FIFO (First-In, First-Out)	-	no	might throw out important pages
Clock Page Replacement	+	intermediate	good improvement over FIFO

Page replacement algorithms
"First-In-First-Out (FIFO)" (1)

First-in-first-out (FIFO) policy treats the page frames allocated to a process as a circular buffer. The logic behind this choice is to replace the page that has been in memory the longest. This reasoning will often be wrong, because there will often be regions of program that are heavily used. Those pages will be repeatedly paged out and in by the FIFO algorithm.

time t	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Page address stream	2	3	2	1	5	2	4	5	3	2	5	2
Page frames	2	2	2	2	5	5	5	5	3	3	3	3
		3	3	3	3	2	2	2	2	2	5	5
				1	1	1	4	4	4	4	4	2
Page fault	×	×		×	F	F	F		F		F	F

(a) the page 2 is replaced with 5 in FIFO order

(b) the page 3 is replaced with 2 in FIFO order

(c) the page 1 is replaced with 4 in FIFO order

(d) the page 5 is replaced with 3 in FIFO order

(e) the page 2 is replaced with 5 in FIFO order

(f) the page 4 is replaced with 2 in FIFO order

Page replacement algorithms “First-In-First-Out (FIFO)” (2)

First-in-first-out (FIFO) policy treats the page frames allocated to a process as a circular buffer. The logic behind this choice is to replace the page that has been in memory the longest. This reasoning will often be wrong, because there will often be regions of program that are heavily used. Those pages will be repeatedly paged out and in by the FIFO algorithm.

Belady’s anomaly: a bad replacement policy and setting could increase the page-fault rate and slows the process execution. Some investigators noticed that for some page-replacement algorithms as FIFO, the page-fault rate may increase as the number of allocated frames increases. This is known as the Belady’s anomaly.

time t	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Page address stream	1	2	3	4	1	2	5	1	2	3	4	5
Page frames	1	1	1	4	4	4	5	5	5	5	5	5
		2	2	2	1	1	1	1	1	3	3	3
			3	3	3	2	2	2	2	2	4	4
Page fault	F	F	F	F	F	F	F			F	F	

With 3 frames we have 9 faults F.

Page replacement algorithms
"First-In-First-Out (FIFO)" (3)

First-in-first-out (FIFO) policy treats the page frames allocated to a process as a circular buffer. The logic behind this choice is to replace the page that has been in memory the longest. This reasoning will often be wrong, because there will often be regions of program that are heavily used. Those pages will be repeatedly paged out and in by the FIFO algorithm.

Belady's anomaly: a bad replacement policy and setting could increase the page-fault rate and slows the process execution. Some investigators noticed that for some page-replacement algorithms as FIFO, the page-fault rate may increase as the number of allocated frames increases. This is known as the Belady's anomaly.

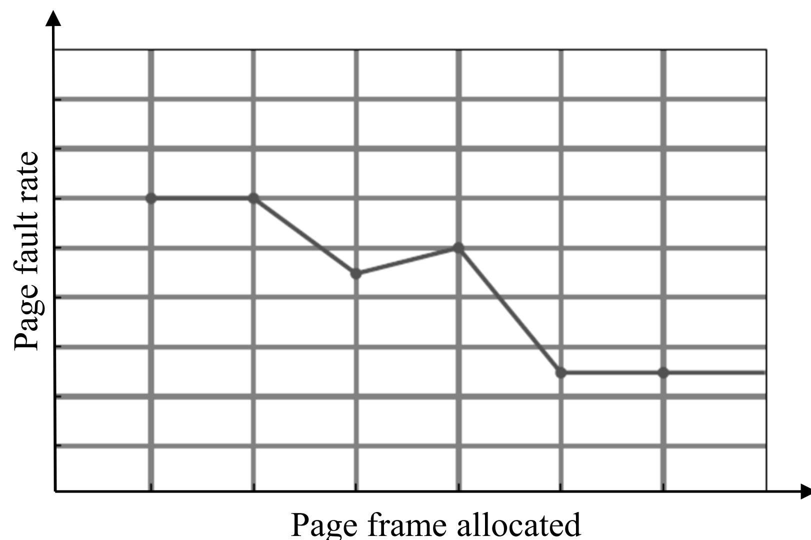
time t	0	1	2	3	4	5	6	7	8	9	10	11
Page address stream	1	2	3	4	1	2	5	1	2	3	4	5
Page frames	1	1	1	1	1	1	5	5	5	5	4	4
		2	2	2	2	2	2	1	1	1	1	5
			3	3	3	3	3	3	2	2	2	2
				4	4	4	4	4	4	3	3	3
Page fault	F	F	F	F			F	F	F	F	F	F

With 4 frames we have 10 faults F, the number of faults for four frames 10 is greater than the number of faults for three frames 9.

Page replacement algorithms
“First-In-First-Out (FIFO)” (4)

First-in-first-out (FIFO) policy treats the page frames allocated to a process as a circular buffer. The logic behind this choice is to replace the page that has been in memory the longest. This reasoning will often be wrong, because there will often be regions of program that are heavily used. Those pages will be repeatedly paged out and in by the FIFO algorithm.

Belady’s anomaly: a bad replacement policy and setting could increase the page-fault rate and slows the process execution. Some investigators noticed that for some page-replacement algorithms as FIFO, the page-fault rate may increase as the number of allocated frames increases. This is known as the Belady’s anomaly.



The Belady’s anomaly characterizes a page fault rate curve that is not monotonically decreasing for the algorithm.

Algorithm	Performances	Hardware support	Comment
Optimal	+++	Na	not implementable, but useful as a benchmark
LRU (Least Recently Used)	++	high	excellent, but difficult to implement exactly
FIFO (First-In, First-Out)	-	no	might throw out important pages
Clock Page Replacement	+	intermediate	good improvement over FIFO

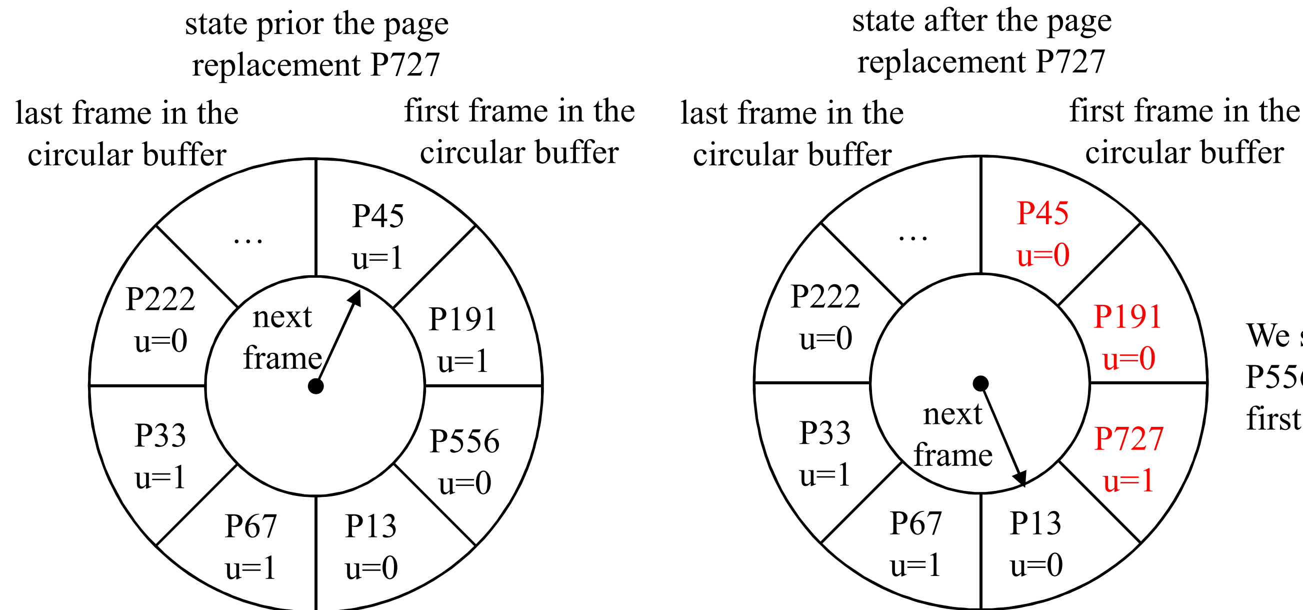
Page replacement algorithms
"Clock Page Replacement" (1)

Clock page replacement (simplest form): requires to employ the use bit (u). During the scanning, each time it encounters a frame with u=1, it resets that bit and continues on. The first frame encountered with u = 0 is chosen for replacement. If all of the frames have u =1 then the pointer will make one complete cycle through the buffer and stop at its original position.

u = 1 when a page is first loaded into a frame in memory or whenever a page is subsequently referenced.

u = 0 when it comes time to replace a page, the OS scans the buffer to find a frame with u=0. Each time it encounters a frame with a use bit of 1, it resets that bit and continues on.

e.g



We set u=0 for P45, P191 when scanning, P556 is replaced with P727 as it is the first frame encountered with u=0,

Page replacement algorithms
"Clock Page Replacement" (2)

Clock page replacement (simplest form): requires to employ the use bit (u). During the scanning, each time it encounters a frame with u=1, it resets that bit and continues on. The first frame encountered with u = 0 is chosen for replacement. If all of the frames have u =1 then the pointer will make one complete cycle through the buffer and stop at its original position.

* indicates u=1, → refers to the stack position

time t	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Page address stream	2	3	2	1	5	2	4	5	3	2	5	2
Page frames	2*	2*	2*	→ 2*	5*	5*	→ 5*	→ 5*	3*	3*	→ 3*	→ 3*
	→	3*	3*	3*	→ 3	2*	2*	2*	→ 2	→ 2*	2	2*
		→	→	1*	1	→ 1	4*	4*	4	4	5*	5*
Page fault	×	×		×	F	F	F		F		F	

(a) (b) one step in the buffer as frames are empty

(c) the page is already here

(d) one step in the buffer, the circular scan is completed and pointer goes back to the first frame position

(e) all u=1, the clock algorithm performs a complete cycle and puts all u=0, then stops at its original position, the page 2 is replaced

(f) (g) one step in the buffer as u=0 for the page 3 and 1

Page replacement algorithms
“Clock Page Replacement” (3)

Clock page replacement (simplest form): requires to employ the use bit (u). During the scanning, each time it encounters a frame with u=1, it resets that bit and continues on. The first frame encountered with u = 0 is chosen for replacement. If all of the frames have u =1 then the pointer will make one complete cycle through the buffer and stop at its original position.

* indicates u=1, → refers to the stack position

time t	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Page address stream	2	3	2	1	5	2	4	5	3	2	5	2
Page frames	2*	2*	2*	→ 2*	5*	5*	→ 5*	→ 5*	3*	3*	→ 3*	→ 3*
	→	3*	3*	3*	→ 3	2*	2*	2*	→ 2	→ 2*	2	2*
		→	→	1*	1	→ 1	4*	4*	4	4	5*	5*
Page fault	×	×		×	F	F	F		F		F	

the pages 4, 5 are already here (h) (i)

all u=1, the clock algorithm performs a complete cycle and puts all (j)
u=0, then stops at its original position, the page 5 is replaced

u is put to 1 for the page 2 recently used (k)

the clock algorithm goes to page 4 with u=0, we set u=0 for the page 2 when scanning (l)
the circular scan is completed and pointer goes back to the first frame position

u is put to 1 for the page 2 recently used (m)

Page replacement algorithms
"Clock Page Replacement" (4)

Clock page replacement (advanced form): is be made more powerful by combining the use (u) and the modified (m) bits. A page with m=0 is a good for replacement because it is unmodified, it does not need to be written back to memory.

The (u, m) vector

u \ m	0	1
0	(0, 0) not accessed recently, not modified	(0, 1) not accessed recently, modified
1	(1, 0) accessed recently, not modified	(1, 1) accessed recently, modified

With this classification, the clock algorithm performs as follows:

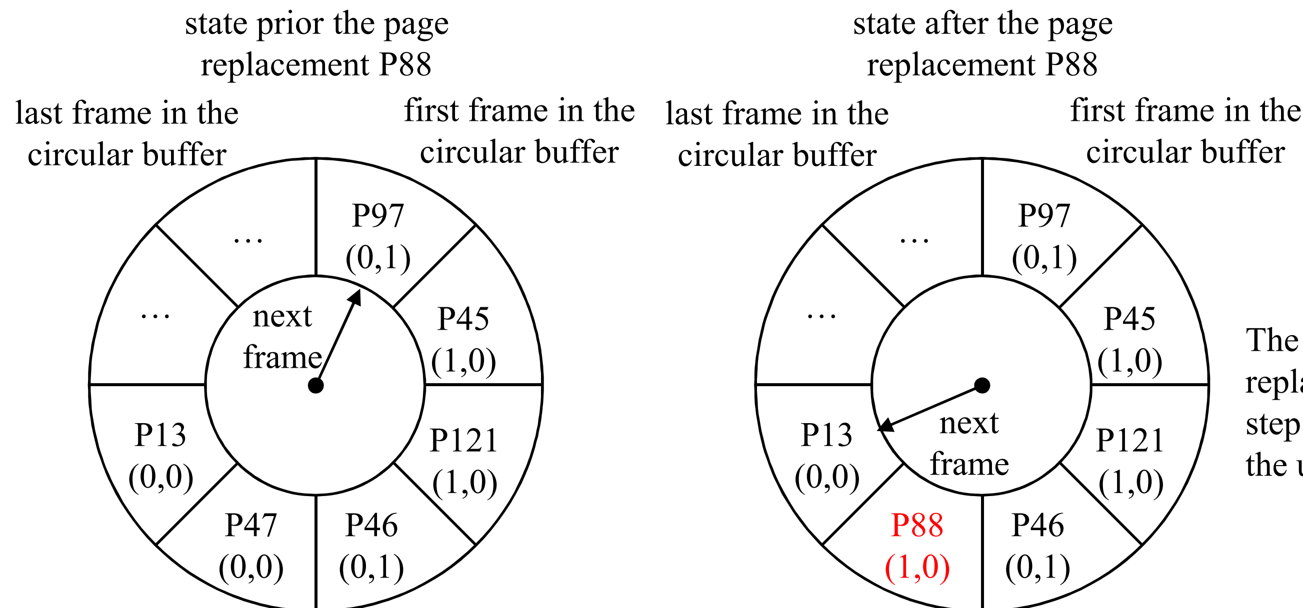
Step 1 during the scanning, the first frame encountered with (u = 0, m = 0) is chosen for replacement. During this scan, make no changes to the use bit.

Step 2 if step 1 fails, scan again and apply the simplest form of clock page replacement with (u = 0, m = 1).

Page replacement algorithms
"Clock Page Replacement" (5)

Clock page replacement (advanced form): is be made more powerful by combining the use (u) and the modified (m) bits. A page with m=0 is a good for replacement because it is unmodified, it does not need to be written back to memory.

e.g.



The clock algorithm goes to P47 for replacement with (u=0, m=0) through the step 1, during the scan no need to change the use bit for P97, P145, P121 and P46.