Virtual Memory (Part I)

Amir H. Payberah
amir@sics.se

Amirkabir University of Technology
(Tehran Polytechnic)

Motivation and Background

Motivation

- A program needs to be in memory to execute.
- But, entire program rarely used.
  - Error code, unusual routines, large data structures
- Entire program code not needed at same time.

Motivation

- Consider ability to execute partially-loaded program.

Motivation

- Consider ability to execute partially-loaded program.
- Program no longer constrained by limits of physical memory.

Motivation

- Consider ability to execute partially-loaded program.
- Program no longer constrained by limits of physical memory.
- Each program takes less memory while running: more programs run at the same time.
  - Increased CPU utilization and throughput with no increase in response time or turnaround time.

Motivation

- Consider ability to execute partially-loaded program.
- Program no longer constrained by limits of physical memory.
- Each program takes less memory while running: more programs run at the same time.
  - Increased CPU utilization and throughput with no increase in response time or turnaround time.
- Less I/O needed to load or swap programs into memory: each user program runs faster.

Virtual Memory

Virtual Memory (1/2)

- Separation of user logical memory from physical memory.

Virtual Memory (1/2)

- Separation of user logical memory from physical memory.
- Only part of the program needs to be in memory for execution.

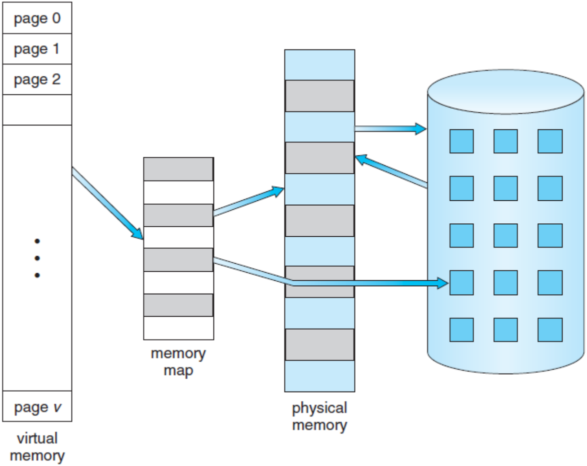
Virtual Memory (1/2)

- Separation of user logical memory from physical memory.
- Only part of the program needs to be in memory for execution.
- Logical address space can therefore be much larger than physical address space.

Virtual Memory (1/2)

- Separation of user logical memory from physical memory.
- Only part of the program needs to be in memory for execution.
- Logical address space can therefore be much larger than physical address space.
- More programs running concurrently.

Virtual Memory (2/2)



Virtual Address Space (1/3)

- Virtual address space: logical view of how process is stored in memory.

Virtual Address Space (1/3)

- Virtual address space: logical view of how process is stored in memory.
- Usually starts at address 0, contiguous addresses until end of space.

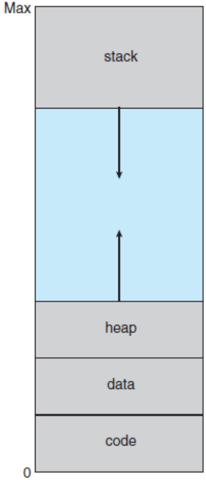
Virtual Address Space (1/3)

- Virtual address space: logical view of how process is stored in memory.
- Usually starts at address 0, contiguous addresses until end of space.
- Meanwhile, physical memory organized in page frames.

Virtual Address Space (1/3)

- Virtual address space: logical view of how process is stored in memory.
- Usually starts at address 0, contiguous addresses until end of space.
- Meanwhile, physical memory organized in page frames.
- MMU must map logical to physical.

Virtual Address Space (2/3)



Virtual Address Space (3/3)

- The heap grows upward in memory, and the stack grows downward.

Virtual Address Space (3/3)

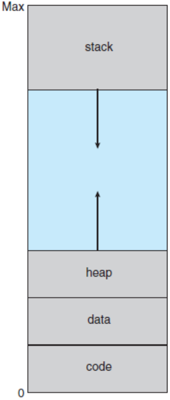
- The heap grows upward in memory, and the stack grows downward.
- The hole between the heap and the stack is part of the virtual address space, but will require actual physical pages only if the heap or stack grows.

Virtual Address Space (3/3)

- The heap grows upward in memory, and the stack grows downward.
- The hole between the heap and the stack is part of the virtual address space, but will require actual physical pages only if the heap or stack grows.
- Virtual address spaces that include holes are known as sparse address spaces.

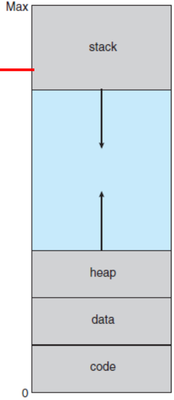
Heap vs. Stack

- **Stack**
  - Stores local variables created by functions.
  - New variables are pushed onto the stack.
  - When a function exits, all of the its pushed varial are freed.



Heap vs. Stack

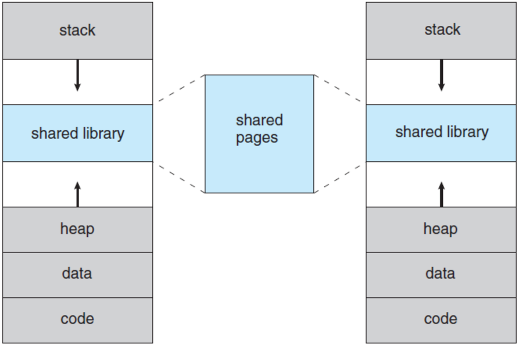
- **Stack**
  - Stores local variables created by functions.
  - New variables are pushed onto the stack.
  - When a function exits, all of the its pushed varial are freed.
- **Heap**
  - Used for dynamic allocation.
  - Use `malloc()` or `calloc()` to allocate memory on the heap.
  - Use `free()` to deallocate the memory.



Virtual Memory Benefits

- Separating logical memory from physical memory.

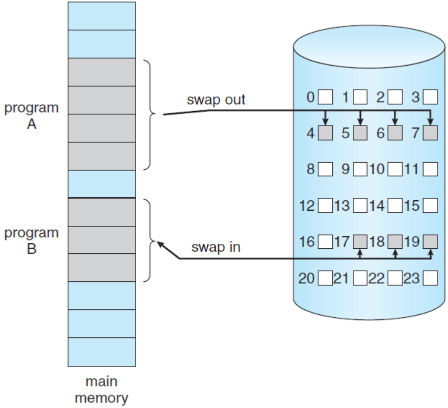
- Page sharing.



Demand Paging

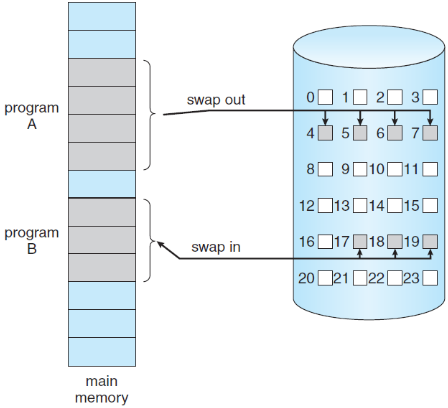
Demand Paging (1/2)

- Could bring entire process into memory at load time, or bring a page into memory only when it is needed.



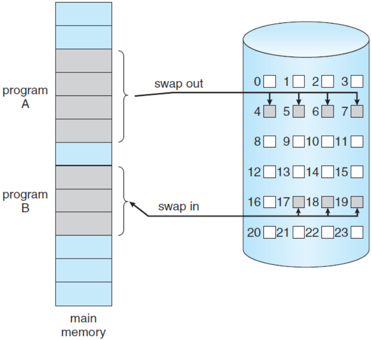
Demand Paging (1/2)

- Could bring entire process into memory at load time, or bring a page into memory only when it is needed.
- A demand-paging system is similar to a paging system with swapping.



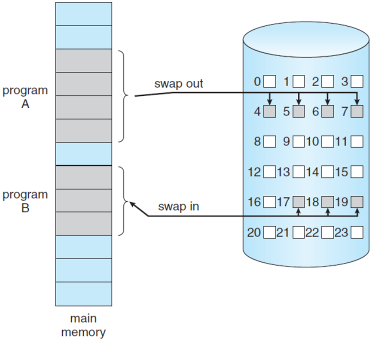
Demand Paging (2/2)

- Rather than swapping the entire process into memory, we use a lazy swapper.
- A lazy swapper never swaps a page into memory unless that page will be needed.



Demand Paging (2/2)

- Rather than swapping the entire process into memory, we use a lazy swapper.
- A lazy swapper never swaps a page into memory unless that page will be needed.
- A swapper that deals with pages is a pager.



Basic Concepts

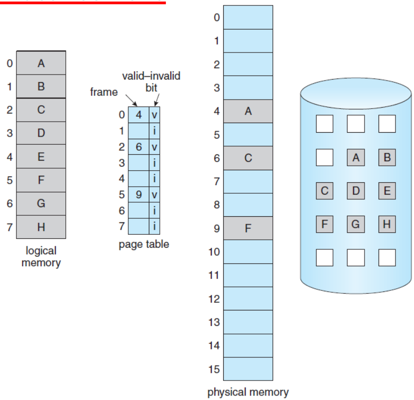
- The pager guesses which pages will be used before swapping out again.

Basic Concepts

- The pager guesses which pages will be used before swapping out again.
- The pager brings only the needed pages into memory.

Basic Concepts

- The pager guesses which pages will be used before swapping out again.
- The pager brings only the needed pages into memory.
- Uses valid-invalid bit to distinguish between the pages that are in memory and the pages that are on the disk?
  v: memory resident
  i: not in memory

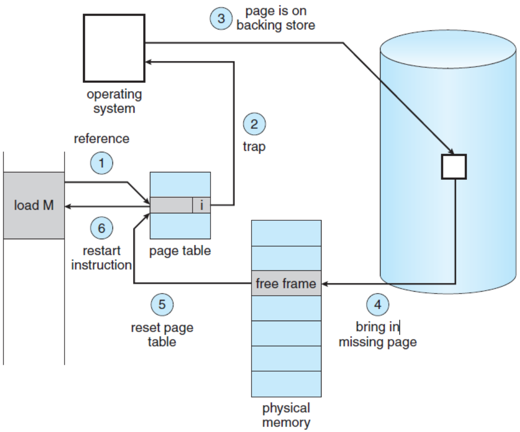


Page Fault

- Access to a page marked invalid causes a page fault.
- Causing a trap to the OS: brings the desired page into memory.

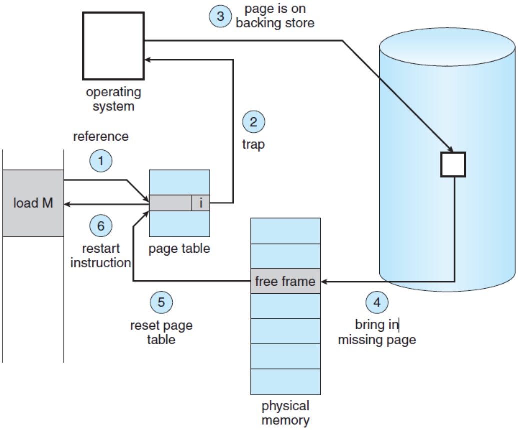
Handling Page Fault (1/6)

- Check an internal table for the process to determine whether the reference was a valid or an invalid memory access.



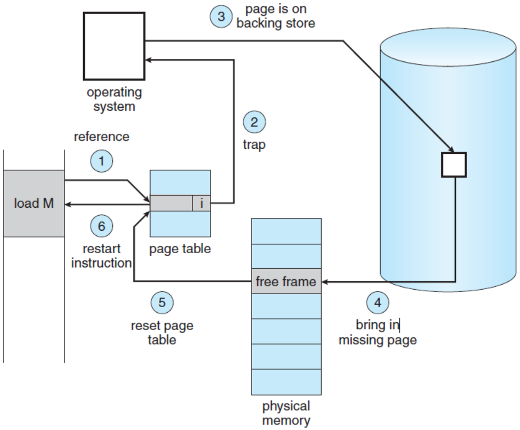
Handling Page Fault (2/6)

- If the reference was invalid, we terminate the process.
- If it was valid but we have not yet brought in that page, we now page it in.



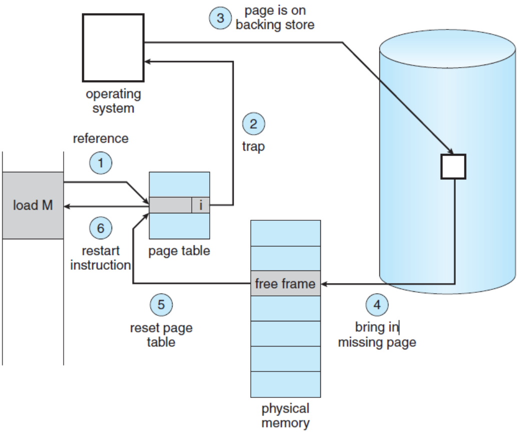
Handling Page Fault (3/6)

- We find a free frame.



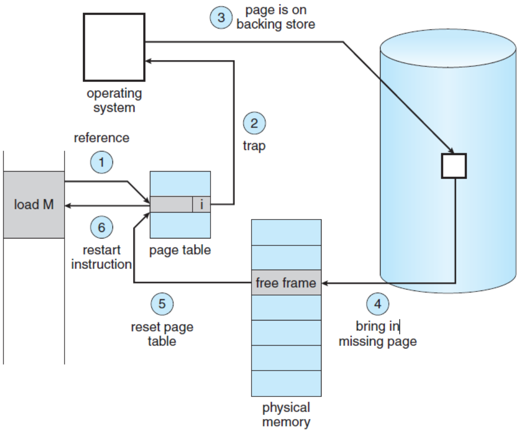
Handling Page Fault (4/6)

- We schedule a disk operation to read the desired page into the newly allocated frame.



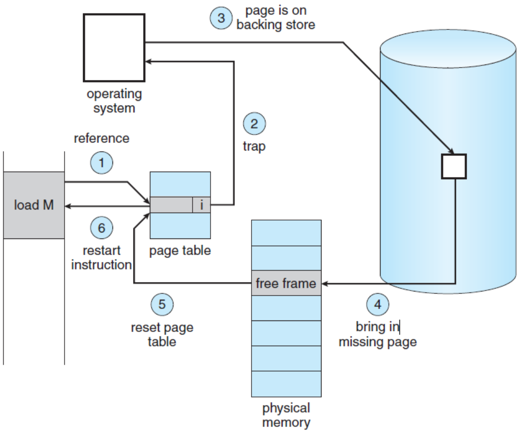
Handling Page Fault (5/6)

- When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.



Handling Page Fault (6/6)

- We restart the instruction that was interrupted by the trap.



Aspects of Demand Paging (1/2)

- Extreme case: start process with no pages in memory.

Aspects of Demand Paging (1/2)

- Extreme case: start process with no pages in memory.
- OS sets instruction pointer to first instruction of process, non-memory-resident: page fault

Aspects of Demand Paging (1/2)

- Extreme case: start process with no pages in memory.
- OS sets instruction pointer to first instruction of process, non-memory-resident: page fault
- And for every other process pages on first access.

Aspects of Demand Paging (1/2)

- Extreme case: start process with no pages in memory.
- OS sets instruction pointer to first instruction of process, non-memory-resident: page fault
- And for every other process pages on first access.
- Pure demand paging

Aspects of Demand Paging (2/2)

- A program could access several new pages of memory with each instruction execution: multiple page faults per instruction.

Aspects of Demand Paging (2/2)

- A program could access several new pages of memory with each instruction execution: multiple page faults per instruction.

- Unacceptable system performance

Aspects of Demand Paging (2/2)

- A program could access several new pages of memory with each instruction execution: multiple page faults per instruction.

- Unacceptable system performance

?

- Locality of reference results in reasonable performance from demand paging.

Demand Paging Hardware

- The hardware to support demand paging is the same as the hardware for paging and swapping:
  - Page table with valid-invalid bit
  - Secondary memory with swap space

- The ability to restart any instruction after a page fault.

Stages in Demand Paging (1/3)

- 1. Trap to the OS.

Stages in Demand Paging (1/3)

- 1. Trap to the OS.
- 2. Save the user registers and process state.

Stages in Demand Paging (1/3)

- 1. Trap to the OS.
- 2. Save the user registers and process state.
- 3. Determine that the interrupt was a page fault.

Stages in Demand Paging (1/3)

- 1. Trap to the OS.
- 2. Save the user registers and process state.
- 3. Determine that the interrupt was a page fault.
- 4. Check that the page reference was legal and determine the location of the page on the disk.

Stages in Demand Paging (2/3)

- 5. Issue a read from the disk to a free frame:
  - Wait in a queue for this device until the read request is serviced.
  - Wait for the device seek and/or latency time.
  - Begin the transfer of the page to a free frame.

Stages in Demand Paging (2/3)

- 5. Issue a read from the disk to a free frame:
  - Wait in a queue for this device until the read request is serviced.
  - Wait for the device seek and/or latency time.
  - Begin the transfer of the page to a free frame.
- 6. While waiting, allocate the CPU to some other user.

Stages in Demand Paging (2/3)

- 5. Issue a read from the disk to a free frame:
  - Wait in a queue for this device until the read request is serviced.
  - Wait for the device seek and/or latency time.
  - Begin the transfer of the page to a free frame.
- 6. While waiting, allocate the CPU to some other user.
- 7. Receive an interrupt from the disk I/O subsystem (I/O completed).

Stages in Demand Paging (3/3)

- 8. Save the registers and process state for the other user.

Stages in Demand Paging (3/3)

- 8. Save the registers and process state for the other user.
- 9. Determine that the interrupt was from the disk.

Stages in Demand Paging (3/3)

- 8. Save the registers and process state for the other user.
- 9. Determine that the interrupt was from the disk.
- 10. Correct the page table and other tables to show page is now in memory.

Stages in Demand Paging (3/3)

- 8. Save the registers and process state for the other user.
- 9. Determine that the interrupt was from the disk.
- 10. Correct the page table and other tables to show page is now in memory.
- 11. Wait for the CPU to be allocated to this process again.

Stages in Demand Paging (3/3)

- 8. Save the registers and process state for the other user.
- 9. Determine that the interrupt was from the disk.
- 10. Correct the page table and other tables to show page is now in memory.
- 11. Wait for the CPU to be allocated to this process again.
- 12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction.

Performance of Demand Paging

- Probability of a page fault: $0 \leq p \leq 1$

Performance of Demand Paging

- Probability of a page fault: $0 \leq p \leq 1$
- If $p = 0$: no page faults
- If $p = 1$: every reference is a fault

Performance of Demand Paging

- Probability of a page fault: $0 \leq p \leq 1$
- If $p = 0$: no page faults
- If $p = 1$: every reference is a fault
- Effective Access Time (EAT)
- $EAT = (1 - p) \times memory\_access + p \times page\_fault\_time$

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- $EAT = (1 - p) \times 200 + p \times 8,000,000 = 200 + p \times 7,999,800$

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- $EAT = (1 - p) \times 200 + p \times 8,000,000 = 200 + p \times 7,999,800$
- If $p = 0.001$, then $EAT = 8.2 microseconds$: this is a slowdown by a factor of 40.

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- $EAT = (1 - p) \times 200 + p \times 8,000,000 = 200 + p \times 7,999,800$
- If $p = 0.001$, then $EAT = 8.2 microseconds$: this is a slowdown by a factor of 40.
- If want performance degradation < 10 percent:
  - $220 > 200 + p \times 7,999,800$
  - $20 > p \times 7,999,800$
  - $p < .0000025$: less than one page fault in every 400,000 memory accesses

Demand Paging Optimization

- Swap space I/O faster than file system I/O even if on the same device.
    - Swap allocated in larger chunks, less management needed than file system.

Demand Paging Optimization

- Swap space I/O faster than file system I/O even if on the same device.
  - Swap allocated in larger chunks, less management needed than file system.
- Copy entire process image to swap space at process load time.
  - Then page in and out of swap space.

Demand Paging Optimization

- Swap space I/O faster than file system I/O even if on the same device.
  - Swap allocated in larger chunks, less management needed than file system.
- Copy entire process image to swap space at process load time.
  - Then page in and out of swap space.
- Demand page in from program binary on disk, but discard rather than paging out when freeing frame.
  - Pages not associated with a file, i.e., heap and stack (anonymous memory) till need to write to swap space.

?

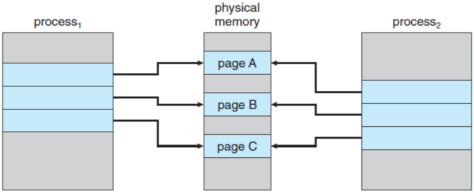
Copy-on-Write

Copy-on-Write

- Copy-on-Write allows both parent and child processes to initially share the same pages in memory.

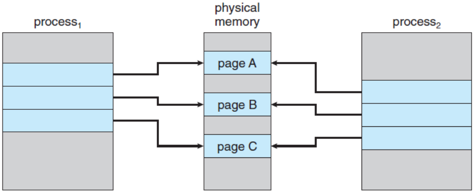
- If either process modifies a shared page, only then is the page copied.

Copy-on-Write Example

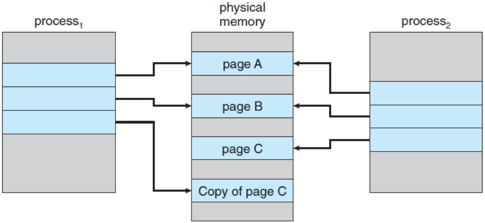


Before modification

Copy-on-Write Example



Before modification



After modification

More About Copy-on-Write

- Copy-on-Write allows more efficient process creation as only modified pages are copied.

More About Copy-on-Write

- Copy-on-Write allows more efficient process creation as only modified pages are copied.

- Free pages are allocated from a pool of zero-fill-on-demand pages.
  - Pool should always have free frames for fast demand page execution.
  - Zero-fill-on-demand pages have been zeroed-out before being allocated, thus erasing the previous contents.

More About Copy-on-Write

- Copy-on-Write allows more efficient process creation as only modified pages are copied.
- Free pages are allocated from a pool of zero-fill-on-demand pages.
  - Pool should always have free frames for fast demand page execution.
  - Zero-fill-on-demand pages have been zeroed-out before being allocated, thus erasing the previous contents.

?

- `vfork()` variation on `fork()` does not use Copy-on-Write: with `vfork()`, the parent process is suspended, and the child process uses the address space of the parent.

Page Replacement

What Happens if There is no Free Frame?

- Assume, we had forty frames in physical memory.

- And, we run six processes, each of which is ten pages in size, but actually uses only five pages.

What Happens if There is no Free Frame?

- Assume, we had forty frames in physical memory.

- And, we run six processes, each of which is ten pages in size, but actually uses only five pages.

- It is possible that each of these processes may suddenly try to use all ten of its pages: resulting in a need for sixty frames when only forty are available.

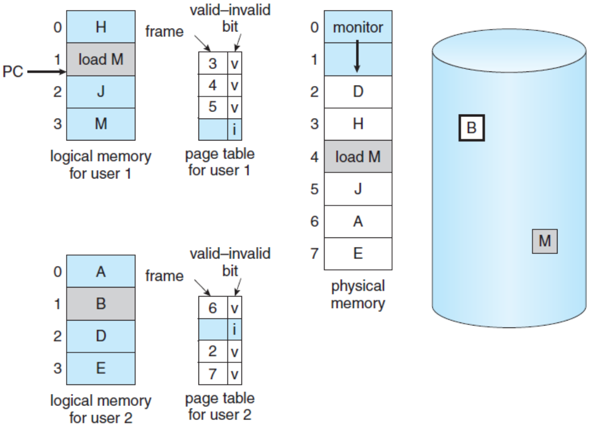
What Happens if There is no Free Frame?

- Assume, we had forty frames in physical memory.
- And, we run six processes, each of which is ten pages in size, but actually uses only five pages.
- It is possible that each of these processes may suddenly try to use all ten of its pages: resulting in a need for sixty frames when only forty are available.
- Increasing the degree of multiprogramming: over-allocating memory

Over-Allocation of Memory

- While a user process is executing, a page fault occurs.
- The OS determines where the desired page is residing on the disk.
- But, it finds that there are no free frames on the free-frame list.
- Need for page replacement

Need For Page Replacement



Basic Page Replacement

1. Find the location of the desired page on disk.

Basic Page Replacement

1. Find the location of the desired page on disk.

2. Find a free frame.
   - If there is a free frame, use it.
   - If there is no free frame, use a page replacement algorithm to select a victim frame: write victim frame to disk if dirty.

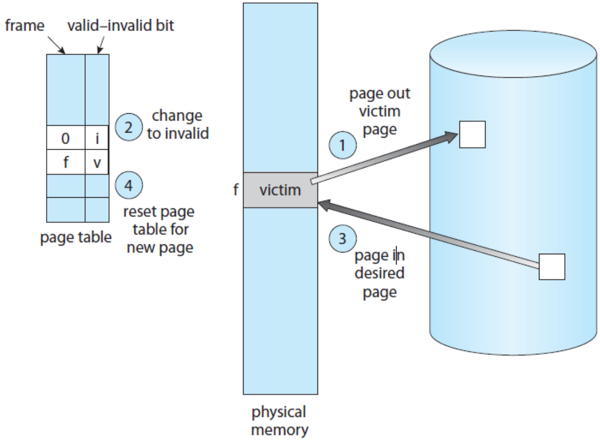
Basic Page Replacement

1. Find the location of the desired page on disk.
2. Find a free frame.
   - If there is a free frame, use it.
   - If there is no free frame, use a page replacement algorithm to select a victim frame: write victim frame to disk if dirty.
3. Bring the desired page into the (newly) free frame; update the page and frame tables

Basic Page Replacement

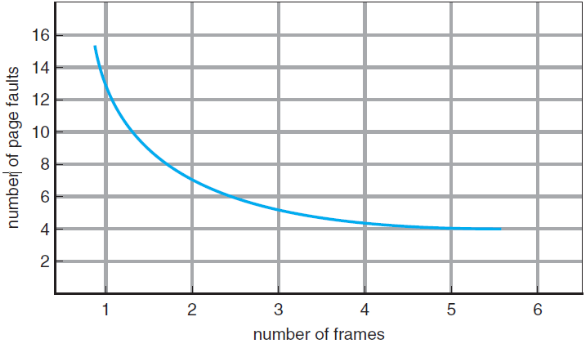
1. Find the location of the desired page on disk.
2. Find a free frame.
   - If there is a free frame, use it.
   - If there is no free frame, use a page replacement algorithm to select a victim frame; write victim frame to disk if dirty.
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap.

Page Replacement



- Use modify (dirty) bit to reduce overhead of page transfers - only modified pages are written to disk.

Page Faults vs. The Number of Frames



Evaluate Page Replacement Algorithms

- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string.

- String is just page numbers, not full addresses.

- For example, a reference string could be
  7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

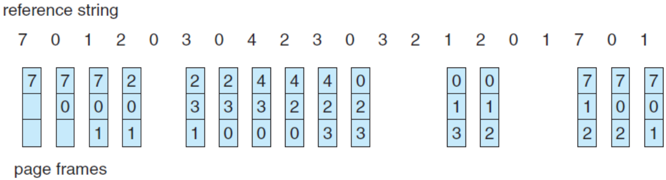
Page Replacement Algorithms

- First-In-First-Out (FIFO) page replacement
- Optimal page replacement
- Least Recently Used (LRU) page replacement
- LRU-Approximation page replacement
- Counting-Based page replacement

FIFO Page Replacement

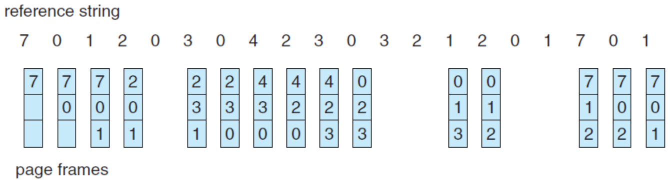
FIFO Page Replacement

- Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1
- 3 frames (3 pages can be in memory at a time per process)



FIFO Page Replacement

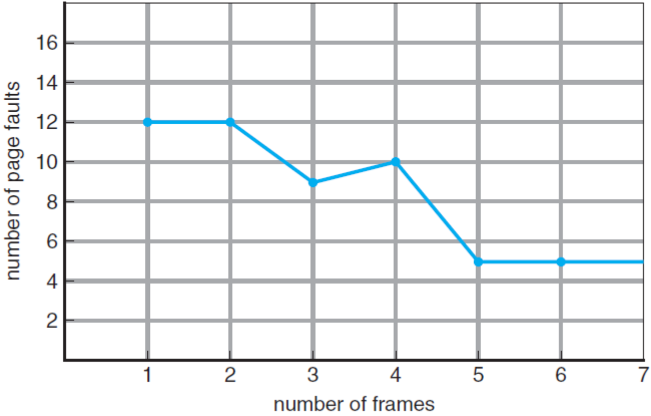
- Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1
- 3 frames (3 pages can be in memory at a time per process)



- 15 page faults

FIFO Belady's Anomaly

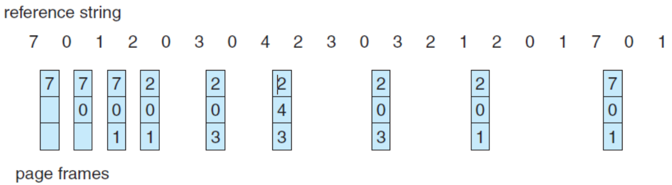
- Adding more frames can cause more page faults: Belady's Anomaly



Optimal Page Replacement

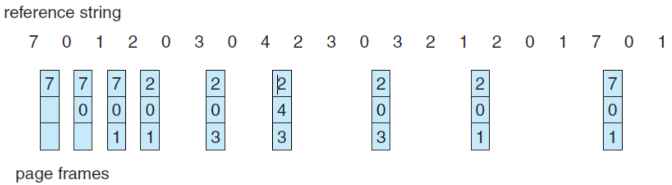
Optimal Page Replacement

- Replace page that will not be used for longest period of time: 9 page fault is optimal for the example.
- How do you know this? Can't read the future



Optimal Page Replacement

- Replace page that will not be used for longest period of time: 9 page fault is optimal for the example.
- How do you know this? Can't read the future

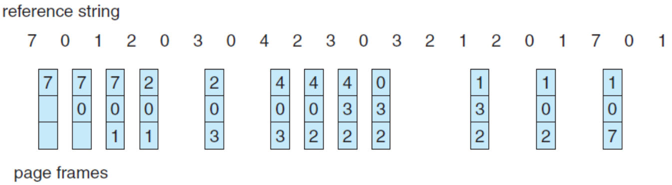


- Used for measuring how well your algorithm performs.

LRU Page Replacement

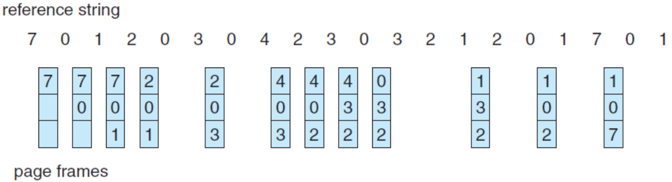
LRU Page Replacement

- Use past knowledge rather than the future.
- Replace page that has not been used in the most amount of time



LRU Page Replacement

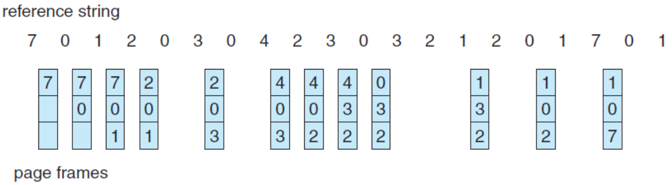
- Use past knowledge rather than the future.
- Replace page that has not been used in the most amount of time



- 12 faults: better than FIFO but worse than OPT

LRU Page Replacement

- Use past knowledge rather than the future.
- Replace page that has not been used in the most amount of time



- 12 faults: better than FIFO but worse than OPT
- Generally good algorithm and frequently used

LRU Implementation (1/2)

- Counter implementation

- Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter.

- When a page needs to be changed, look at the counters to find smallest value.

- Search through table needed.

LRU Implementation (2/2)

- Stack implementation
- Keep a stack of page numbers in a double link form.

LRU Implementation (2/2)

- Stack implementation
- Keep a stack of page numbers in a double link form.
- Page referenced:
  - Move it to the top
  - Requires 6 pointers to be changed

LRU Implementation (2/2)

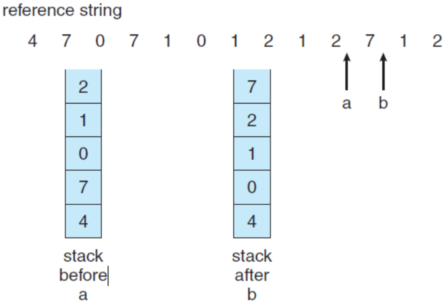
- Stack implementation
- Keep a stack of page numbers in a double link form.
- Page referenced:
  - Move it to the top
  - Requires 6 pointers to be changed
- No search for replacement.

LRU Implementation (2/2)

- Stack implementation
- Keep a stack of page numbers in a double link form.
- Page referenced:
  - Move it to the top
  - Requires 6 pointers to be changed
- No search for replacement.
- LRU and OPT are cases of stack algorithms with no Belady's Anomaly.

Stack Implementation

- Use of a stack to record most recent page references.



LRU-Approximation Page Replacement

LRU-Approximation Page Replacement

- LRU needs special hardware and still slow
- Improvements: LRU-Approximation
  - Reference bit
  - Second-chance algorithm
  - Enhanced second-chance algorithm

Reference Bit

- With each page associate a bit, initially $= 0$

Reference Bit

- With each page associate a bit, initially $= 0$
- When page is referenced, bit set to 1

Reference Bit

- With each page associate a bit, initially $= 0$
- When page is referenced, bit set to 1
- Replace any with reference bit $= 0$ (if one exists)

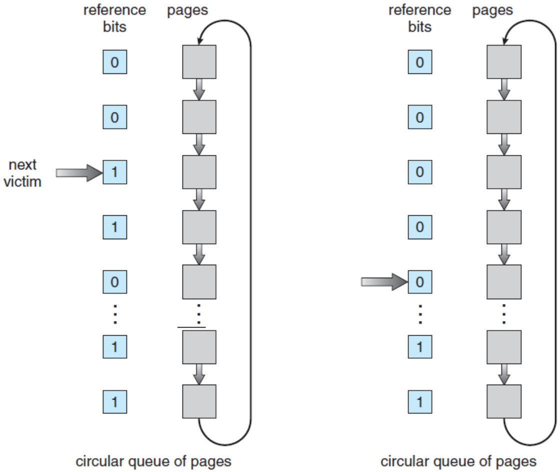
Reference Bit

- With each page associate a bit, initially $= 0$
- When page is referenced, bit set to 1
- Replace any with reference bit $= 0$ (if one exists)
- We do not know the order

Second-Chance Algorithm (1/2)

- It is also called clock algorithm.
- Generally FIFO, plus hardware-provided reference bit
- If page to be replaced has
  - Reference bit $= 0 \rightarrow$ replace it
  - Reference bit $= 1$ then, set reference bit 0, leave page in memory, and replace next page, subject to same rules.

Second-Chance Algorithm (2/2)



Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (reference, modify)

Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (reference, modify)
- Take ordered pair:
  1. (0, 0) neither recently used not modified: best page to replace
  2. (0, 1) not recently used but modified: not quite as good, must write out before replacement
  3. (1, 0) recently used but clean: probably will be used again soon
  4. (1, 1) recently used and modified: probably will be used again soon and need to write out before replacement

Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (reference, modify)
- Take ordered pair:
  1. (0, 0) neither recently used not modified: best page to replace
  2. (0, 1) not recently used but modified: not quite as good, must write out before replacement
  3. (1, 0) recently used but clean: probably will be used again soon
  4. (1, 1) recently used and modified: probably will be used again soon and need to write out before replacement
- When page replacement called for, use the clock scheme but use the four classes replace page in lowest non-empty class

Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (reference, modify)
- Take ordered pair:
  1. (0, 0) neither recently used not modified: best page to replace
  2. (0, 1) not recently used but modified: not quite as good, must write out before replacement
  3. (1, 0) recently used but clean: probably will be used again soon
  4. (1, 1) recently used and modified: probably will be used again soon and need to write out before replacement
- When page replacement called for, use the clock scheme but use the four classes replace page in lowest non-empty class
- Might need to search circular queue several times.

Counting Page Replacement

Counting Page Replacement

- Keep a counter of the number of references that have been made to each page.

Counting Page Replacement

- Keep a counter of the number of references that have been made to each page.

- Lease Frequently Used (LFU) algorithm: replaces page with smallest count.

Counting Page Replacement

- Keep a counter of the number of references that have been made to each page.
- Lease Frequently Used (LFU) algorithm: replaces page with smallest count.
- Most Frequently Used (MFU) algorithm: based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

Summary

Summary

- Partially-loaded programs

Summary

- Partially-loaded programs
- Virtual memory: much larger than physical memory

Summary

- Partially-loaded programs
- Virtual memory: much larger than physical memory
- Demand paging similar to paging + swapping

Summary

- Partially-loaded programs
- Virtual memory: much larger than physical memory
- Demand paging similar to paging + swapping
- Page fault

Summary

- Partially-loaded programs
- Virtual memory: much larger than physical memory
- Demand paging similar to paging + swapping
- Page fault
- Page replacement algorithms:
  - FIFO, optimal, LRU, LRU-approximate, counting-based

Questions?

Acknowledgements

Some slides were derived from Avi Silberschatz slides.