Systems and Networking – Unit I

B.Sc. in Applied Computer Science and Artificial Intelligence

2021-2022

Gabriele Tolomei

Department of Computer Science

Sapienza Università di Roma

tolomei@di.uniroma1.it

SAPIENZA UNIVERSITÀ DI ROMA

Paging + Segmentation

- Paging (OS' view of memory)
  - Divide memory into fixed-size pages and map them to physical frames
- Segmentation (compiler's view of memory)
  - Divide process into logical segments (e.g., code, data, stack, heap)
- Combine paging with segmentation to get the best of both worlds
  - Segmented Paging

Paging + Segmentation

- Paging (OS' view of memory)
  - Divide memory into fixed-size pages and map them to physical frames
- Segmentation (compiler's view of memory)
  - Divide process into logical segments (e.g., code, data, stack, heap)
- Combine paging with segmentation to get the best of both worlds
  - Segmented Paging

So far, the entire virtual address space of a process was assumed to fit and be all in memory

Virtual Memory

- In practice, most real processes do not need all their pages loaded in memory, or at least not all at once, e.g.,:

Virtual Memory

- In practice, most real processes do not need all their pages loaded in memory, or at least not all at once, e.g.,:
  - Error handling code is not needed unless that specific error occurs, some of which are quite rare
  - Arrays are often over-sized for worst-case scenarios, and only a small fraction of the arrays is actually used in practice
  - Some features of certain programs are rarely used

Virtual Memory

- In practice, most real processes do not need all their pages loaded in memory, or at least not all at once, e.g.,:
  - Error handling code is not needed unless that specific error occurs, some of which are quite rare
  - Arrays are often over-sized for worst-case scenarios, and only a small fraction of the arrays is actually used in practice
  - Some features of certain programs are rarely used

Virtual Memory uses backing storage (i.e., disk) to store unused pages and give the illusion of infinite virtual address space

Virtual Memory: Benefits

- The ability to load only the portions of processes that are actually needed (and only when needed) from disk has several benefits:

Virtual Memory: Benefits

- The ability to load only the portions of processes that are actually needed (and only when needed) from disk has several benefits:
  - Programs could be written for a much larger address space than physically exists on the computer

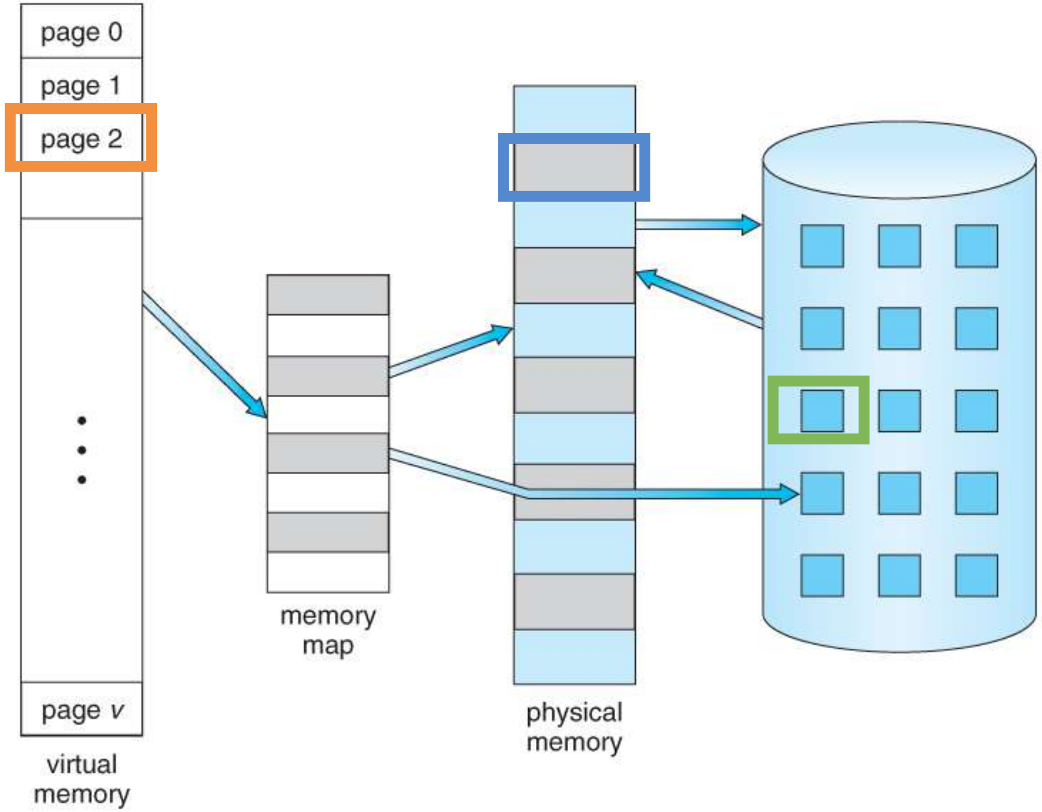
Virtual Memory: Benefits

- The ability to load only the portions of processes that are actually needed (and only when needed) from disk has several benefits:
  - Programs could be written for a much larger address space than physically exists on the computer
  - More memory is left for other programs, improving CPU utilization

Virtual Memory: Benefits

- The ability to load only the portions of processes that are actually needed (and only when needed) from disk has several benefits:
  - Programs could be written for a much larger address space than physically exists on the computer
  - More memory is left for other programs, improving CPU utilization
  - Less I/O is needed for swapping processes in and out of memory, speeding things up

Virtual Memory: The Big Picture



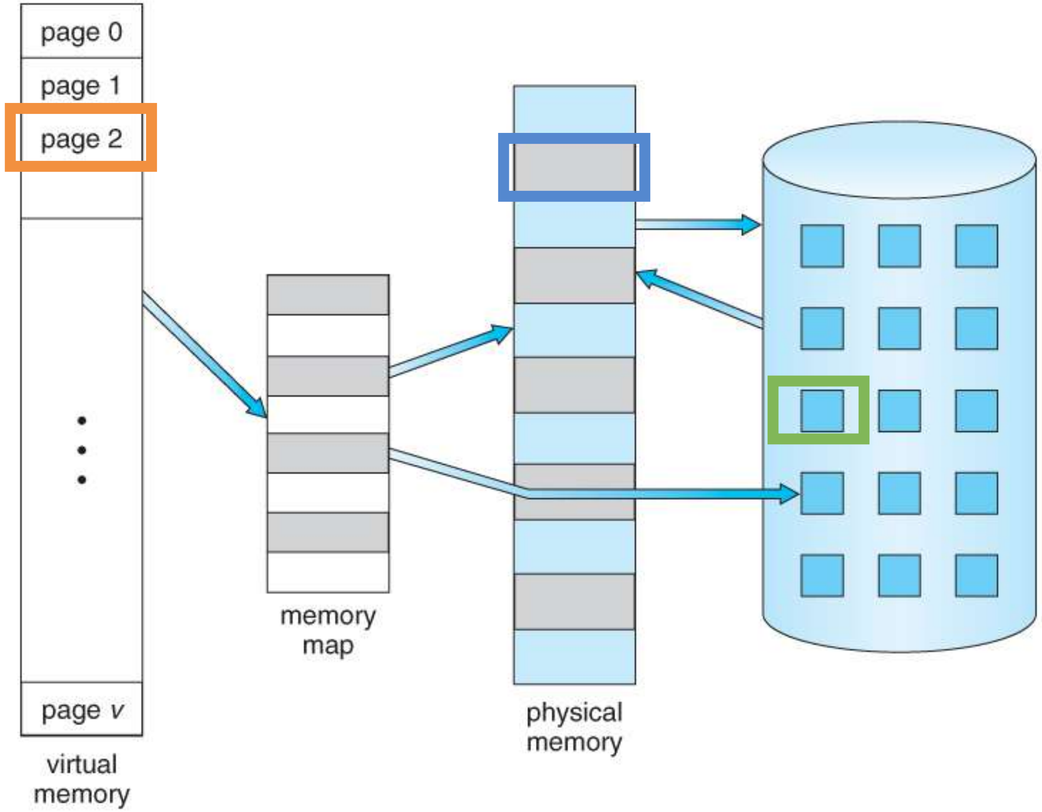
At any given time, each page can be:

in memory (physical frame)

on backing store (disk)

Virtual Memory: The Big Picture

virtual memory can be much larger than physical memory

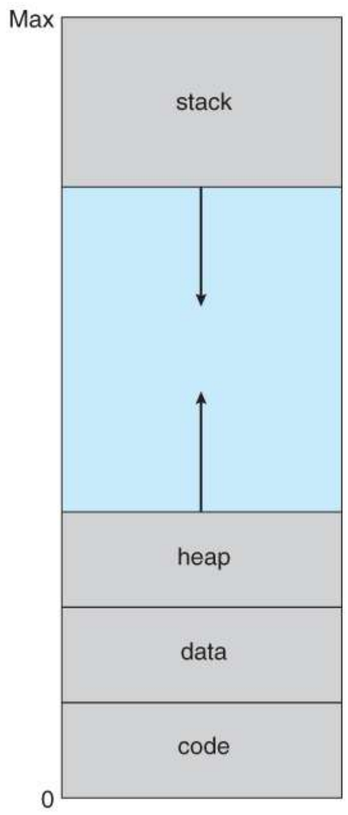


At any given time, each page can be:

in memory (physical frame)

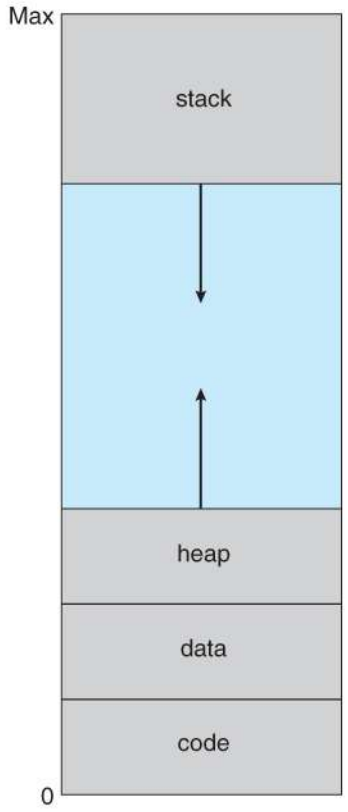
on backing store (disk)

The Sparseness of Virtual Address Space



Typically, virtual address space is highly sparse

The Sparseness of Virtual Address Space



Typically, virtual address space is highly sparse

A lot of virtual memory addresses remain unreferenced

Virtual Memory: Idea

- Use the memory as a cache for the disk

Virtual Memory: Idea

- Use the memory as a cache for the disk
- The page table must also indicate if the page is on disk or in memory (just using a single invalid bit)

Virtual Memory: Idea

- Use the memory as a cache for the disk
- The page table must also indicate if the page is on disk or in memory (just using a single invalid bit)
- Once the page is loaded from disk to memory, the OS updates the corresponding entry of the page table along with the valid bit

Virtual Memory: Idea

- Use the memory as a cache for the disk
- The page table must also indicate if the page is on disk or in memory (just using a single invalid bit)
- Once the page is loaded from disk to memory, the OS updates the corresponding entry of the page table along with the valid bit
- Remember: access to disk is extremely slower than access to memory

Virtual Memory: Idea

- Use the memory as a cache for the disk
- The page table must also indicate if the page is on disk or in memory (just using a single invalid bit)
- Once the page is loaded from disk to memory, the OS updates the corresponding entry of the page table along with the valid bit
- Remember: access to disk is extremely slower than access to memory
- Therefore, memory accesses must reference pages that are in memory with high probability

Virtual Memory: The Locality Assumption

- The 90÷10 rule claims that on a particular time frame, most of the memory references made by a process is around a small "area"

Virtual Memory: The Locality Assumption

- The 90÷10 rule claims that on a particular time frame, most of the memory references made by a process is around a small "area"
- We call this area as the **working set** of the process

Virtual Memory: The Locality Assumption

- The 90÷10 rule claims that on a particular time frame, most of the memory references made by a process is around a small "area"
- We call this area as the working set of the process
- Since the working set is fairly small compared to the whole virtual address space, it will likely fit in memory

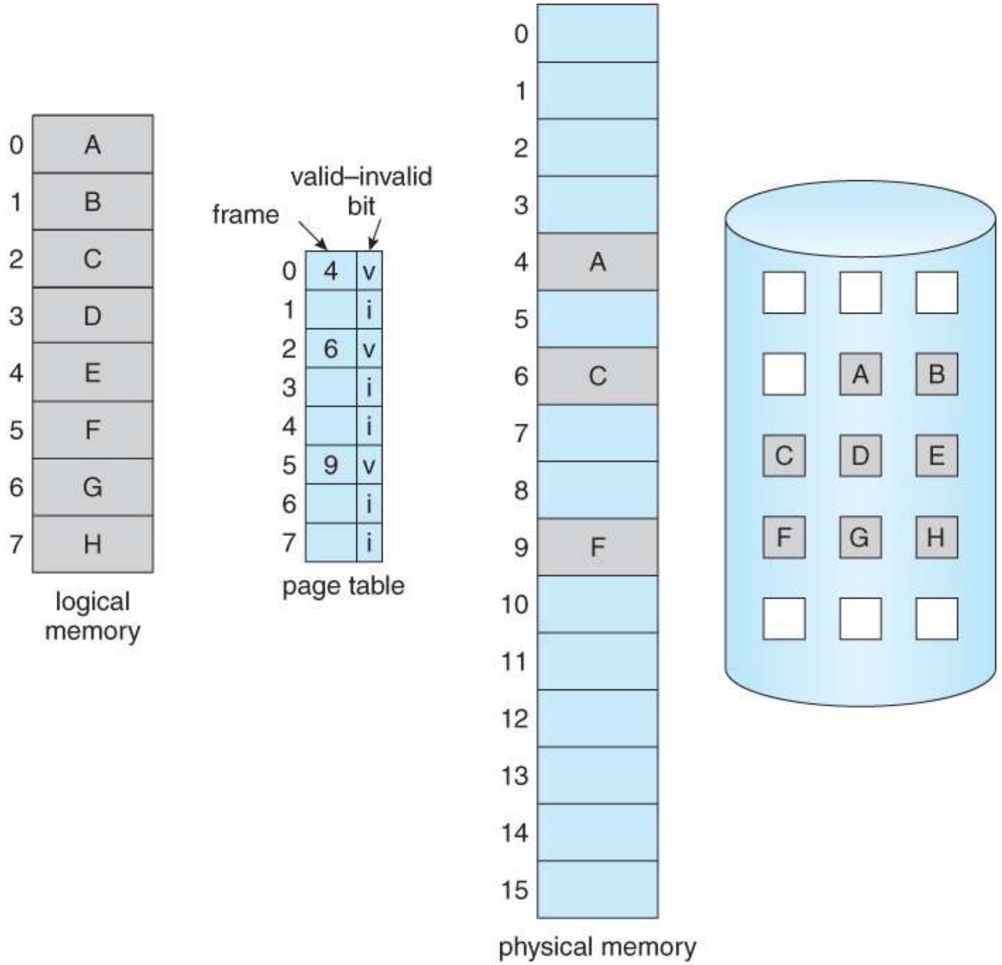
Virtual Memory: The Locality Assumption

- The 90÷10 rule claims that on a particular time frame, most of the memory references made by a process is around a small "area"
- We call this area as the working set of the process
- Since the working set is fairly small compared to the whole virtual address space, it will likely fit in memory
- Of course, during the lifetime of a process its working set may change (i.e., a process may eventually refer *all* of its virtual address space)

Virtual Memory: The Locality Assumption

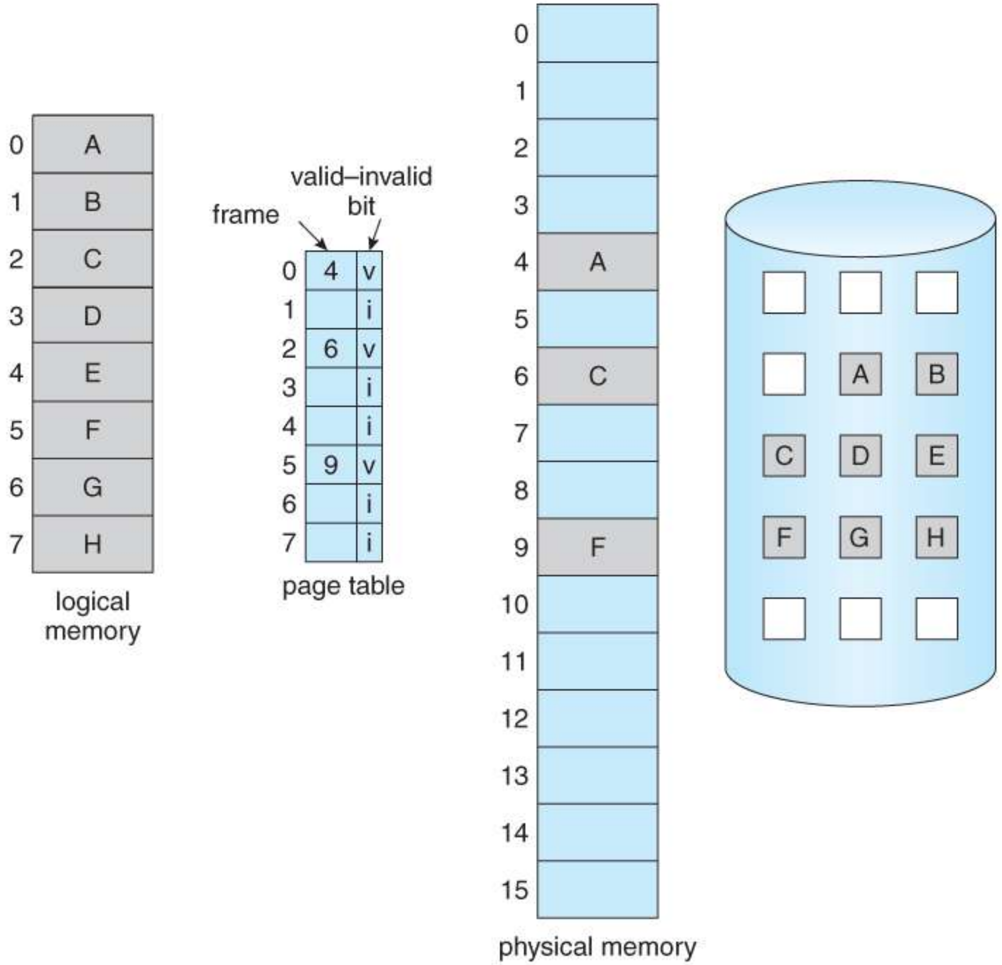
- The 90÷10 rule claims that on a particular time frame, most of the memory references made by a process is around a small "area"
- We call this area as the working set of the process
- Since the working set is fairly small compared to the whole virtual address space, it will likely fit in memory
- Of course, during the lifetime of a process its working set may change (i.e., a process may eventually refer *all* of its virtual address space)
- But in a reasonably small time frame, the working set stays the same

Virtual Memory: Basic Concepts



At each logical memory reference, a page table lookup is performed as usual

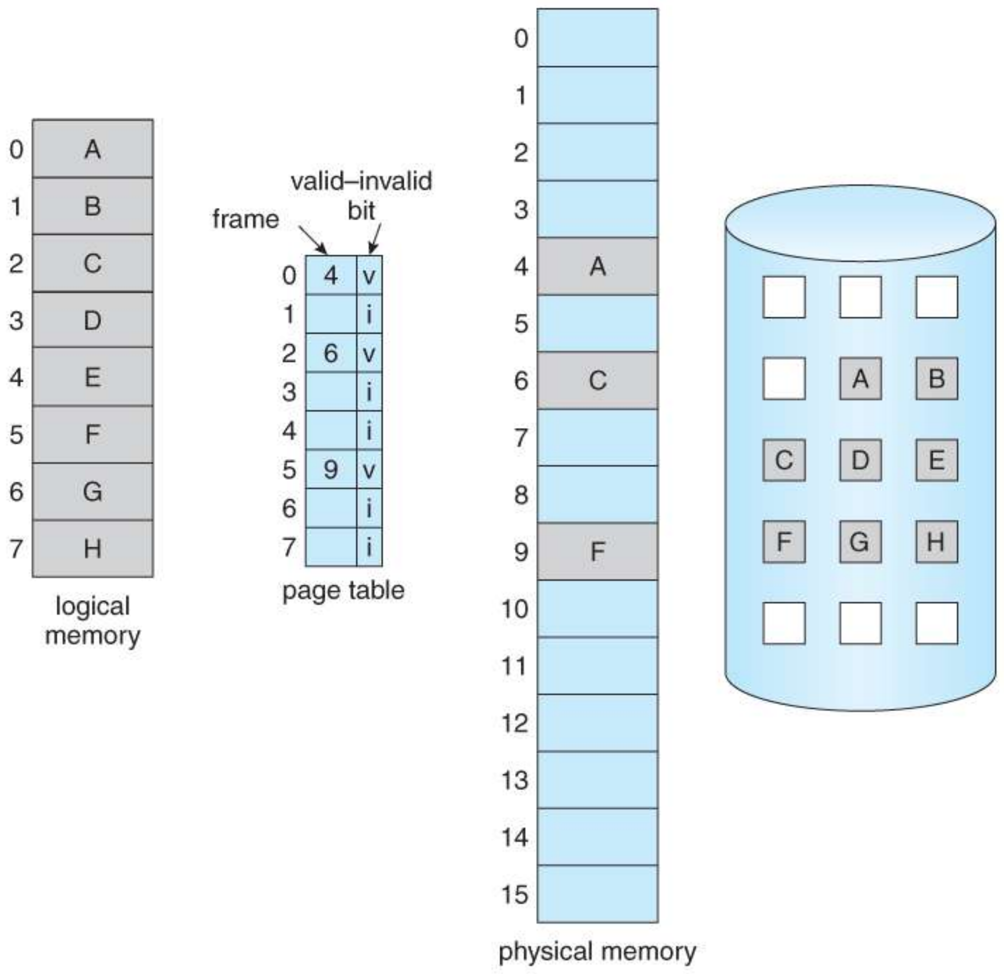
Virtual Memory: Basic Concepts



At each logical memory reference, a page table lookup is performed as usual

In the page table, the valid-invalid bit is checked for the corresponding entry

Virtual Memory: Basic Concepts

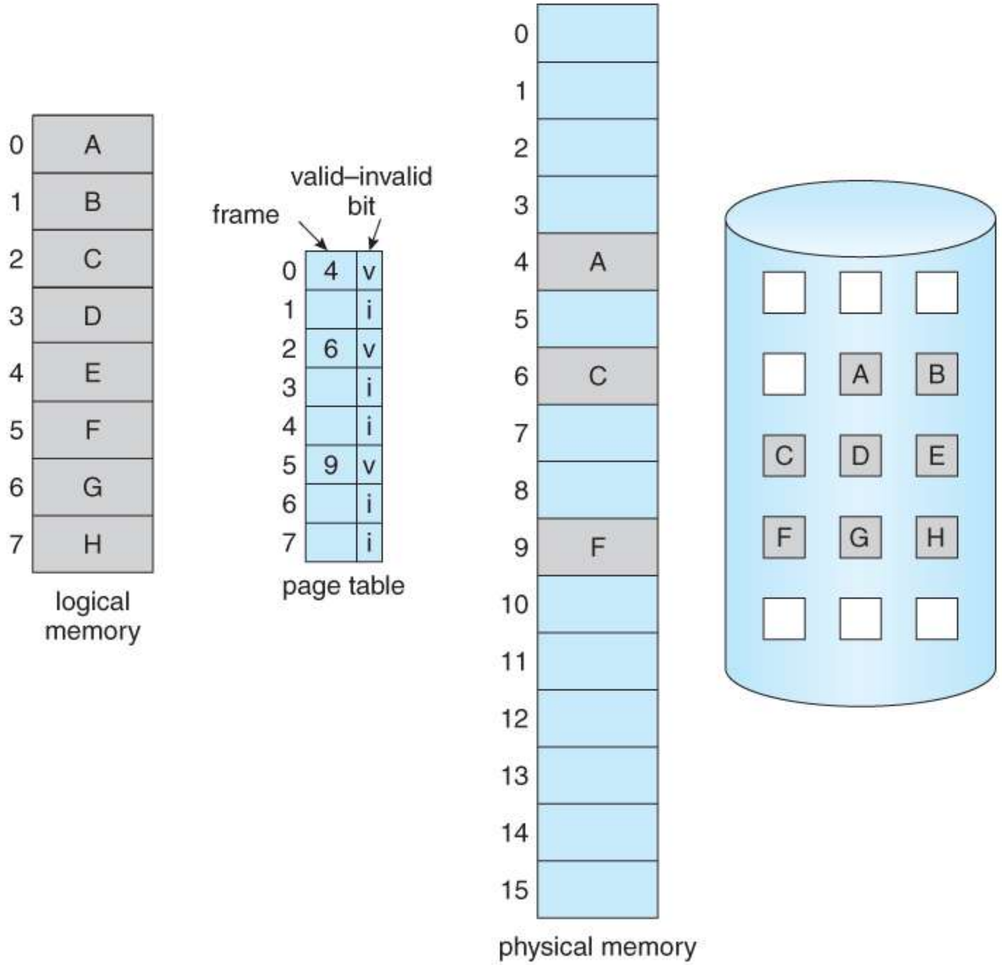


At each logical memory reference, a page table lookup is performed as usual

In the page table, the valid-invalid bit is checked for the corresponding entry

If the bit is set to 1 it means the page entry is valid (i.e., the requested page is in memory)

Virtual Memory: Basic Concepts



At each logical memory reference, a page table lookup is performed as usual

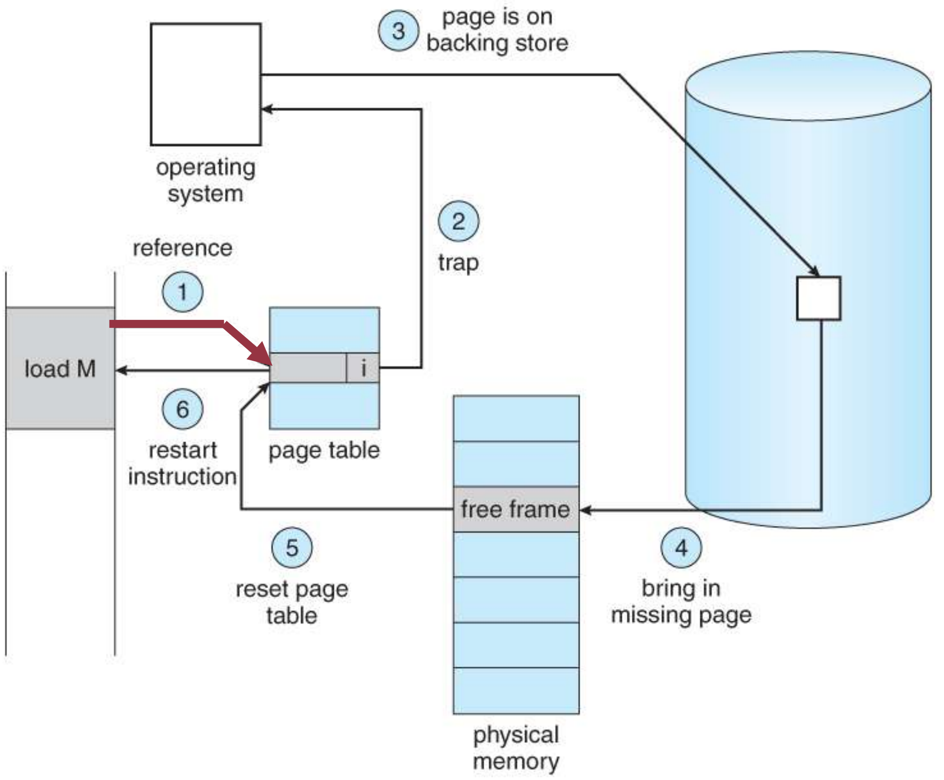
In the page table, the valid-invalid bit is checked for the corresponding entry

If the bit is set to 1 it means the page entry is valid (i.e., the requested page is in memory)

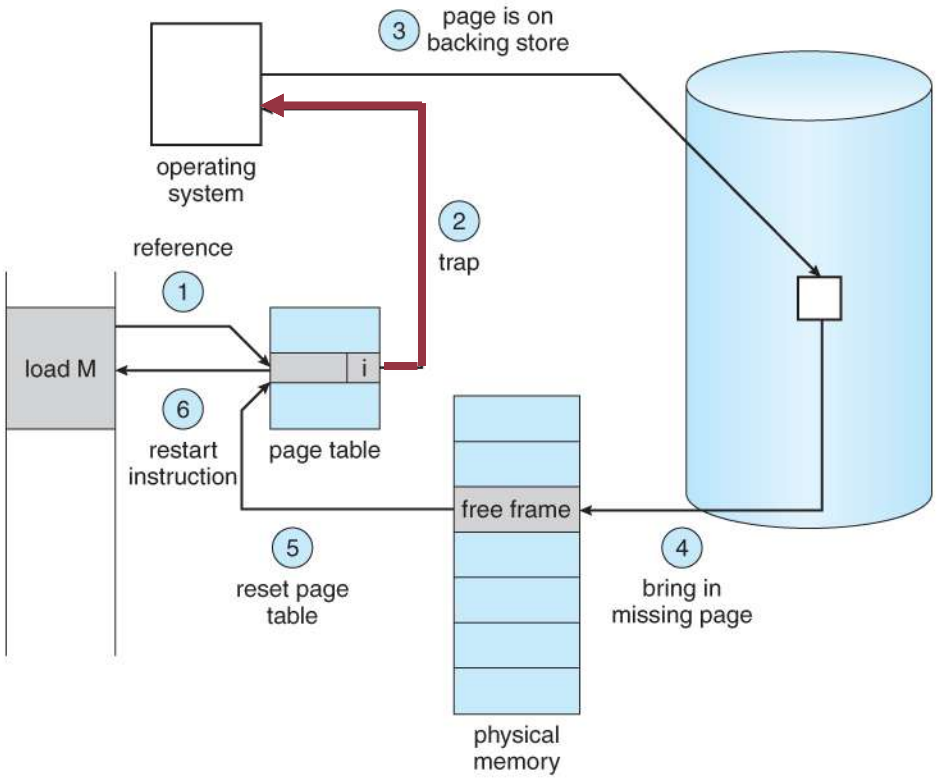
Otherwise, a page fault trap occurs, and the page has to be loaded (i.e., fetched) from disk

Page Fault Handling

1. The memory address is first checked, to see if it is legitimate



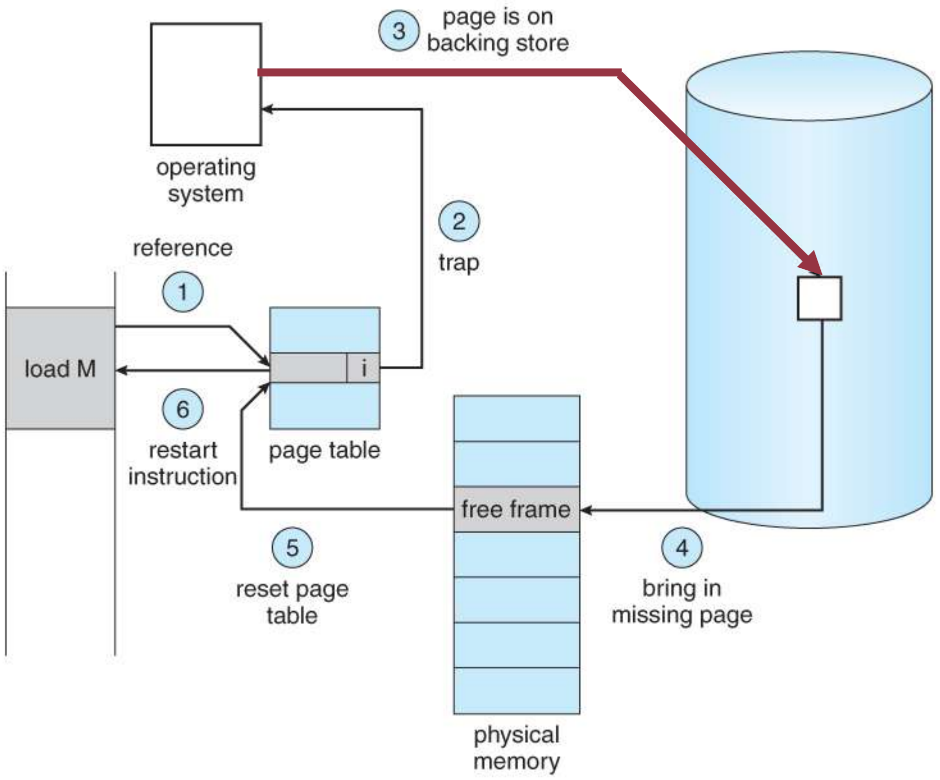
Page Fault Handling



1. The memory address is first checked, to see if it is legitimate

2. If the address is legitimate the page must be fetched from disk

Page Fault Handling

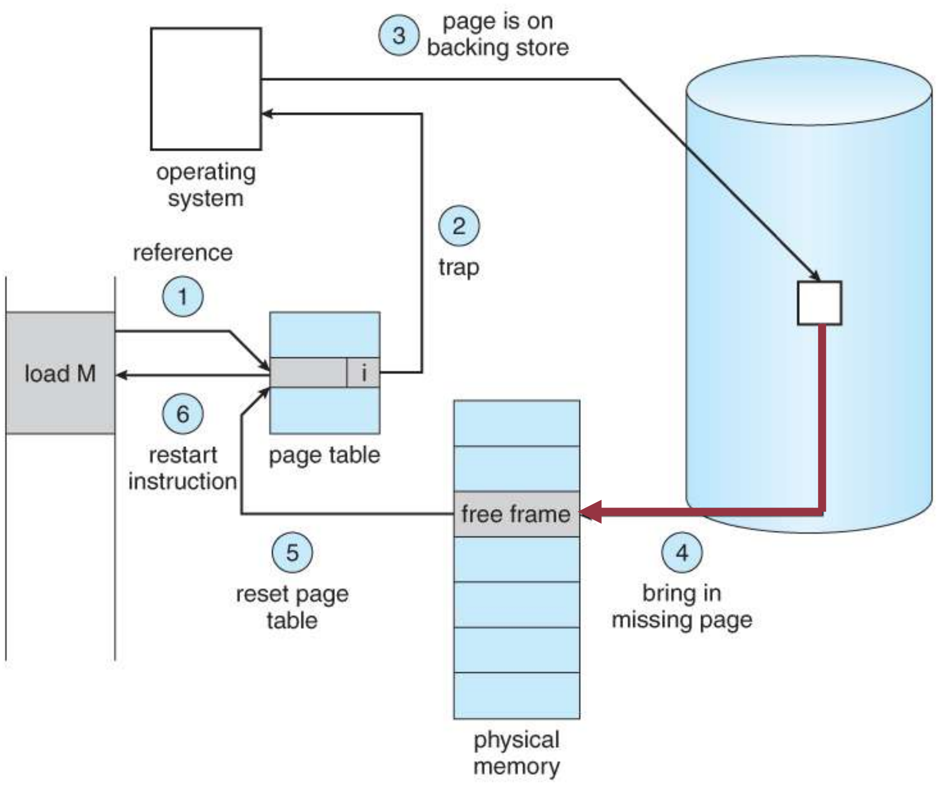


1. The memory address is first checked, to see if it is legitimate

2. If the address is legitimate the page must be fetched from disk

3. A free frame is located, possibly from a free-frame list (the OS might need to pick a frame to unload if all memory is full)

Page Fault Handling



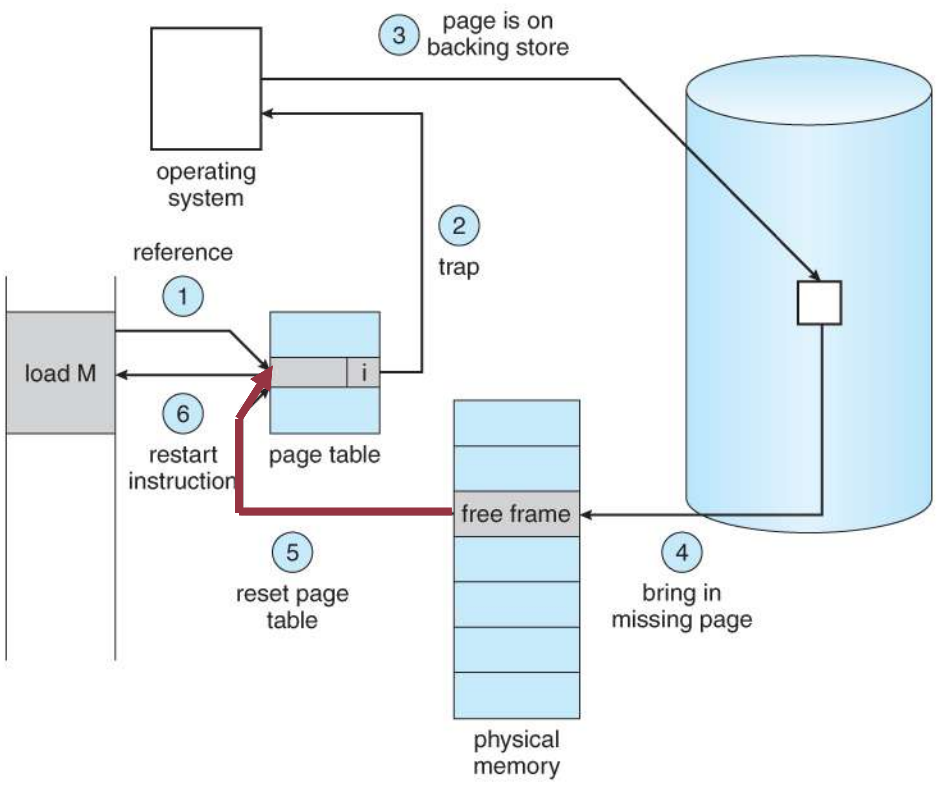
1. The memory address is first checked, to see if it is legitimate

2. If the address is legitimate the page must be fetched from disk

3. A free frame is located, possibly from a free-frame list (the OS might need to pick a frame to unload if all memory is full)

4. A disk operation is scheduled to bring in the necessary page from disk (this will block the process on an I/O wait, allowing some other process to run)

Page Fault Handling



1. The memory address is first checked, to see if it is legitimate

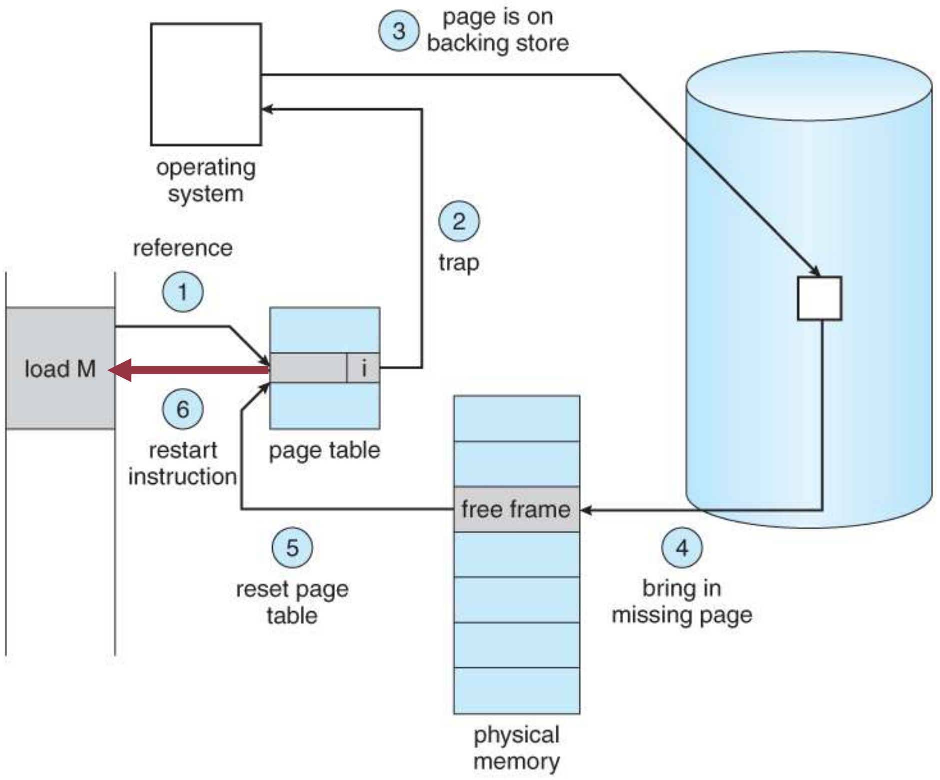
2. If the address is legitimate the page must be fetched from disk

3. A free frame is located, possibly from a free-frame list (the OS might need to pick a frame to unload if all memory is full)

4. A disk operation is scheduled to bring in the necessary page from disk (this will block the process on an I/O wait, allowing some other process to run)

5. When the I/O operation is complete, the process's page table is updated with the new frame number, and the bit is set to valid

Page Fault Handling



1. The memory address is first checked, to see if it is legitimate

2. If the address is legitimate the page must be fetched from disk

3. A free frame is located, possibly from a free-frame list (the OS might need to pick a frame to unload if all memory is full)

4. A disk operation is scheduled to bring in the necessary page from disk (this will block the process on an I/O wait, allowing some other process to run)

5. When the I/O operation is complete, the process's page table is updated with the new frame number, and the bit is set to valid

6. The current process gets interrupted and the instruction that caused the page fault must be restarted from the beginning

Page Fault Handling: TLB Hit

- The TLB also uses the valid bit to indicate the fact that the page is in main memory

Page Fault Handling: TLB Hit

- The TLB also uses the valid bit to indicate the fact that the page is in main memory
- If we get a TLB hit but the frame is not actually in main memory, we have to go fetch the page from disk anyway!

Page Fault Handling: TLB Hit

- The TLB also uses the valid bit to indicate the fact that the page is in main memory
- If we get a TLB hit but the frame is not actually in main memory, we have to go fetch the page from disk anyway!
- TLB hit means the requested page entry is in the cache and the referenced frame is also in memory

Page Fault Handling: TLB Miss

- If the requested page is not in the cache (TLB miss) but it is in memory:

Page Fault Handling: TLB Miss

- If the requested page is not in the cache (TLB miss) but it is in memory:
    - The OS picks a TLB entry to replace and fills it with the new entry

Page Fault Handling: TLB Miss

- If the requested page is not in the cache (TLB miss) but it is in memory:
  - The OS picks a TLB entry to replace and fills it with the new entry
- If the requested page is not in the cache (TLB miss) and it is not even in memory (i.e., it is sitting on disk):

Page Fault Handling: TLB Miss

- If the requested page is not in the cache (TLB miss) but it is in memory:
  - The OS picks a TLB entry to replace and fills it with the new entry
- If the requested page is not in the cache (TLB miss) and it is not even in memory (i.e., it is sitting on disk):
  - The OS picks a TLB entry to replace and fills it with the new entry as follows
    - invalidates the TLB entry
    - performs page fault trap operations
    - updates the TLB entry
    - restarts the faulting instruction

Page Fault Handling: Faulty Address

- How does the OS figure out which page generated the fault?

Page Fault Handling: Faulty Address

- How does the OS figure out which page generated the fault?
- Architecture-dependent:
  - x86: hardware saves the virtual address that caused the fault (`CR2` register)
  - On some platforms, OS gets only address of faulting instruction, must simulate the instruction and try every address to find the one that generated the fault

Page Fault Handling: Transparency

- Transparently restarting process execution after a page fault is tricky, since the fault may have occurred in the middle of an instruction

Page Fault Handling: Transparency

- Transparently restarting process execution after a page fault is tricky, since the fault may have occurred in the middle of an instruction
- To restart (from scratch) a faulty instruction the OS needs hardware support for saving:
  - The faulting instruction
  - The CPU state

Page Fault Handling: Transparency

- idempotent vs. non-idempotent instructions

Page Fault Handling: Transparency

- idempotent vs. non-idempotent instructions
- idempotent → just restart the faulting instruction (hardware saves instruction address during page fault)

Page Fault Handling: Transparency

- idempotent vs. non-idempotent instructions
- idempotent → just restart the faulting instruction (hardware saves instruction address during page fault)
- non-idempotent → much more difficult to restart
  - `MOV [%R1], +(%R2)` → increment the value of `R2` and store it to memory address in `R1`
  - What if memory address `[%R1]` causes the page fault?
  - Cannot naively redo the instruction from scratch, otherwise `R2` gets incremented twice

Page Fault Handling: Transparency

- Even harder when using instructions that are not easily undoable
  - E.g., instructions that are used to move a block of memory at once
  - The block may span multiple pages: some of them can be in memory while some others not
  - Pages that are in memory can be changed meanwhile a page fault occurs

Page Fault Handling: Transparency

- Even harder when using instructions that are not easily undoable
  - E.g., instructions that are used to move a block of memory at once
  - The block may span multiple pages: some of them can be in memory while some others not
  - Pages that are in memory can be changed meanwhile a page fault occurs

How to unwind those complicated side-effects?

Page Fault Handling: Transparency

- Even harder when using instructions that are not easily undoable
  - E.g., instructions that are used to move a block of memory at once
  - The block may span multiple pages: some of them can be in memory while some others not
  - Pages that are in memory can be changed meanwhile a page fault occurs

How to unwind those complicated side-effects?

Make sure all the addresses within the block to be transferred are in memory before starting executing the instruction

Virtual Memory: Performance

- Theoretically, a page fault may occur at each process instruction
  - A process may reference addresses belonging to different page at each step

Virtual Memory: Performance

- Theoretically, a page fault may occur at each process instruction
  - A process may reference addresses belonging to different page at each step
- Luckily, processes usually exhibit so-called locality of reference

Virtual Memory: Performance

- Theoretically, a page fault may occur at each process instruction
  - A process may reference addresses belonging to different page at each step
- Luckily, processes usually exhibit so-called locality of reference
  - temporal → if a process accesses an item in memory, it will tend to reference the same item again soon

Virtual Memory: Performance

- Theoretically, a page fault may occur at each process instruction
  - A process may reference addresses belonging to different page at each step
- Luckily, processes usually exhibit so-called locality of reference
  - temporal → if a process accesses an item in memory, it will tend to reference the same item again soon
  - spatial → if a process accesses an item in memory, it will tend to reference a close item again soon

Virtual Memory: Performance

- Theoretically, a page fault may occur at each process instruction
  - A process may reference addresses belonging to different page at each step
- Luckily, processes usually exhibit so-called locality of reference
  - temporal → if a process accesses an item in memory, it will tend to reference the same item again soon
  - spatial → if a process accesses an item in memory, it will tend to reference a close item again soon
- Still, an overhead must be paid every time a page fault occurs as the OS needs to interact with slower disk

Virtual Memory: Performance

$$t_{MA} = \text{physical memory access time}$$
$$t_{FAULT} = \text{time to handle a page fault}$$
$$p \in [0, 1] = \text{probability of page fault}$$
$$t_{ACCESS} = \text{effective time for each memory reference}$$

$$t_{ACCESS} = (1 - p) * t_{MA} + p * t_{FAULT}$$

Let's assume: $t_{MA} = 100$ nsec and $t_{FAULT} = 20$ msec $= 20,000,000$ nsec

$$t_{ACCESS} = (1 - p) * 100 + p * 20,000,000$$

Virtual Memory: Performance

$$t_{MA} = \text{physical memory access time}$$
$$t_{FAULT} = \text{time to handle a page fault}$$
$$p \in [0, 1] = \text{probability of page fault}$$
$$t_{ACCESS} = \text{effective time for each memory reference}$$

$$\boxed{t_{ACCESS} = (1 - p) * t_{MA} + p * t_{FAULT}}$$

Let's assume: $t_{MA} = 100$ nsec and $t_{FAULT} = 20$ msec $= 20,000,000$ nsec

$$t_{ACCESS} = (1 - p) * 100 + p * 20,000,000$$

This heavily depends on p!

Virtual Memory: Performance Example

$$t_{ACCESS} = (1 - p) * 100 + p * 20,000,000$$

What if only 1 every 1,000 memory references causes a page fault (i.e., p = 0.001)

Virtual Memory: Performance Example

$$t_{ACCESS} = (1 - p) * 100 + p * 20,000,000$$

What if only 1 every 1,000 memory references causes a page fault (i.e., $p = 0.001$)

The access time increases from just 100 nsec up to ~20.1 microsec

200 times slowdown factor

Virtual Memory: Performance Example

$$t_{ACCESS} = (1-p)*100 + p*20,000,000$$

What if we want the time access to be at most 10% slower than basic memory access?

Virtual Memory: Performance Example

$$t_{ACCESS} = (1 - p) * 100 + p * 20,000,000$$

What if we want the time access to be at most 10% slower than basic memory access?

We have to solve for p the following equation:

$$1.1 * 100 = (1 - p) * 100 + p * 20,000,000$$

Virtual Memory: Performance Example

$$t_{ACCESS} = (1-p)*100 + p*20,000,000$$

What if we want the time access to be at most 10% slower than basic memory access?

We have to solve for *p* the following equation:

$$1.1*100 = (1-p)*100 + p*20,000,000$$

$$1.1*100 = 100 - 100p + 20,000,000p =$$
$$19,999,900p = 110 - 100 =$$

To achieve that goal, we can tolerate at most 1 page fault every about 2 million accesses!

$$p = \frac{10}{19,999,900} = \frac{1}{1,999,990} \boxed{\approx 0,0000005 = 5*10^{-7}}$$

Virtual Memory: Performance Example

More generally, given t_{MA}, t_{FAULT}, and a threshold $\varepsilon > 0$ if we want to find p s.t.:

$$t_{ACCESS} = (1 + \epsilon) * t_{MA}$$

We substitute t_{ACCESS} and solve for p the resulting equation:

$$(1 - p) * t_{MA} + p * t_{FAULT} = (1 + \epsilon) * t_{MA} =$$
$$t_{MA} - p * t_{MA} + p * t_{FAULT} = t_{MA} + \epsilon * t_{MA}$$
$$p(t_{FAULT} - t_{MA}) = \epsilon * t_{MA} =$$

$$\boxed{p = \frac{\epsilon * t_{MA}}{t_{FAULT} - t_{MA}}}$$

Virtual Memory: Considerations

- So far, we have described how the OS (with the support of HW) manages page faults

Virtual Memory: Considerations

- So far, we have described how the OS (with the support of HW) manages page faults
- Still, the OS has to answer 2 fundamental questions:
  - When to load process' pages into main memory (page fetching)
  - Which page to remove from memory if this gets filled (page replacement)

Page Fetching Goals

- The overall goal is still to make physical memory look larger than it is
- Exploiting the locality reference of programs
- Keep in memory only those pages that is being used
- Keep on disk those pages that are unused
- Ideally, producing a memory system with the performance of main memory and the cost/capacity of disk!

Page Fetching Strategies

3 page fetching strategies

Page Fetching Strategies

3 page fetching strategies

Startup

This is a special case where all the pages of the process are loaded at once

virtual address space cannot be larger than physical memory

Page Fetching Strategies

3 page fetching strategies

Startup

This is a special case where all the pages of the process are loaded at once

virtual address space cannot be larger than physical memory

Overlays

Let the programmer says when pages are loaded/ removed

virtual address space can be larger than physical memory but hard and error-prone

Page Fetching Strategies

3 page fetching strategies

Startup

This is a special case where all the pages of the process are loaded at once

virtual address space cannot be larger than physical memory

Overlays

Let the programmer says when pages are loaded/ removed

virtual address space can be larger than physical memory but hard and error-prone

Demand

Process tells the OS when it needs a page

The OS manages page requests

Page Fetching Strategies

3 page fetching strategies

Startup

This is a special case where all the pages of the process are loaded at once

virtual address space cannot be larger than physical memory

Overlays

Let the programmer says when pages are loaded/ removed

virtual address space can be larger than physical memory but hard and error-prone

Demand

Process tells the OS when it needs a page

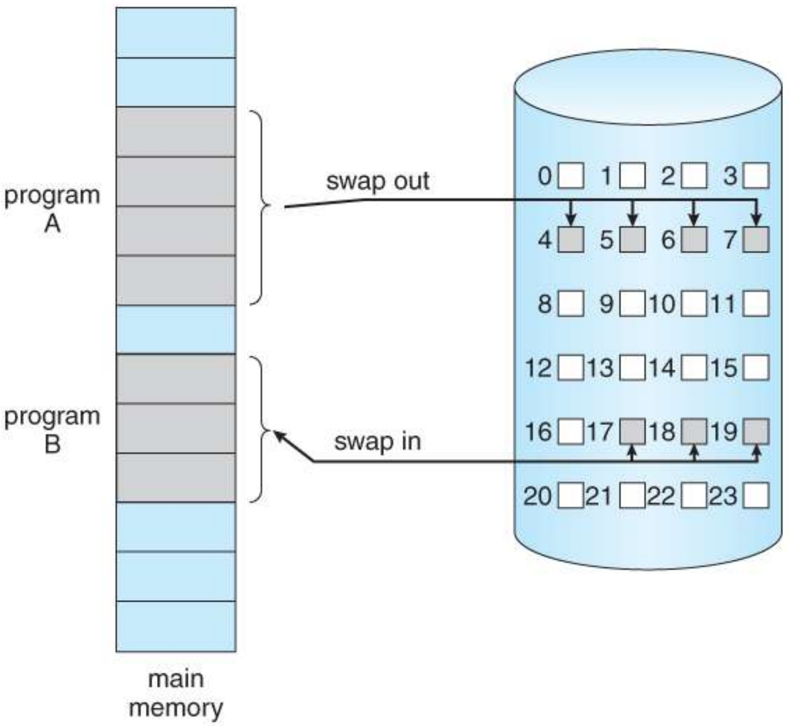
The OS manages page requests

Most modern OSs use demand fetching

(Pure) Demand Paging

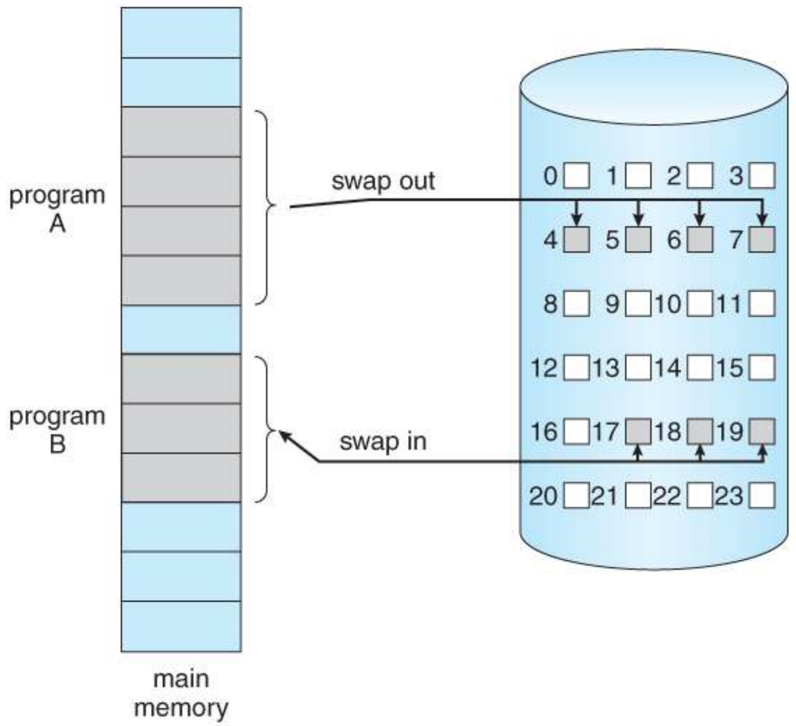
- When a process starts up, none of its pages are loaded
- Rather, a page is swapped in only when the process references it (upon a page fault)
- This is termed a lazy swapper or pager
- Opposite of loading all the pages at process startup!

Prefetching



The pager guesses when pages will be needed and load them ahead of time

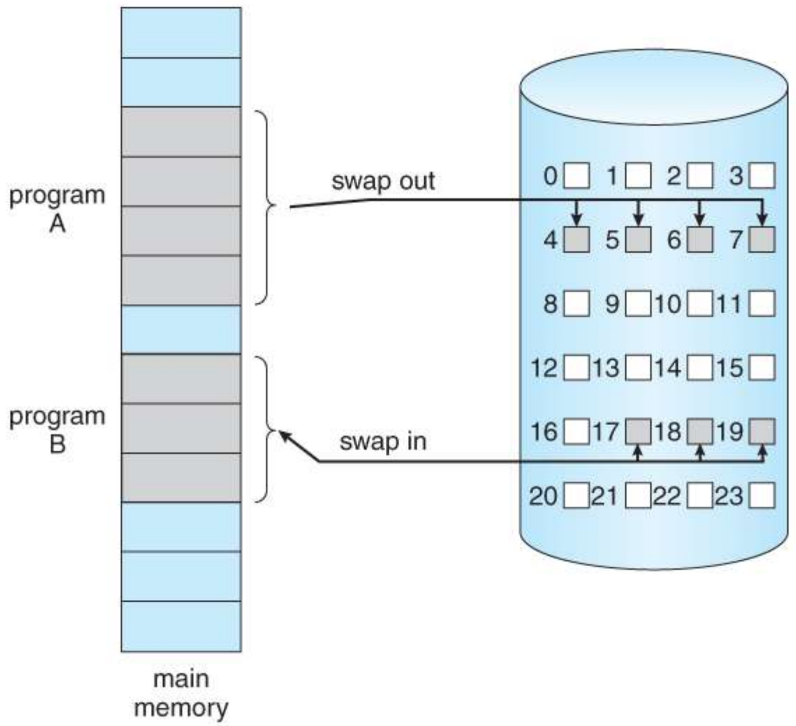
Prefetching



The pager guesses when pages will be needed and load them ahead of time

Trying to avoid page faults

Prefetching

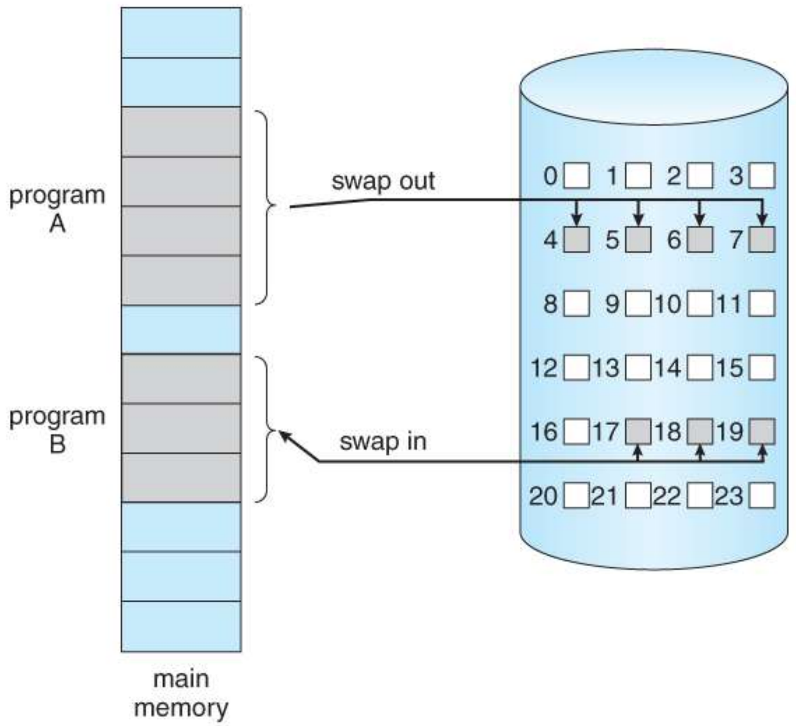


The pager guesses when pages will be needed and load them ahead of time

Trying to avoid page faults

Requires predicting the future → very hard!

Prefetching



The pager guesses when pages will be needed and load them ahead of time

Trying to avoid page faults

Requires predicting the future → very hard!

Possible approach: upon page fault, load many pages instead of only the faulty one

works if program accesses memory sequentially

Swap Space

- A portion of the disk reserved for storing pages that are evicted from memory

Swap Space

- A portion of the disk reserved for storing pages that are evicted from memory
- May be not part of the actual disk file system

Swap Space

- A portion of the disk reserved for storing pages that are evicted from memory
- May be not part of the actual disk file system
- On Linux there exists a dedicated swap partition (on disk)
  - no actual files are stored in that partition

Swap Space

- A portion of the disk reserved for storing pages that are evicted from memory
- May be not part of the actual disk file system
- On Linux there exists a dedicated swap partition (on disk)
  - no actual files are stored in that partition
- On Mac, instead, swap space is part of the file system
  - swapfiles

Swap Out

- When a page needs to be swapped out, it will be generally copied to disk

Swap Out

- When a page needs to be swapped out, it will be generally copied to disk
- The pages for a process are divided into 2 groups:
  - Code (read-only)
  - Data (initialized/uninitialized)

Swap Out

- When a page needs to be swapped out, it will be generally copied to disk
- The pages for a process are divided into 2 groups:
    - Code (read-only)
    - Data (initialized/uninitialized)
- Depending on which kind of page is removed, different optimizations may apply upon page swap-out

Swap Out Optimizations

- Code page (read-only):
  - Code content does not change!
  - Just remove it and re-load it back from executable file stored on disk
  - Make use of the filesystem

Swap Out Optimizations

- Code page (read-only):
  - Code content does not change!
  - Just remove it and re-load it back from executable file stored on disk
  - Make use of the filesystem
- Data page:
  - Data content does actually change!
  - Save it to a separate paging file, so that no changes are lost when it will be loaded in the future
  - Need to use the dedicated swap space

Page Replacement: Motivation

- On a page fault, we need to load a page from disk into memory
- If physical memory has still free frames, the page can be safely loaded into one of those
- However, if physical memory is full (i.e., all of its frames are loaded) a frame must be swapped out to make room for the swap-in page
- Several algorithms to select the page to evict from memory

Page Replacement Algorithms

- Random: pick any page at random (works surprisingly well!)

Page Replacement Algorithms

- Random: pick any page at random (works surprisingly well!)
- FIFO (First-In-First-Out): throw out the page that has been in memory for longest time (i.e., the oldest)
  - Easy to implement but may remove frequently accessed pages

Page Replacement Algorithms

- Random: pick any page at random (works surprisingly well!)
- FIFO (First-In-First-Out): throw out the page that has been in memory for longest time (i.e., the oldest)
  - Easy to implement but may remove frequently accessed pages
- MIN (OPT): remove the page that will not be accessed for the longest time (provably optimal [Belady 1966])
  - Needs to predict the future → very hard!

Page Replacement Algorithms

- Random: pick any page at random (works surprisingly well!)
- FIFO (First-In-First-Out): throw out the page that has been in memory for longest time (i.e., the oldest)
  - Easy to implement but may remove frequently accessed pages
- MIN (OPT): remove the page that will not be accessed for the longest time (provably optimal [Belady 1966])
  - Needs to predict the future → very hard!
- LRU (Least Recently Used): approximation of MIN, remove the page that has not been used in the longest time
  - Assumes the past is a good predictor of the future (not always true!)

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1											
F_2											
F_3											

How many page faults (denoted by *)?

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1											
F_2											
F_3											

Initially, no frame is loaded in memory at all (pure demand paging)

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1											
F_2											
F_3											

Virtual address within page A is referenced

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1											
F_2											
F_3											

Virtual address within page A is referenced ⇒ **page fault**

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*										
F_2											
F_3											

Virtual address within page A is referenced → page fault → A loaded

FIFO = A

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A									
F_2											
F_3											

Virtual address within page B is referenced

FIFO = A

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A									
F_2											
F_3											

Virtual address within page B is referenced → page fault

FIFO = A

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A									
F_2		B*									
F_3											

Virtual address within page B is referenced → page fault → B loaded

FIFO = A → B

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A								
F_2		B*	B								
F_3											

Virtual address within page C is referenced

FIFO = A → B

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A								
F_2		B*	B								
F_3											

Virtual address within page C is referenced ⇒ page fault

FIFO = A → B

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A								
F_2		B*	B								
F_3			C*								

Virtual address within page C is referenced ⇒ page fault ⇒ C loaded

FIFO = A → B → C

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A							
F_2		B*	B	B							
F_3			C*	C							

Virtual address within page A is referenced → A is already loaded

FIFO = A → B → C

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A						
F_2		B*	B	B	B						
F_3			C*	C	C						

Virtual address within page B is referenced → B is already loaded

FIFO = A → B → C

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	A					
F_2		B*	B	B	B	B					
F_3			C*	C	C	C					

Virtual address within page D is referenced

FIFO = A → B → C

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	A					
F_2		B*	B	B	B	B					
F_3			C*	C	C	C					

Virtual address within page D is referenced ⇨ page fault

FIFO = A → B → C

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	D*					
F_2		B*	B	B	B	B					
F_3			C*	C	C	C					

Virtual address within page D is referenced → page fault → A replaced D loaded

FIFO = B → C → D

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	$\boxed{A}$	D	B	C	A
F_1	A*	A	A	A	A	D*	D				
F_2		B*	B	B	B	B	B				
F_3			C*	C	C	C	C				

Virtual address within page A is referenced

FIFO = B → C → D

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	D*	D				
F_2		B*	B	B	B	B	B				
F_3			C*	C	C	C	C				

Virtual address within page A is referenced → page fault

FIFO = B → C → D

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	$\boxed{A}$	D	B	C	A
F_1	A*	A	A	A	A	D*	D				
F_2		B*	B	B	B	B	A*				
F_3			C*	C	C	C	C				

Virtual address within page A is referenced ⇒ page fault ⇒ B replaced A loaded

FIFO = C → D → A

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	D*	D	D			
F_2		B*	B	B	B	B	A*	A			
F_3			C*	C	C	C	C	C			

Virtual address within page D is referenced → D is already loaded

FIFO = C → D → A

FIFO Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	D*	D	D	D	C*	C
F_2		B*	B	B	B	B	A*	A	A	A	A
F_3			C*	C	C	C	C	C	B*	B	B

Eventually, we get a total of 7 page faults

MIN Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1											
F_2											
F_3											

How many page faults (denoted by *)?

MIN Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1											
F_2											
F_3											

Initially, no frame is loaded in memory at all
(pure demand paging)

MIN Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A						
F_2		B*	B	B	B						
F_3			C*	C	C						

Up to this point, the same as FIFO

MIN Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	A					
F_2		B*	B	B	B	B					
F_3			C*	C	C	C					

Virtual address within page D is referenced

MIN Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	A					
F_2		B*	B	B	B	B					
F_3			C*	C	C	C					

Virtual address within page D is referenced → page fault

MIN Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	A					
F_2		B*	B	B	B	B					
F_3			C*	C	C	C					

Virtual address within page D is referenced → **page fault**

What's the page that will be requested the furthest away?

MIN Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	A					
F_2		B*	B	B	B	B					
F_3			C*	C	C	D*					

Virtual address within page D is referenced → page fault → C replaced D loaded

C is the page that will be requested the furthest away

MIN Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	A	A	A	A		
F_2		B*	B	B	B	B	B	B	B		
F_3			C*	C	C	D*	D	D	D		

Up to this point, no more page faults

MIN Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	A	A	A	A	A	
F_2		B*	B	B	B	B	B	B	B	B	
F_3			C*	C	C	D*	D	D	D	D	

Virtual address within page C is referenced

MIN Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	A	A	A	A	A	
F_2		B*	B	B	B	B	B	B	B	B	
F_3			C*	C	C	D*	D	D	D	D	

Virtual address within page C is referenced ⇒ page fault

What's the page that will be requested the furthest away?

MIN Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	$\boxed{C}$	A
F_1	A*	A	A	A	A	A	A	A	A	A	
F_2		B*	B	B	B	B	B	B	B	C*	
F_3			C*	C	C	D*	D	D	D	D	

Virtual address within page C is referenced → page fault → B replaced C loaded

B or D will be requested the furthest away (surely not A): pick one (e.g., B)

MIN Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	A	A	A	A	A	A
F_2		B*	B	B	B	B	B	B	B	C*	C
F_3			C*	C	C	D*	D	D	D	D	D

Eventually, we get a total of 5 page faults

LRU Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1											
F_2											
F_3											

How many page faults (denoted by *)?

LRU Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1											
F_2											
F_3											

Initially, no frame is loaded in memory at all (pure demand paging)

LRU Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A						
F_2		B*	B	B	B						
F_3			C*	C	C						

Up to this point, the same as FIFO

LRU Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	A					
F_2		B*	B	B	B	B					
F_3			C*	C	C	C					

Virtual address within page D is referenced ⇒ page fault

LRU Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	A					
F_2		B*	B	B	B	B					
F_3			C*	C	C	C					

Virtual address within page D is referenced ⇒ page fault

We can't look forward anymore!

LRU Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	$\boxed{D}$	A	D	B	C	A
F_1	A*	A	A	A	A	A					
F_2		B*	B	B	B	B					
F_3			C*	C	C	D*					

Virtual address within page D is referenced → **page fault** → C replaced, D loaded

C is the page that has not been used for the longest time in the past

LRU Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	A	A	A	A		
F_2		B*	B	B	B	B	B	B	B		
F_3			C*	C	C	D*	D	D	D		

Up to this point, no more page faults

LRU Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	A	A	A	A	A	
F_2		B*	B	B	B	B	B	B	B	B	
F_3			C*	C	C	D*	D	D	D	D	

Virtual address within page C is referenced

LRU Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	A	A	A	A	A	
F_2		B*	B	B	B	B	B	B	B	B	
F_3			C*	C	C	D*	D	D	D	D	

Virtual address within page C is referenced ⇒ page fault

We can't look forward anymore!

LRU Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	A	A	A	A	C*	
F_2		B*	B	B	B	B	B	B	B	B	
F_3			C*	C	C	D*	D	D	D	D	

Virtual address within page C is referenced → page fault → A replaced C loaded

A is the page that has not been used for the longest time in the past

LRU Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	A	A	A	A	C*	C
F_2		B*	B	B	B	B	B	B	B	B	B
F_3			C*	C	C	D*	D	D	D	D	D

Virtual address within page A is referenced

LRU Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	A	A	A	A	C*	C
F_2		B*	B	B	B	B	B	B	B	B	B
F_3			C*	C	C	D*	D	D	D	D	D

Virtual address within page A is referenced → page fault

We can't look forward anymore!

LRU Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	A	A	A	A	C*	C
F_2		B*	B	B	B	B	B	B	B	B	B
F_3			C*	C	C	D*	D	D	D	D	A*

Virtual address within page A is referenced → page fault → D replaced A loaded

D is the page that has not been used for the longest time in the past

LRU Page Replacement: Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, A, B, D, A, D, B, C, A

	A	B	C	A	B	D	A	D	B	C	A
F_1	A*	A	A	A	A	A	A	A	A	C*	C
F_2		B*	B	B	B	B	B	B	B	B	B
F_3			C*	C	C	D*	D	D	D	D	A*

Eventually, we get a total of 6 page faults

LRU Page Replacement: (An Unlucky) Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, D, A, B, C, D, A, B, C

	A	B	C	D	A	B	C	D	A	B	C
F_1											
F_2											
F_3											

How many page faults (denoted by *)?

LRU Page Replacement: (An Unlucky) Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, D, A, B, C, D, A, B, C

	A	B	C	D	A	B	C	D	A	B	C
F_1	A*	A	A								
F_2		B*	B								
F_3			C*								

LRU Page Replacement: (An Unlucky) Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, D, A, B, C, D, A, B, C

	A	B	C	D	A	B	C	D	A	B	C
F_1	A*	A	A	D*							
F_2		B*	B	B							
F_3			C*	C							

LRU Page Replacement: (An Unlucky) Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, D, A, B, C, D, A, B, C

	A	B	C	D	$\boxed{A}$	B	C	D	A	B	C
F_1	A*	A	A	D*	D						
F_2		B*	B	B	A*						
F_3			C*	C	C						

LRU Page Replacement: (An Unlucky) Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, D, A, B, C, D, A, B, C

	A	B	C	D	A	B	C	D	A	B	C
F_1	A*	A	A	D*	D	D					
F_2		B*	B	B	A*	A					
F_3			C*	C	C	B*					

LRU Page Replacement: (An Unlucky) Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, D, A, B, C, D, A, B, C

	A	B	C	D	A	B	C	D	A	B	C
F_1	A*	A	A	D*	D	D	C*				
F_2		B*	B	B	A*	A	A				
F_3			C*	C	C	B*	B				

LRU Page Replacement: (An Unlucky) Example

3 physical frames: F_1, F_2, F_3

4 virtual pages: A, B, C, D

Reference sequence of pages: A, B, C, D, A, B, C, D, A, B, C

	A	B	C	D	A	B	C	D	A	B	C
F_1	A*	A	A	D*	D	D	C*	C	C	B*	B
F_2		B*	B	B	A*	A	A	D*	D	D	C*
F_3			C*	C	C	B*	B	B	A*	A	A

Eventually, we get a total of 11 page faults

Page Replacement: What If We Add Memory?

- Does adding memory always reduce the number of page faults?
- Intuitively, it would seem so…
- The answer, in fact, depends on the page replacement algorithm
- Let's see this with an example, using FIFO page replacement

FIFO Page Replacement: Example

5 virtual pages: A, B, C, D, E

3 physical frames: F_1, F_2, F_3

Scenario 1

4 physical frames: F_1, F_2, F_3, F_4

Scenario 2

Reference sequence of pages: A, B, C, D, A, B, E, A, B, C, D, E

FIFO Page Replacement: Example

	A	B	C	D	A	B	E	A	B	C	D	E

F_1	A*	A	A	D*	D	D	E*	E	E	E	E	E
F_2		B*	B	B	A*	A	A	A	A	C*	C	C
F_3			C*	C	C	B*	B	B	B	B	D*	D

F_1	A*	A	A	A	A	A	E*	E	E	E	D*	D
F_2		B*	B	B	B	B	B	A*	A	A	A	E*
F_3			C*	C	C	C	C	C	B*	B	B	B
F_4				D*	D	D	D	D	D	C*	C	C

FIFO Page Replacement: Example

	A	B	C	D	A	B	E	A	B	C	D	E

F_1	A*	A	A	D*	D	D	E*	E	E	E	E	E
F_2		B*	B	B	A*	A	A	A	A	C*	C	C
F_3			C*	C	C	B*	B	B	B	B	D*	D

10 page faults

F_1	A*	A	A	A	A	A	E*	E	E	E	D*	D
F_2		B*	B	B	B	B	B	A*	A	A	A	E*
F_3			C*	C	C	C	C	C	B*	B	B	B
F_4				D*	D	D	D	D	D	C*	C	C

11 page faults

Belady's Anomaly

Adding page frames may cause more page faults with some algorithms

LRU Page Replacement: Example

	A	B	C	D	A	B	E	A	B	C	D	E

F_1	A*	A	A	D*	D	D	E*	E	E	C*	C	C
F_2		B*	B	B	A*	A	A	A	A	A	D*	D
F_3			C*	C	C	B*	B	B	B	B	B	B

9 page faults

F_1	A*	A	A	A	A	A	A	A	A	A	A	E*
F_2		B*	B	B	B	B	B	B	B	B	B	B
F_3			C*	C	C	C	E*	E	E	E	D*	D
F_4				D*	D	D	D	D	D	C*	C	C

8 page faults

With LRU, adding page frames always decreases the number of page faults

LRU Page Replacement: Example

	A	B	C	D	A	B	E	A	B	C	D	E
F_1	A*	A	A	D*	D	D	E*	E	E	C*	C	C
F_2		B*	B	B	A*	A	A	A	A	A	D*	D
F_3			C*	C	C	B*	B	B	B	B	B	B

9 page faults

	A	B	C	D	A	B	E	A	B	C	D	E
F_1	A*	A	A	A	A	A	A	A	A	A	A	E*
F_2		B*	B	B	B	B	B	B	B	B	B	B
F_3			C*	C	C	C	E*	E	E	E	D*	D
F_4				D*	D	D	D	D	D	D	C*	C

8 page faults

With LRU, adding page frames always decreases the number of page faults

Why?

LRU Page Replacement: Example

	A	B	C	D	A	B	E	A	B	C	D	E

F_1	A*	A	A	D*	D	D	E*	E	E	C*	C	C
F_2		B*	B	B	A*	A	A	A	A	A	D*	D
F_3			C*	C	C	B*	B	B	B	B	B	B

F_1	A*	A	A	A	A	A	A	A	A	A	A	E*
F_2		B*	B	B	B	B	B	B	B	B	B	B
F_3			C*	C	C	C	E*	E	E	E	D*	D
F_4				D*	D	D	D	D	D	C*	C	C

At each point in time 4-frame memory contains a subset of 3-frame

Can't do any worst!

Page Replacement: Summary

- FIFO is easy to implement but may lead to too many page faults
  - May suffer from Belady's Anomaly

Page Replacement: Summary

- FIFO is easy to implement but may lead to too many page faults
  - May suffer from Belady's Anomaly
- MIN is the optimal choice but cannot be used in practice since future memory references are never known in advance

Page Replacement: Summary

- FIFO is easy to implement but may lead to too many page faults
  - May suffer from Belady's Anomaly
- MIN is the optimal choice but cannot be used in practice since future memory references are never known in advance
- LRU is a fair approximation of MIN assuming the past is a good predictor of the future
  - Exploits the locality reference (small working set that fits in memory)
  - Works poorly when the locality reference doesn't hold (large working set)

LRU: Implementation Details

How could we implement LRU page replacement algorithm?

LRU: Implementation Details

How could we implement LRU page replacement algorithm?

⇩

First Idea

Keep a timestamp for each page with the time it has been last accessed

Remove the page with the highest difference w.r.t. current timestamp

LRU: Implementation Details

How could we implement LRU page replacement algorithm?

⇩

First Idea

Keep a timestamp for each page with the time it has been last accessed

Remove the page with the highest difference w.r.t. current timestamp

Problems?

LRU: Implementation Details

How could we implement LRU page replacement algorithm?

First Idea

Keep a timestamp for each page with the time it has been last accessed

Remove the page with the highest difference w.r.t. current timestamp

Problems?

Every time a page is accessed its timestamp must be updated

LRU: Implementation Details

How could we implement LRU page replacement algorithm?

First Idea

Keep a timestamp for each page with the time it has been last accessed
Remove the page with the highest difference w.r.t. current timestamp

Problems?

Every time a page is accessed its timestamp must be updated

Linear scan of all the pages to select the one to be removed

LRU: Implementation Details

How could we implement LRU page replacement algorithm?

⇩

<u>Second Idea</u>

Keep a list of pages with the most recently used in front and the least recently used at the end: every time a page is accessed move it to front

LRU: Implementation Details

How could we implement LRU page replacement algorithm?

⇩

Second Idea

Keep a list of pages with the most recently used in front and the least recently used at the end: every time a page is accessed move it to front

Problems?

LRU: Implementation Details

How could we implement LRU page replacement algorithm?

Second Idea

Keep a list of pages with the most recently used in front and the least recently used at the end: every time a page is accessed move it to front

Problems?

Still too expensive as the OS must change multiple pointers on each memory access

LRU: Approximated Implementation

- In practice, no need for perfect LRU

LRU: Approximated Implementation

- In practice, no need for perfect LRU
- Many systems provide some HW support which is enough to approximate LRU fairly well

LRU: Approximated Implementation

- In practice, no need for perfect LRU
- Many systems provide some HW support which is enough to approximate LRU fairly well
- Single-Reference Bit → Maintain 1 bit for each page table entry
  - Initially, all bits for all pages are set to 0
  - On each access to a page, the HW sets the reference bit to 1
  - Enough to distinguish pages that have been accessed since the last clear
  - No total order of page access

LRU: Approximated Implementation

- Additional-Reference-Bits → Maintain 8 bits for each page table entry

LRU: Approximated Implementation

- Additional-Reference-Bits → Maintain 8 bits for each page table entry
  - At periodic intervals (clock interrupts), the OS takes over and right-shifts each of the reference bytes by one bit

LRU: Approximated Implementation

- Additional-Reference-Bits → Maintain 8 bits for each page table entry
  - At periodic intervals (clock interrupts), the OS takes over and right-shifts each of the reference bytes by one bit
  - The high-order (leftmost) bit is then filled in with the current value of the reference bit, and the reference bits are cleared

LRU: Approximated Implementation

- Additional-Reference-Bits → Maintain 8 bits for each page table entry
  - At periodic intervals (clock interrupts), the OS takes over and right-shifts each of the reference bytes by one bit
  - The high-order (leftmost) bit is then filled in with the current value of the reference bit, and the reference bits are cleared
  - At any given time, the page with the smallest value for the reference byte is the LRU page

LRU: Approximated Implementation

- Additional-Reference-Bits → Maintain 8 bits for each page table entry
  - At periodic intervals (clock interrupts), the OS takes over and right-shifts each of the reference bytes by one bit
  - The high-order (leftmost) bit is then filled in with the current value of the reference bit, and the reference bits are cleared
  - At any given time, the page with the smallest value for the reference byte is the LRU page
- The specific number of bits used and the frequency with which the reference byte is updated are adjustable

LRU: Approximated Implementation

- Second Chance Algorithm → Single-Reference Bit + FIFO

LRU: Approximated Implementation

- Second Chance Algorithm → Single-Reference Bit + FIFO
- OS keeps frames in a FIFO circular list

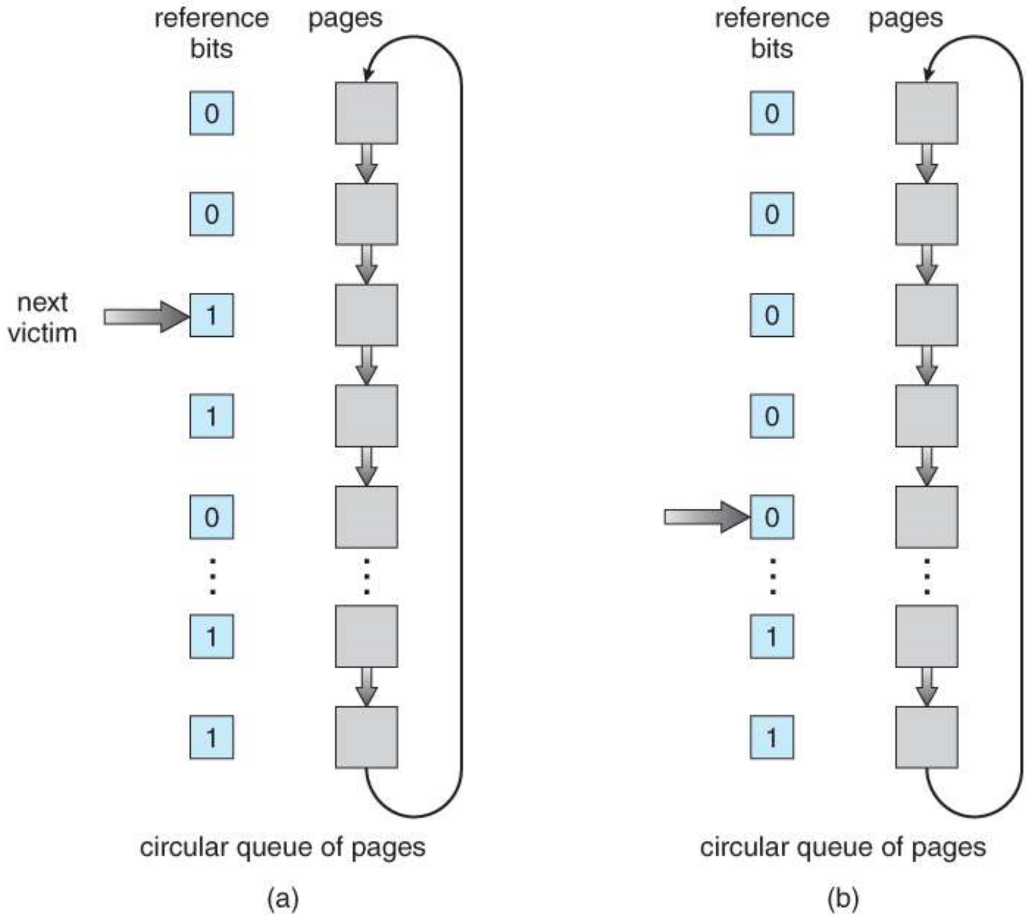
LRU: Approximated Implementation

- Second Chance Algorithm → Single-Reference Bit + FIFO
- OS keeps frames in a FIFO circular list
- On every memory access, the reference bit is set to 1

LRU: Approximated Implementation

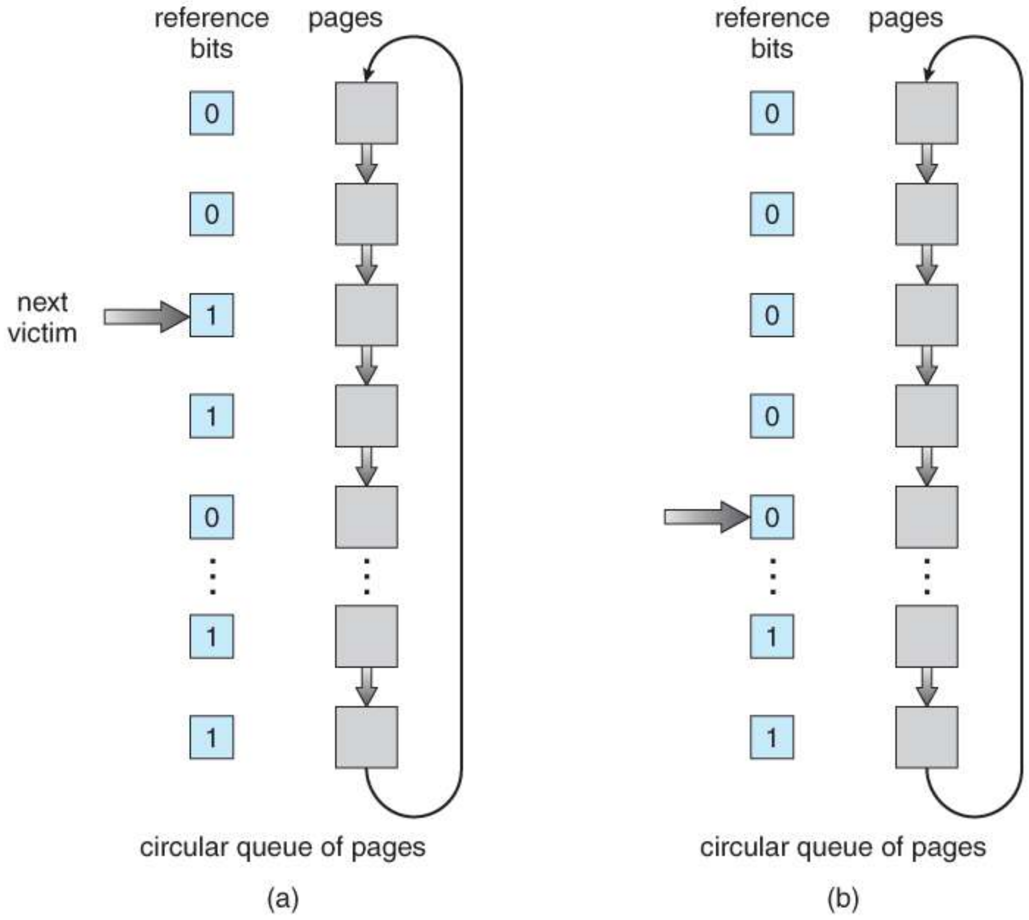
- Second Chance Algorithm → Single-Reference Bit + FIFO
- OS keeps frames in a FIFO circular list
- On every memory access, the reference bit is set to 1
- On a page fault, the OS scans the list of page table, checking the reference bit of the frame:
  - If this is 0, it replaces the page and sets it to 1
  - If this is 1, it sets it to 0 (second chance) and move to the next frame

Second Chance Algorithm (Clock)



A raw partitioning into: young vs. old frames

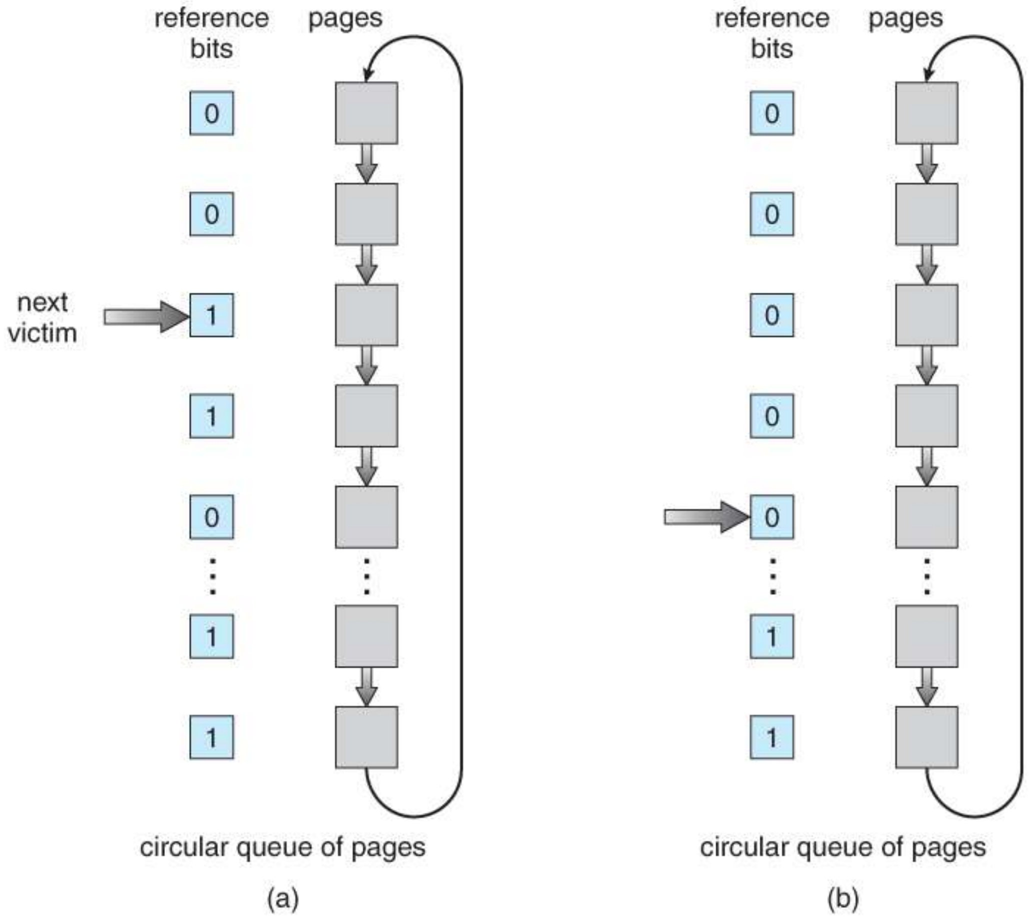
Second Chance Algorithm (Clock)



A raw partitioning into: young vs. old frames

Less accurate than additional-reference-bits

Second Chance Algorithm (Clock)

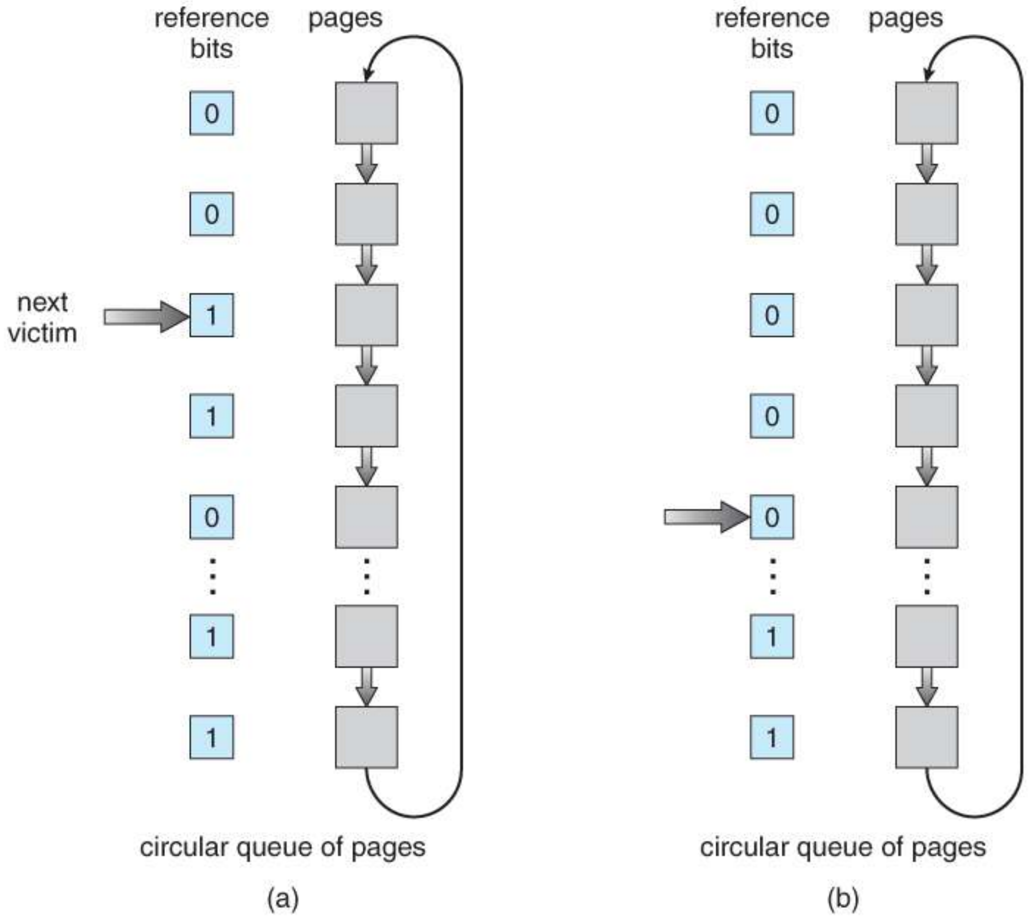


A raw partitioning into: young vs. old frames

Less accurate than additional-reference-bits

Fast, since it needs only to set a single bit on each memory reference (no need for a shift)

Second Chance Algorithm (Clock)



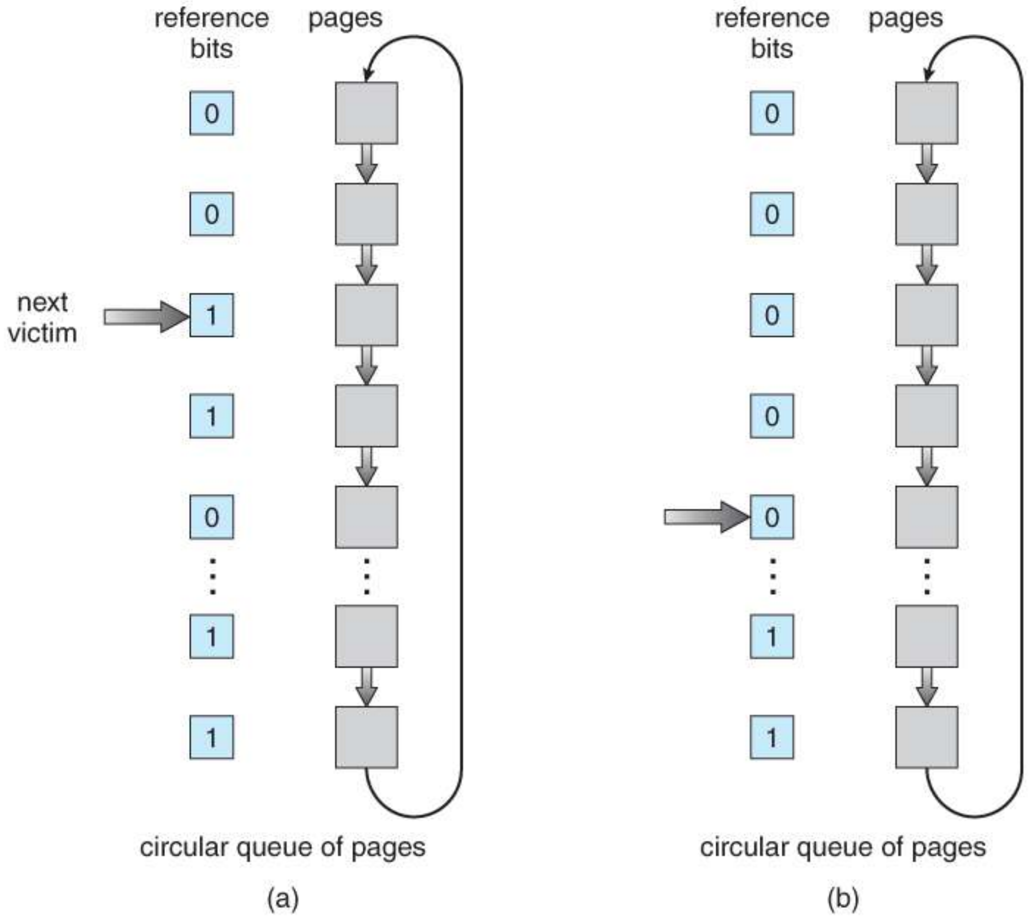
A raw partitioning into: young vs. old frames

Less accurate than additional-reference-bits

Fast, since it needs only to set a single bit on each memory reference (no need for a shift)

Page fault management is quicker as there is no need to scan the whole list of frames (on average) unless every frame has its bit set

Second Chance Algorithm (Clock)



A raw partitioning into: young vs. old frames

Less accurate than additional-reference-bits

Fast, since it needs only to set a single bit on each memory reference (no need for a shift)

Page fault management is quicker as there is no need to scan the whole list of frames (on average) unless every frame has its bit set

This algorithm is also known as clock because it mimics the hands of a clock

Enhanced Second Chance Algorithm

- Page replacement generally involves 2 I/O operations:
  - write the evicted page back to disk
  - read the newly referenced page from disk

Enhanced Second Chance Algorithm

- Page replacement generally involves 2 I/O operations:
  - write the evicted page back to disk
  - read the newly referenced page from disk
- Intuition: It is cheaper to replace a page which has not been modified, since the OS does not need to write this back to disk

Enhanced Second Chance Algorithm

- Page replacement generally involves 2 I/O operations:
  - write the evicted page back to disk
  - read the newly referenced page from disk
- Intuition: It is cheaper to replace a page which has not been modified, since the OS does not need to write this back to disk
- OS should give preference to paging-out un-modified frames
  - Yet, it can proactively write to disk modified frames for later

Enhanced Second Chance Algorithm

- HW keeps a modify bit (in addition to the reference bit)
    - 1 means the page has been modified (different from the copy on disk)
    - 0 means the page is the same as the one stored on disk

Enhanced Second Chance Algorithm

- HW keeps a modify bit (in addition to the reference bit)
  - 1 means the page has been modified (different from the copy on disk)
  - 0 means the page is the same as the one stored on disk
- Use both the reference and modify bits (r, m) to classify pages into:
  - (0, 0): neither recently used nor modified;
  - (0, 1): not recently used, but modified;
  - (1, 0): recently used, but clean
  - (1, 1): recently used and modified

Enhanced Second Chance Algorithm

- This algorithm searches the page table in a circular fashion as before, yet making 4 distinct passes (one for each category)

Enhanced Second Chance Algorithm

- This algorithm searches the page table in a circular fashion as before, yet making 4 distinct passes (one for each category)
- First, it looks for a (0, 0) frame, and if it can't find one, it makes another pass looking for a (0, 1), etc.

Enhanced Second Chance Algorithm

- This algorithm searches the page table in a circular fashion as before, yet making 4 distinct passes (one for each category)
- First, it looks for a (0, 0) frame, and if it can't find one, it makes another pass looking for a (0, 1), etc.
- The first page with the lowest numbered category is replaced

Enhanced Second Chance Algorithm

- This algorithm searches the page table in a circular fashion as before, yet making 4 distinct passes (one for each category)
- First, it looks for a (0, 0) frame, and if it can't find one, it makes another pass looking for a (0, 1), etc.
- The first page with the lowest numbered category is replaced
- The main difference between this algorithm and the standard clock is the preference for replacing clean pages if possible

Multiprogramming and Thrashing

- So far, we have implicitly assumed a single process is on the system

Multiprogramming and Thrashing

- So far, we have implicitly assumed a single process is on the system
- Multiple processes can however run concurrently on a single-CPU system

Multiprogramming and Thrashing

- So far, we have implicitly assumed a single process is on the system
- Multiple processes can however run concurrently on a single-CPU system
- The degree of multiprogramming is not fixed apriori, yet it is driven by the locality reference (a.k.a. 90÷10 rule)

Multiprogramming and Thrashing

- So far, we have implicitly assumed a single process is on the system
- Multiple processes can however run concurrently on a single-CPU system
- The degree of multiprogramming is not fixed apriori, yet it is driven by the locality reference (a.k.a. 90÷10 rule)
- This allows a system to load the working set (i.e., few pages) of many processes, thereby increasing the degree of multiprogramming

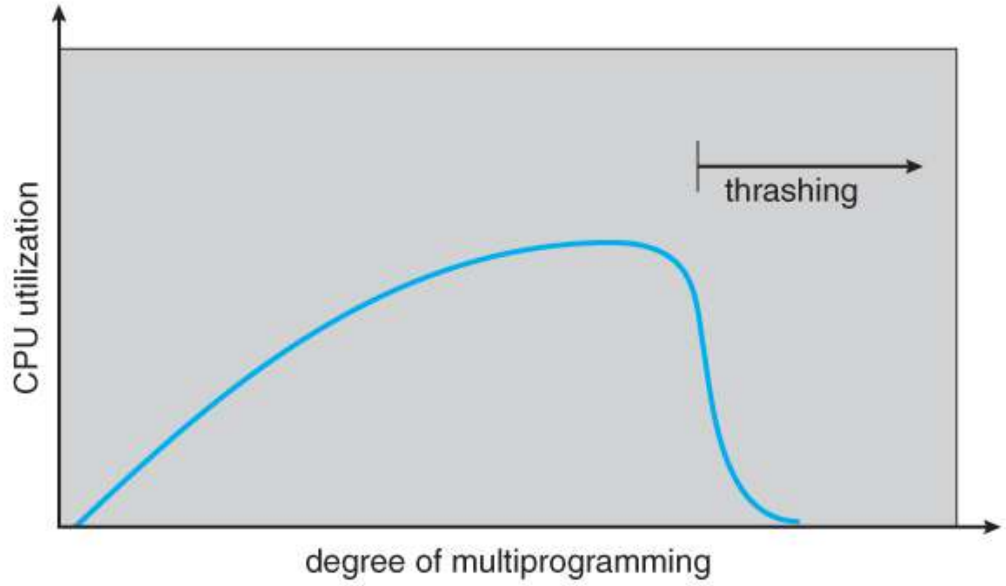
Multiprogramming and Thrashing

- When the degree of multiprogramming is too high, active working sets of running processes may saturate the whole memory capacity

Multiprogramming and Thrashing

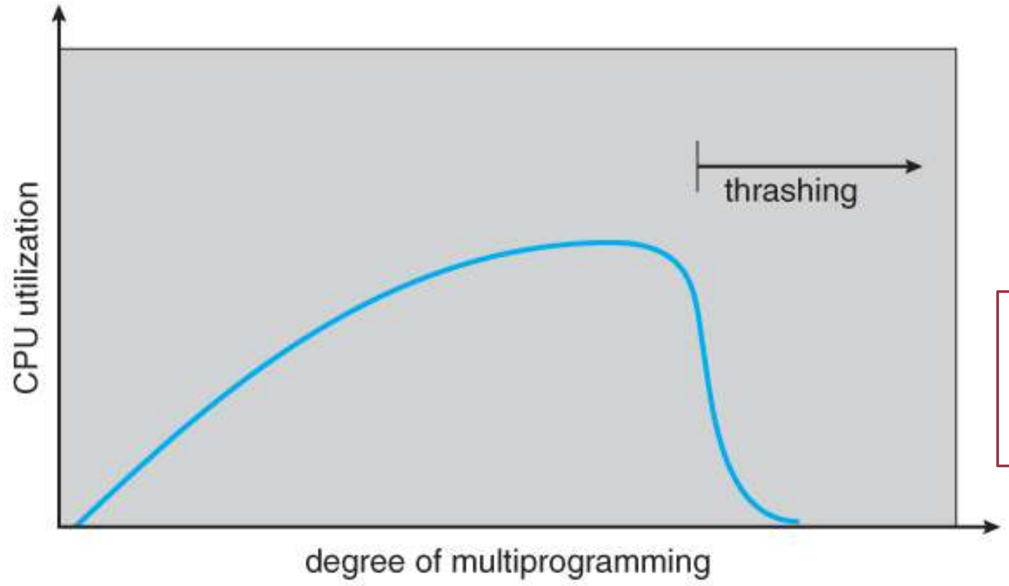
- When the degree of multiprogramming is too high, active working sets of running processes may saturate the whole memory capacity
- Thrashing → Memory is over-committed and pages are continuously tossed out while they are still in use
  - Memory access time approaches disk access time due to many page faults
  - Drastic degradation of performance

Multiprogramming and Thrashing



CPU utilization drops after a certain degree of multiprogramming

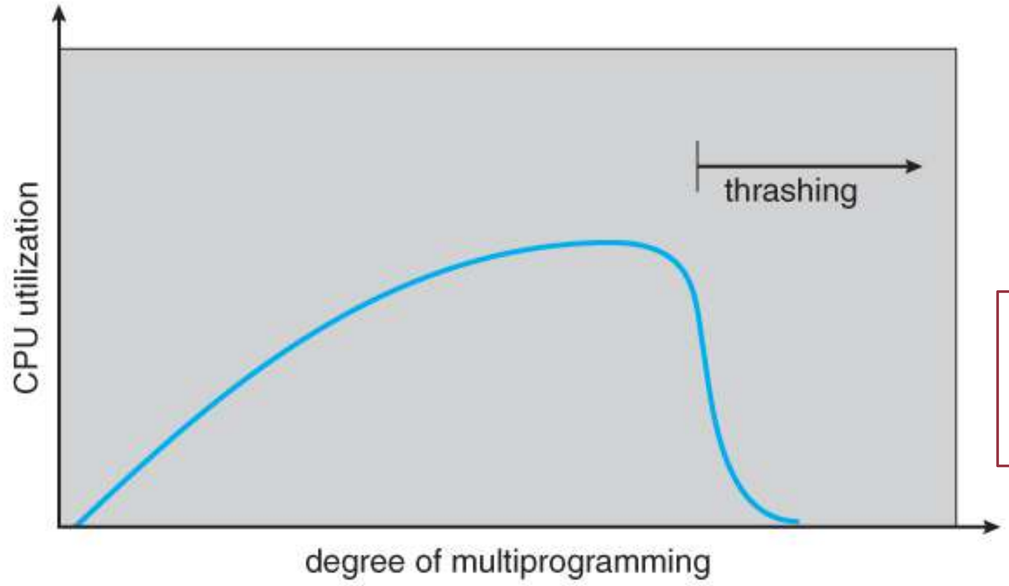
Multiprogramming and Thrashing



CPU utilization drops after a certain degree of multiprogramming

Eventually, also CPU-bound processes turn into I/O-bound ones (due to page faults)

Multiprogramming and Thrashing

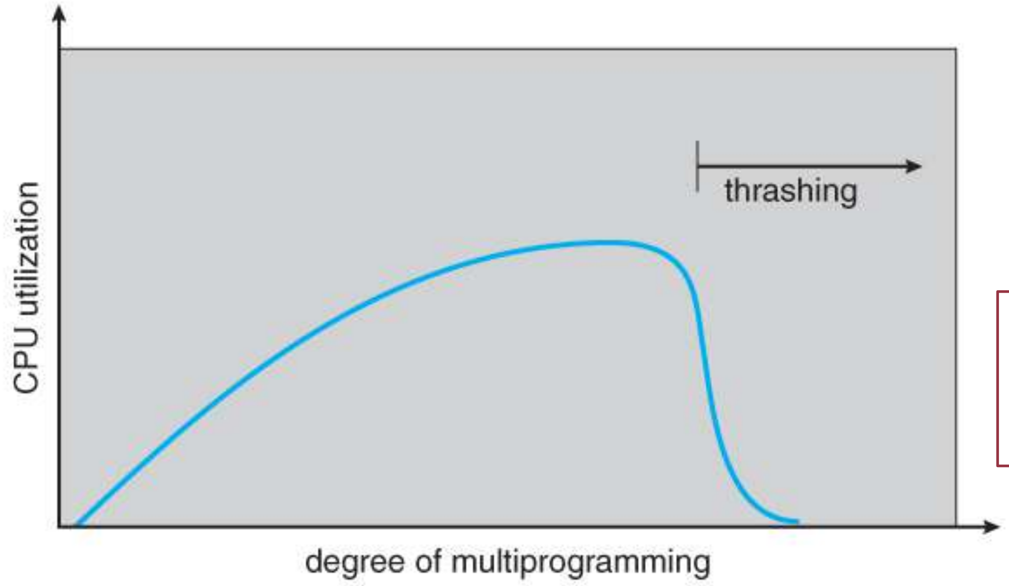


CPU utilization drops after a certain degree of multiprogramming

Eventually, also CPU-bound processes turn into I/O-bound ones (due to page faults)

What can we do to limit thrashing in a multiprogrammed system?

Multiprogramming and Thrashing



CPU utilization drops after a certain degree of multiprogramming

Eventually, also CPU-bound processes turn into I/O-bound ones (due to page faults)

What can we do to limit thrashing in a multiprogrammed system?

Fixing the degree of multi-programming apriori may be a too inflexible option

Allocation/Replacement Policies

Ultimately, we want to give each process enough memory so as to avoid thrashing

Allocation/Replacement Policies

Ultimately, we want to give each process enough memory so as to avoid thrashing

Global Allocation/Replacement

- All pages from all processes are in a single pool (single LRU queue)
- Upon page replacement, any page may be a potential victim, whether it currently belongs to the process seeking a free frame or not
- PRO: flexibility
- CON: thrashing more likely (no isolation)

Allocation/Replacement Policies

Ultimately, we want to give each process enough memory so as to avoid thrashing

Global Allocation/Replacement

- All pages from all processes are in a single pool (single LRU queue)
- Upon page replacement, any page may be a potential victim, whether it currently belongs to the process seeking a free frame or not
- PRO: flexibility
- CON: thrashing more likely (no isolation)

Local Allocation/Replacement

- Each process has its own fixed pool of frames
- Run only group of processes that fits in memory
- LRU replacement affects only each process' frames
- PRO: isolation
- CON: performance (a process may not be given enough memory)

Local Allocation/Replacement

m = number of available physical page frames

n = number of processes

S_i = size of the i-th process; $S = \sum_{i=1}^{n} S_i$ = total size of all processes

Equal Allocation/Replacement: $\frac{m}{n}$

Proportional Allocation/Replacement: $\frac{m * S_i}{S}$

Local Allocation/Replacement

m = number of available physical page frames

n = number of processes

S_i = size of the i-th process; $S = \sum_{i=1}^{n} S_i$ = total size of all processes

Equal Allocation/Replacement: $\frac{m}{n}$

Proportional Allocation/Replacement: $\frac{m * S_i}{S}$

Variations on proportional allocation could consider priority of process rather than just their size

Local Allocation/Replacement

$m =$ number of available physical page frames

$n =$ number of processes

$S_i =$ size of the i-th process; $S = \sum_{i=1}^{n} S_i =$ total size of all processes

Equal Allocation/Replacement: $\frac{m}{n}$

Proportional Allocation/Replacement: $\frac{m * S_i}{S}$

Variations on proportional allocation could consider priority of process rather than just their size

As allocations fluctuate over time, so does *m*
(processes must either be swapped out or not allowed to start if not enough frames)

Proportional Allocation/Replacement

- This implicitly assumes that a large process will also refer to a large amount of memory

Proportional Allocation/Replacement

- This implicitly assumes that a large process will also refer to a large amount of memory
- Intuitively this makes sense: the more memory a process needs the higher the chance it will refer a much larger memory range

Proportional Allocation/Replacement

- This implicitly assumes that a large process will also refer to a large amount of memory
- Intuitively this makes sense: the more memory a process needs the higher the chance it will refer a much larger memory range
- However, there might be cases where this is not true
  - e.g., a process allocates a 1GB array but then only uses a small portion of it

Proportional Allocation/Replacement

- This implicitly assumes that a large process will also refer to a large amount of memory
- Intuitively this makes sense: the more memory a process needs the higher the chance it will refer a much larger memory range
- However, there might be cases where this is not true
    - e.g., a process allocates a 1GB array but then only uses a small portion of it
- In other words, the working set of a process may not be correlated with its memory footprint

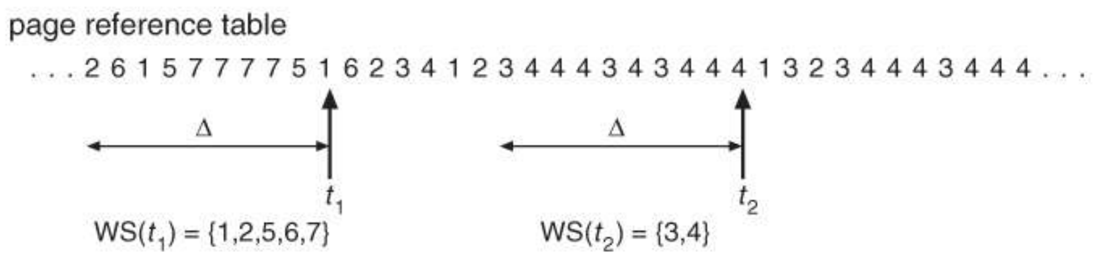
Matching the Working Set

- Goal → Give each process enough frames to contain its working set
  - Informally, the working set is the set of pages the process is using "right now"
  - More formally, it is the set of all pages that the process has referenced during the past T units of time (e.g., seconds)

Matching the Working Set

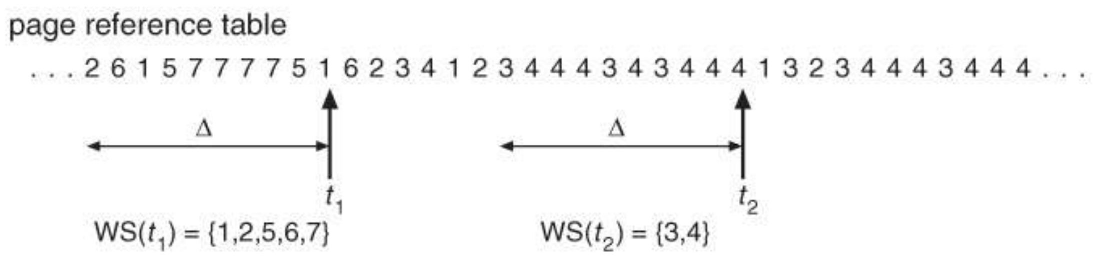
- Goal → Give each process enough frames to contain its working set
  - Informally, the working set is the set of pages the process is using "right now"
  - More formally, it is the set of all pages that the process has referenced during the past T units of time (e.g., seconds)
- How does the OS pick T?
  - 1 page fault takes order of 10 msecs to be served
  - 10 msecs ~ 10 million instructions
  - T needs to account for a lot more than 10 million instructions

Determining the Working Set



The selection of Δ is critical to the success of the working set model

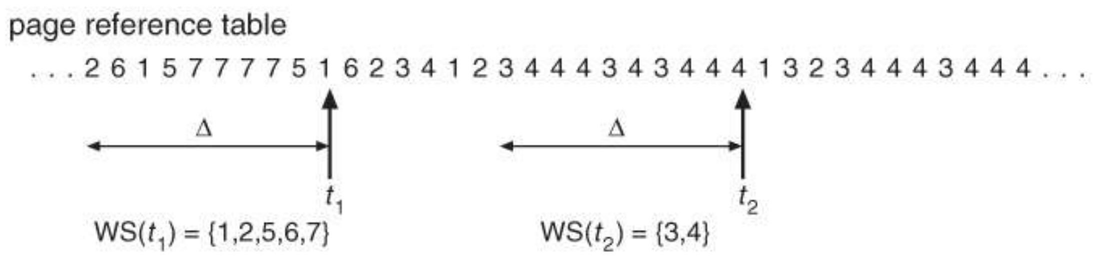
Determining the Working Set



The selection of Δ is critical to the success of the working set model

Δ too small
it does not encompass all of the pages of the current locality

Determining the Working Set

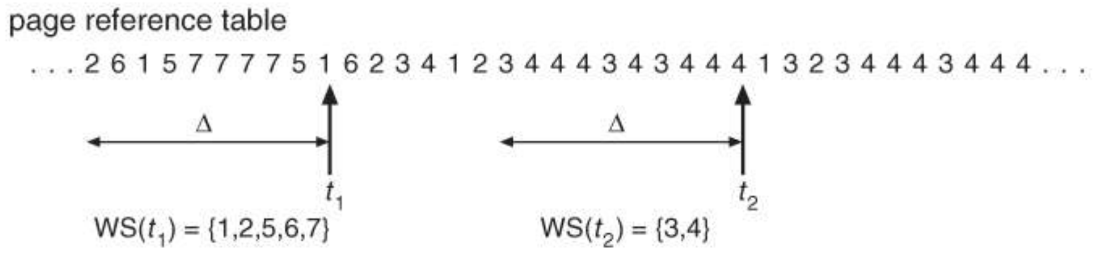


The selection of Δ is critical to the success of the working set model

Δ too small
it does not encompass all of the pages of the current locality

Δ too large
it includes pages that are no longer frequently accessed

Determining the Working Set



The selection of Δ is critical to the success of the working set model

Δ too small
it does not encompass all of the pages of the current locality

Δ too large
it includes pages that are no longer frequently accessed

Exact tracking is expensive: update the working set at each memory access

Approximating the Working Set

- Computing the working set exactly requires to keep track of a moving window of size Δ

Approximating the Working Set

- Computing the working set exactly requires to keep track of a moving window of size Δ
- At each memory reference a new reference appears at one end and the oldest reference drops off the other end

Approximating the Working Set

- Computing the working set exactly requires to keep track of a moving window of size Δ
- At each memory reference a new reference appears at one end and the oldest reference drops off the other end
- To avoid the overhead of keeping a list of the last Δ referenced pages, the working set is often implemented with sampling

Approximating the Working Set

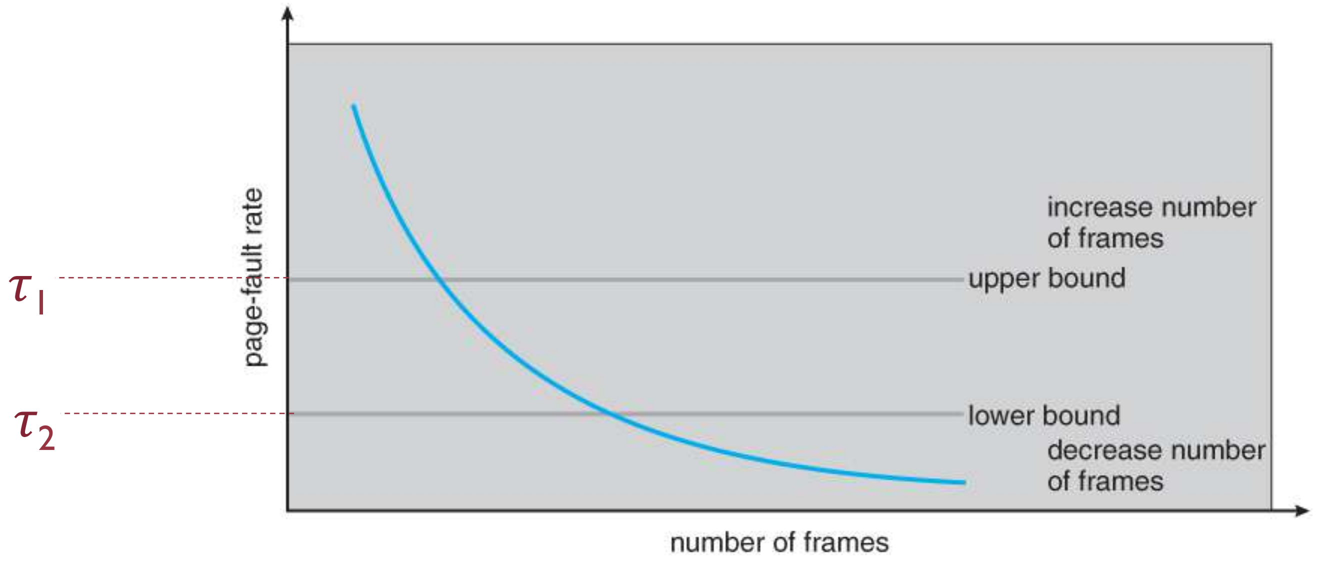
- Computing the working set exactly requires to keep track of a moving window of size Δ
- At each memory reference a new reference appears at one end and the oldest reference drops off the other end
- To avoid the overhead of keeping a list of the last Δ referenced pages, the working set is often implemented with sampling
- Every k memory references (e.g., $k = 1{,}000$), consider the working set to be all pages referenced within *that* period of time

Tracking Page Fault Rate

- Ultimately, our goal is to minimize the page fault rate

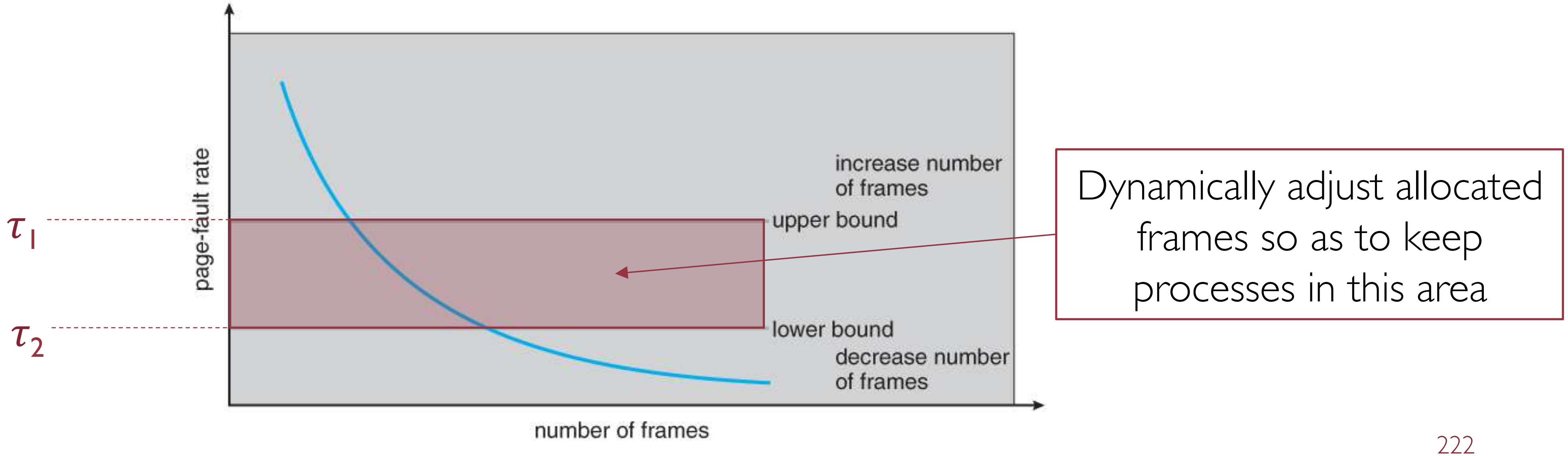
Tracking Page Fault Rate

- Ultimately, our goal is to minimize the page fault rate
- A more direct approach is to track page fault rate of each process:
  - If the page fault rate is above a given threshold τ_1 → give it more frames
  - If the page fault rate is below a given threshold τ_2 → reclaim some frames



Tracking Page Fault Rate

- Ultimately, our goal is to minimize the page fault rate
- A more direct approach is to track page fault rate of each process:
  - If the page fault rate is above a given threshold τ_1 → give it more frames
  - If the page fault rate is below a given threshold τ_2 → reclaim some frames



Kernel Memory

- So far, we only considered memory allocation for user processes
- But kernel needs memory to store things too: code and data structures like PCB, page tables, etc.
- Kernel does not use any of the advanced mechanisms seen so far
  - No paging → what if a page fault occurs for the kernel?

Kernel Memory: Buddy Allocator

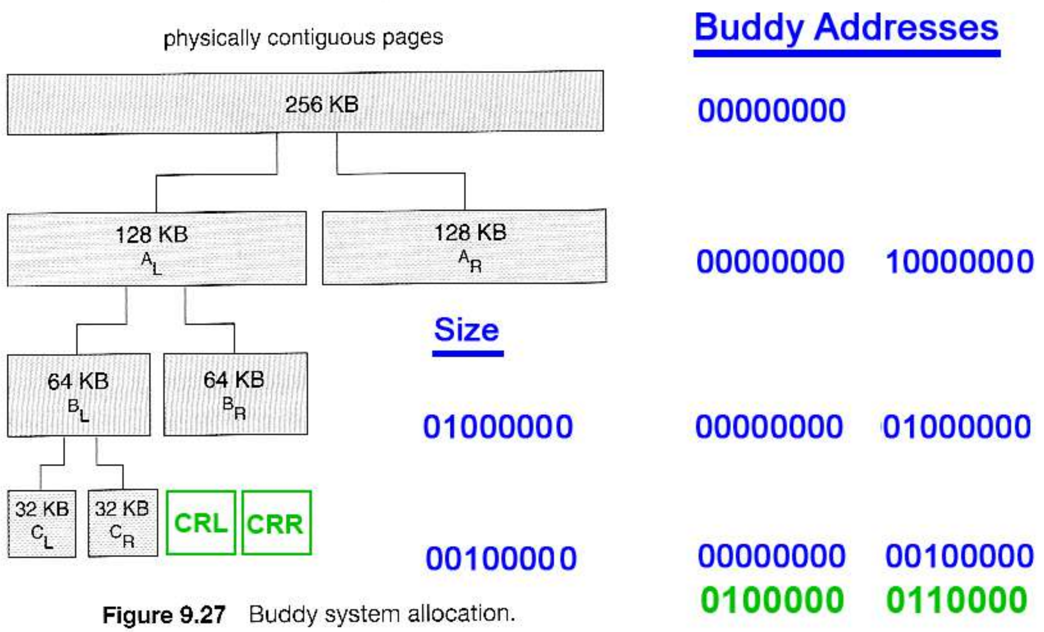
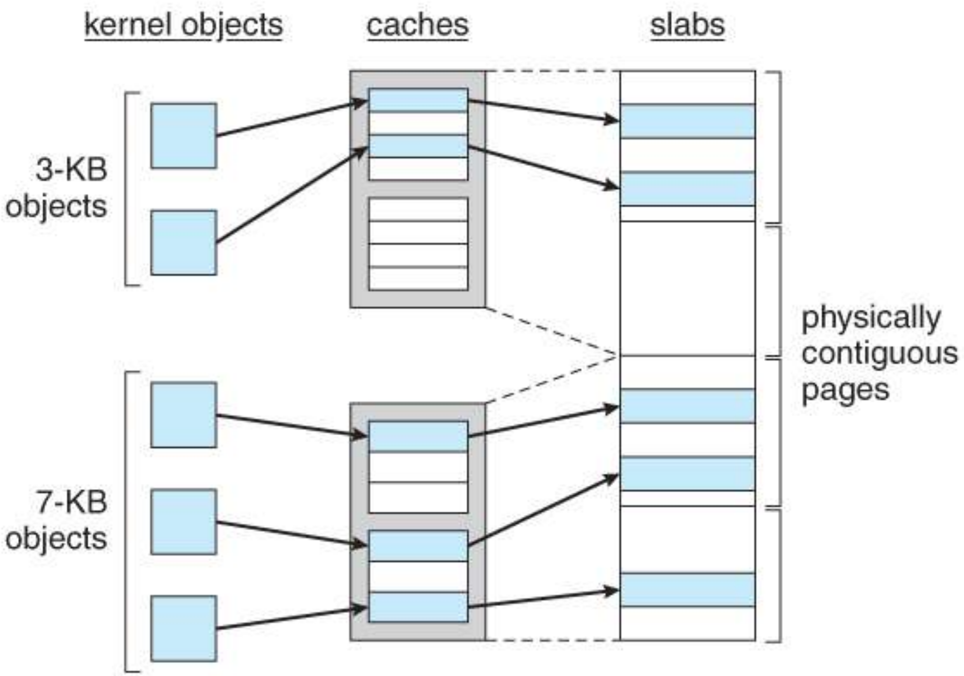


Figure 9.27 Buddy system allocation.

- Allocates memory using a power of 2 allocator (e.g., 4K, 8K, 16K), rounding up to the next nearest power of two if necessary
- If a block of the correct size is not available, then one is formed by (repeatedly) splitting the next larger block in two
- Can lead to internal fragmentation

Kernel Memory: Slab Allocator



- Group of objects of the same size in a slab
- Object cache points to one or more slabs
- Separate cache for each kernel data structure (e.g., PCB)
- No internal fragmentation
- Used in Solaris and Linux

Page Sizes

- Reasons for small pages?

Page Sizes

- Reasons for small pages?
  - Decreasing internal fragmentation
  - Higher degree of multiprogramming

Page Sizes

- Reasons for small pages?
  - Decreasing internal fragmentation
  - Higher degree of multiprogramming
- Reasons for large pages?

Page Sizes

- Reasons for small pages?
  - Decreasing internal fragmentation
  - Higher degree of multiprogramming
- Reasons for large pages?
  - Smaller page table size (i.e., smaller number of page table entries)
  - Fewer page faults (locality reference)
  - Amortizes disk overhead (reading a1KB page from disk takes approximately the same as reading an 8KB one)

Summary of Page Replacement

- The choice of page replacement algorithm is crucial when physical memory is limited
  - All algorithms approach to the optimum as the physical memory allocated to a process approaches to the virtual memory size

Summary of Page Replacement

- The choice of page replacement algorithm is crucial when physical memory is limited
  - All algorithms approach to the optimum as the physical memory allocated to a process approaches to the virtual memory size
- The more processes running concurrently, the less physical memory each one can have

Summary of Page Replacement

- The choice of page replacement algorithm is crucial when physical memory is limited
  - All algorithms approach to the optimum as the physical memory allocated to a process approaches to the virtual memory size
- The more processes running concurrently, the less physical memory each one can have
- The OS must choose how many processes (and the number of frames per process) can share memory simultaneously