

# CSC 539: Operating Systems Structure and Design

Spring 2006

## Virtual memory

- background
- demand paging
- page replacement: FIFO, OPT, LRU
- allocation of frames
- thrashing
- OS examples

1

## Virtual memory

memory management allows us a logical view of memory that is consistent with the program creation process

previously, we discussed techniques for managing multiple processes in physical memory (continuous allocation, paging, segmentation)

- assumed each process would be provided sufficient physical memory
- brief mention of swapping processes in and out, but considered too costly

note: at any given time, only 1 instruction is being executed

- with paging, only 1 instruction page is required to be in physical memory
- may also require 1 or more data pages

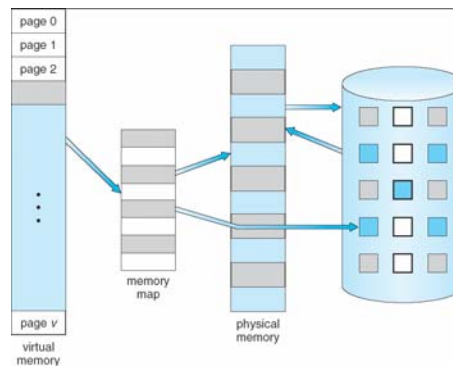
not requiring an entire process to be in physical memory → virtual memory

2

## Virtual space > physical space

logical address space can be much larger than physical address space

- an application is allocated a large virtual address space
- multiple jobs can share the same physical space, each with its own larger virtual space
- requires (hardware) translation of virtual address to physical address
- the virtual address space can be continuous or segmented

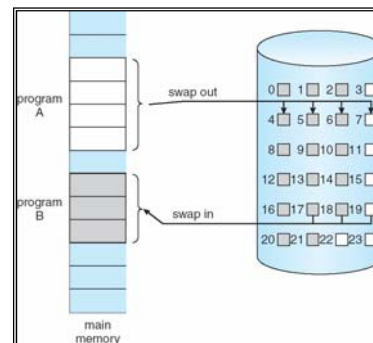


3

## Demand paging

virtual memory is commonly implemented by *demand paging*

- essentially, a paging system with swapping
- processes reside in secondary memory
- when a process is to execute, the pager determines which pages are needed and swaps them into physical memory (not necessarily the entire process)
- if not enough physical memory available, may need to swap out other pages
- ideally, the pager avoids reading into memory those pages that are not needed
  - decreased swap time
  - less memory needed / more users
  - faster response time



4

## Valid-Invalid bit

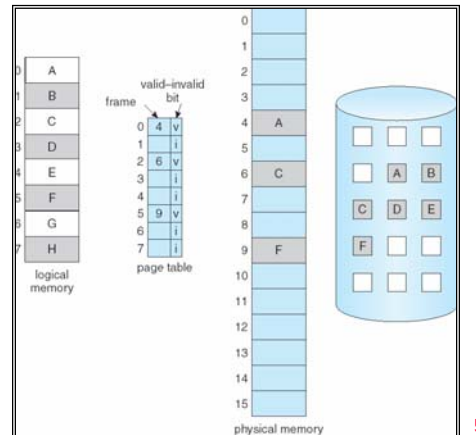
*pure* demand paging brings logical pages into physical memory only when they are actually referenced or used

- need hardware support to determine when page is in physical memory, and react gracefully when not
- in practice, some pages will probably be prefetched when the process is loaded

can utilize valid/invalid bit in page table to determine if in physical memory

v/1 → page is in physical memory, so access is valid

i/0 → page is in secondary memory, so invalid

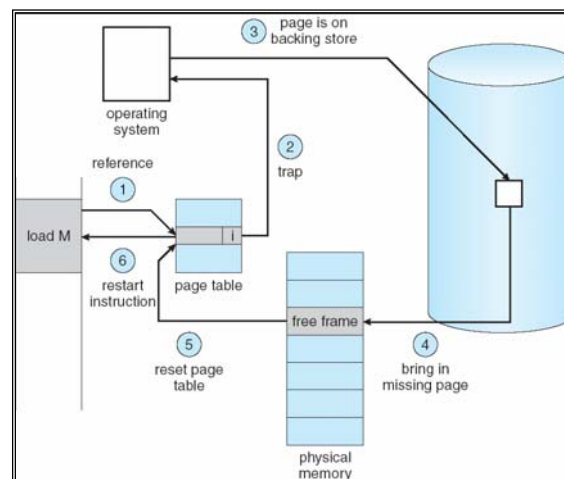


5

## Page fault

if the process references a logical page that is not in physical memory:

1. memory management unit checks page table
2. *page fault exception* is raised due to invalid bit
3. OS must locate logical page in secondary memory
4. schedules a disk operation to swap into memory
5. at disk interrupt, swaps page into a free frame
6. sets valid bit in page table
7. restarts instruction



6

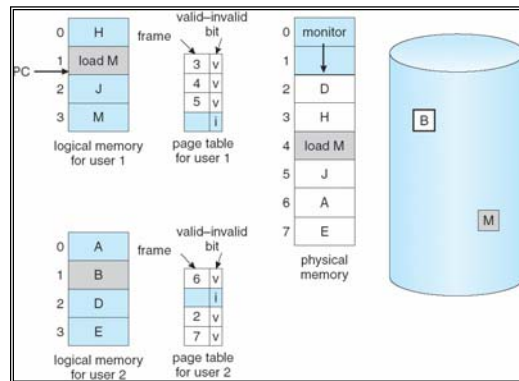
## Page replacement

what if there is no free frame available?

- must select a frame and make it available
- maintain a *free list* of unused frames
- if free list is empty, locate a victim frame
- swap page to secondary memory and change page table entry

can maintain a *dirty bit* with each frame to identify pages that have been modified

- if not modified, then don't need to save page



## Performance of demand paging

can calculate an *effective access time* for a demand paging system

- represents average cost of page access (both in physical memory and page fault)

$$\text{effective access time} = (1 - p) * M + p * F$$

where:  $p$  is the probability of a page fault  
 $M$  is time to access memory  
 $F$  is time to service a page fault

effective access time is directly proportional to page fault rate

- suppose memory access time = 200 ns, page fault service time = 8 ms

$$\begin{aligned} \text{effective access time} &= (1 - p) * M + p * F \\ &= (1 - p) * 200 + p * 8,000,000 \\ &= 200 + 7,999,800 * p \end{aligned}$$

due to high swapping cost, even tiny page fault rate has huge impact

$p = 0.001 \rightarrow$  effective access time = 8.2  $\mu$ s (40 times slower than no faults)

8

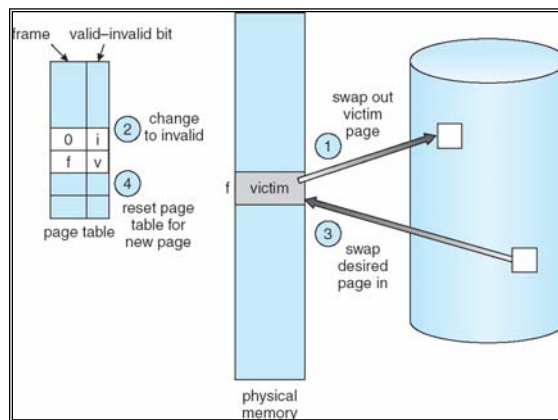
## Page replacement algorithms

page replacement completes separation between logical & physical memory

- large virtual memory can be provided on a smaller physical memory

choices as to which pages are swapped in and out of memory have significant impact on effective access time

we want page replacement algorithms that minimize the page fault rate



9

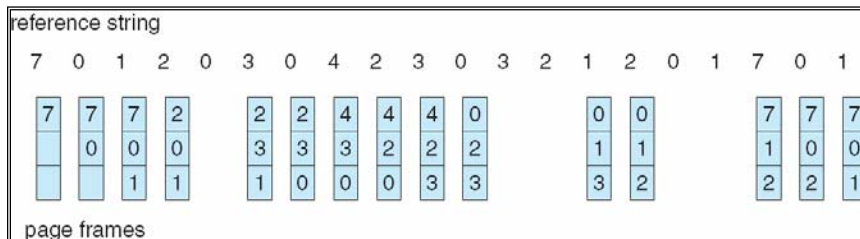
## First-In-First-Out (FIFO) algorithm

select as victim the page that has been in memory the longest

- i.e., treat frame set as a queue, replace pages in FIFO order
- simple, but not very effective in practice  
doesn't favor pages that are used often (e.g., contains frequently used variable)

example: suppose 3 pages can be in memory at a time per process

- process references pages: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1
- requires 15 page faults



10

## Belady's anomaly

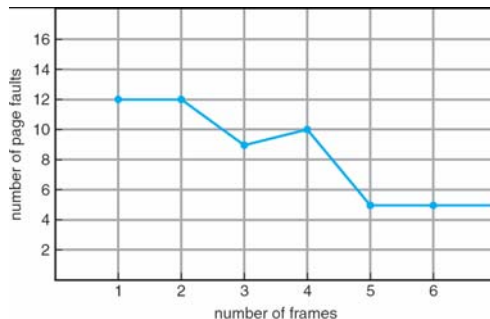
normally, increasing the number of frames allocated to a process will reduce the number of page faults

however, not always the case

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

with this reference string, actually have more page faults with 4 frames than with 3

this rare but highly undesirable situation is known as *Belady's anomaly*



11

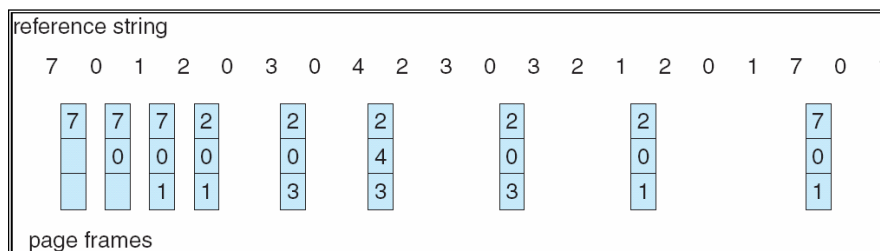
## Optimal algorithm

Belady's anomaly led to the search for an optimal page replacement algorithm

- optimal algorithm will have the lowest page fault rate
- optimal algorithm will never suffer from Belady's anomaly

OPT: select as victim the page that will not be used for the longest time

- same reference string requires only 9 page faults (compared with 15 for FIFO)



12

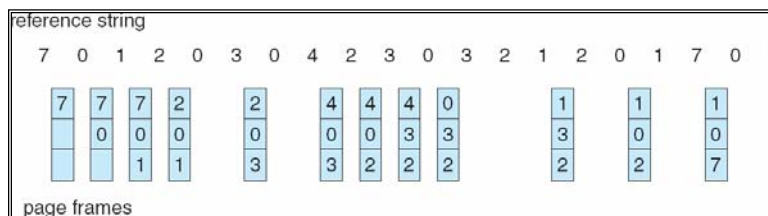
## Least Recently Used (LRU) algorithm

OPT is not practical since it requires future knowledge

- often used for comparative studies, i.e., algorithm X is within Y of optimal

however, we can approximate OPT under certain assumptions

- if we use the recent past as an approximation of the near future, then will select as victim the page that has been used least recently
- LRU works well in practice, does not suffer from Belady's anomaly



13

## OPT & LRU & Belady's anomaly

OPT and LRU are known as *stack algorithms*

- can be proven that:  
 $pages\ in\ memory\ using\ N\ frames \subseteq pages\ in\ memory\ using\ N+1\ frames$
- as a result, cannot suffer from Belady's anomaly

WHY?

what about FIFO?

- recall the reference string that resulted in Belady's anomaly

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

- 9 page faults with 3 frames, 10 page faults with 4 frames

14

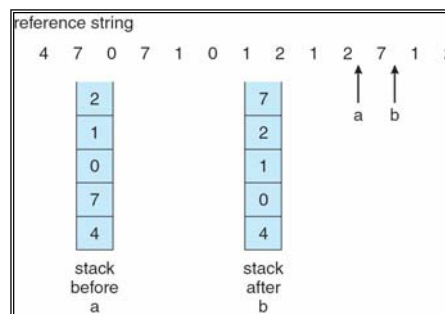
## Implementing LRU

### LRU can be implemented using a clock-based approach

- when referenced, record the time in the page table (takes up a lot of space)
- when selecting victim, sort reference times to locate the earliest (time consuming)

### alternatively, could use a stack-based implementation

- maintain a (pseudo) stack of page numbers, most recent at top
- when a page is referenced, move its number to the top (requires some work)
- when need to select a page, LRU is at bottom of stack



note: implementing LRU requires hardware support

updating clock times or stack must be done for every reference, software interrupts are too slow!

15

## LRU approximation algorithms

### few systems provide sufficient hardware support for full LRU

- instead, limited support can allow approximations

### reference bit (more like RU)

- with each page associate a bit, periodically set = 0
- as pages are referenced, the hardware sets the reference bit
- whenever need to select a frame, select one with reference bit = 0
- could use multiple bits to simulate time segments  
e.g., when referenced, set rightmost bit; left-shift bits periodically

### second chance

- modified FIFO using a reference bit
- if the oldest page has reference bit = 1. then give it a second chance:  
set reference bit 0, leave in memory, replace next page (subject to same rules)
- could even take the dirty bit into account: prefer pages that have not been changed

16



## Paging optimizations

various optimizations can be added to page replacement

- e.g., page buffering  
maintain a list of free frames  
when a page fault occurs, choose a victim page as before
  - first copy new page into one of the free frames
  - then, swap victim page out to disk and add its frame to free listADVANTAGE: can access desired page faster
- e.g., preemptive swapping  
maintain a list of modified pages  
when the paging device is idle, copy a page back to disk & reset its dirty bit  
ADVANTAGE: better utilization of paging device, reduces need for swapping out modified pages

17

## Allocation of frames

how do we allocate frames to multiple processes?

- let processes have all the frames they ask for, until memory runs out
- divide frames equally between processes
- divide frames proportionally based on size of processes
- divide frames proportionally based on priorities

note: each process has a minimum number of frames that it requires

- must be sure to allocate enough for process to execute

e.g., on PDP-11, MOVE instruction can be larger than a word  
thus, instruction might straddle page boundary  
MOVE has two operands, each of which might require 2 pages  
→ may require 6 frames to store the instruction

18

## Local vs. global allocation

when a process asks for a frame, where does the victim come from?

### local replacement

- each process selects from only its own set of allocated frames
- the number of frames allocated to a process is constant
- a process controls its own page fault rate

### global replacement

- process selects a replacement frame from the set of all frames
- one process can take a frame from another
- allows a process to swap out less used (lower priority?) pages, no matter the owner
- commonly used since it improves system throughput

19

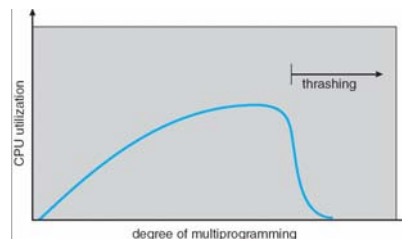
## Thrashing

when a system experiences a high degree of paging activity, we call this *thrashing*

generally, thrashing is caused by processes not having enough pages in memory

- using global replacement, can occur when process steal frames from each other
- but, can even happen using local replacement

- thrashing processes lead to low CPU utilization
- OS (long-term scheduler) thinks it needs to increase degree of multiprogramming
- more processes are added to the system (taking frames from existing processes)
- worse thrashing



20

## Paging & thrashing

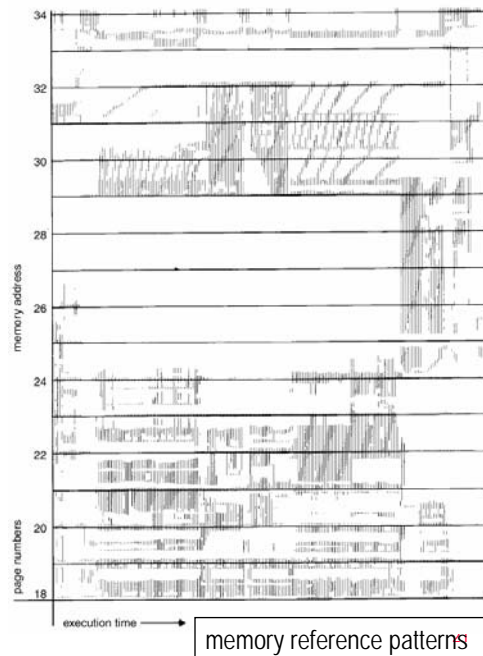
### why does paging work?

#### locality model

- a locality is a set of pages that are actively used together
- program is composed of several different localities, which may overlap
- process migrates from one locality to another.
- if we allocate enough frames to accommodate the locality, faults will only occur when transitioning

### why does thrashing occur?

$\Sigma$  locality sizes > total memory size



## Programmer can influence paging/thrashing

suppose a page can store 1K ints

consider a C/C++/Java program with a 2-D array

```
int data[1024][1024];
```

Program 1:

```
for (j = 0; j < 1024; j++)
  for (i = 0; i < 1024; i++)
    data[i][j] = 0;
```

1,024 x 1,024 = 1,048,576 page faults

Program 2:

```
for (i = 0; i < 1024; i++)
  for (j = 0; j < 1024; j++)
    data[i][j] = 0;
```

1024 page faults

22

## Working-set model

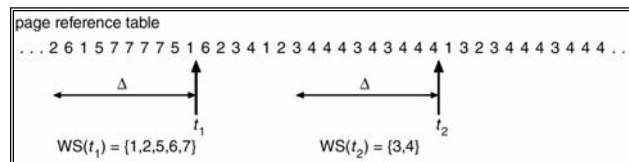
approach to thrashing: first, estimate the frame demand for processes

- at any given time, there is a minimum set of pages required for each process
- can also keep track of the minimum set of pages needed for some number of previous instructions

*working-set window*:  $\Delta$  = a fixed number of page references

*working set*:  $WS(t)$  = set of pages required by process during most recent  $\Delta$

*working set size*:  $WSS_i = |WS(t)|$  for process  $P_i$



the working set model adjusts the level of multiprocessing in the system based on demand during the most recent window :  $D = \sum WSS_i$

- if demand (D) exceeds the number of available frames, then thrashing occurs
- suspend processes until demand lowers, may eventually restart

23

## Implementing the working set

keeping track of the working set is tricky

- can approximate with an interval timer + a reference bit

Example:  $\Delta = 10,000$  (and assume timer interrupt every 1,000 time units)

- maintain a reference bit, 10 memory bits for each page
- whenever a timer interrupts, copy reference bit into next memory bit & reset
- if one of the bits in memory = 1  $\Rightarrow$  page in working set.

why is this not completely accurate?

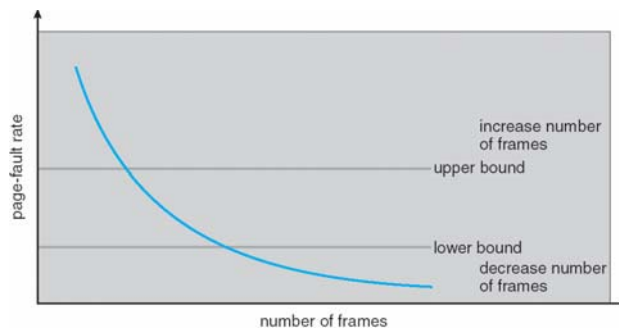
improvements? tradeoffs?

24

## Page-fault frequency scheme

monitoring page fault frequency is a simpler and more direct approach to avoiding thrashing

- if actual rate too low, process loses frame
- if page fault frequency is too high, then processes need more frames  
→ suspend processes to decrease degree of multiprocessing
- if page fault frequency is too low, then processes may have too many frames  
→ restart suspended processes to increase degree of multiprocessing



25

## Other considerations

replacement algorithm and allocation policy are the major decisions required in a paging system, but other considerations apply

- prepaging  
try to predict pages that will be needed and load ahead of time  
e.g., keep track of past working sets for processes, load as a group
- page size  
page size affects the page table size, disk I/O times, etc.  
generally, page size is set by the hardware
- locked pages  
some pages cannot be removed, e.g., I/O buffers, OS kernel  
e.g., don't want to swap out page that is currently having I/O performed
- real-time processing  
virtual memory prevents real-time processing (unless pages can be locked)

26

## Windows XP

uses demand paging with *clustering*

- clustering brings in pages surrounding the faulting page

processes are assigned *working set minimum* and *working set maximum*

- working set minimum is the minimum number of pages the process is guaranteed to have in memory (for most apps, 50...345)
- a process may be assigned as many pages up to its working set maximum
- when the amount of free memory in the system falls below a threshold, *automatic working set trimming* is performed to restore the amount of free memory
- working set trimming removes pages from processes that have pages in excess of their working set minimum

page replacement algorithm varies depending on processor

- single processor Intel CPU: variation of the clock (timestamp) algorithm
- multiprocessor or Alpha CPU: variation of FIFO

27

## Solaris

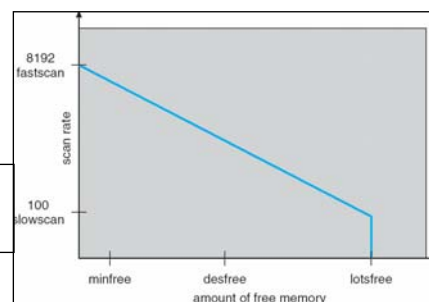
system maintains a list of free pages

- paging occurs when the number of free pages drops below a threshold (e.g., 1/64 size of physical memory)

paging is performed by a *pageout* process

- pageout scans pages using a modified clock algorithm
- pages not referenced since last scan are returned to free list
- the smaller the free list, the more frequent scanning occurs

*lotsfree* – threshold parameter to begin paging  
*desfree* – threshold parameter to increase paging  
*minfree* – threshold parameter to begin swapping



28