LASSONDE SCHOOL OF ENGINEERING | YORK UNIVERSITÉ UNIVERSITY

EECS 3221:
Operating System Fundamentals

Hamzeh Khazaei

Department of Electrical Engineering and Computer Science

Week 10:

Virtual Memory

March 15, 2021

1

Chapter 10: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples

2

Background

- Code 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
- Consider ability to execute partially-loaded program
  1. Program no longer constrained by limits of physical memory
  2. 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
  3. Less I/O needed to load or swap programs into memory -> each user program runs faster

3

Virtual Memory

- **Virtual memory** – 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
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes

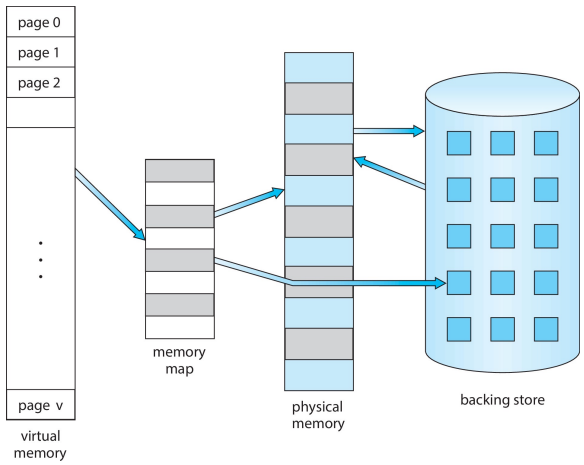
4

Virtual memory (Cont.)

- **Virtual address space** – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

5

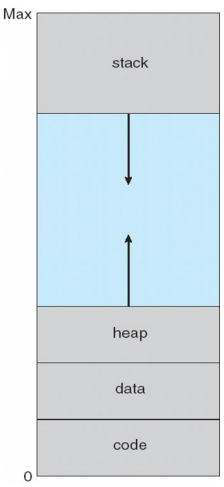
Virtual Memory That is Larger Than Physical Memory



6

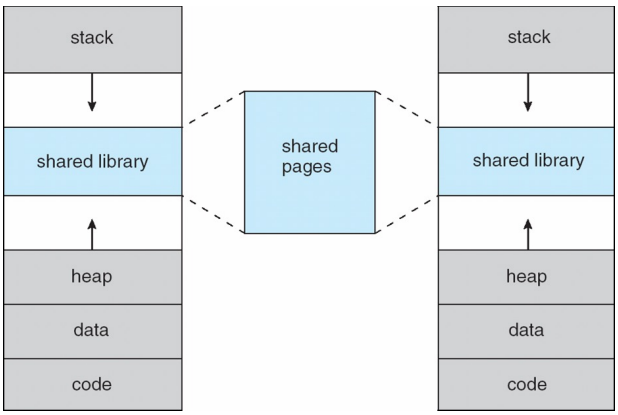
Virtual-address Space

- Usually design logical address space for stack to start at Max logical address and grow “down” while heap grows “up”
  - Maximizes address space use
  - Unused address space between the two is hole
    - No physical memory needed until heap or stack grows to a given new page
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during `fork()`, speeding process creation



7

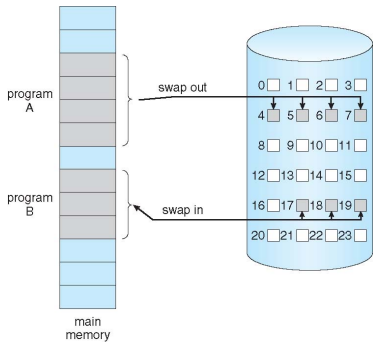
Shared Library Using Virtual Memory



8

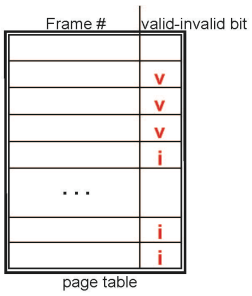
Demand Paging

- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Like paging system with swapping (diagram on right)
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**



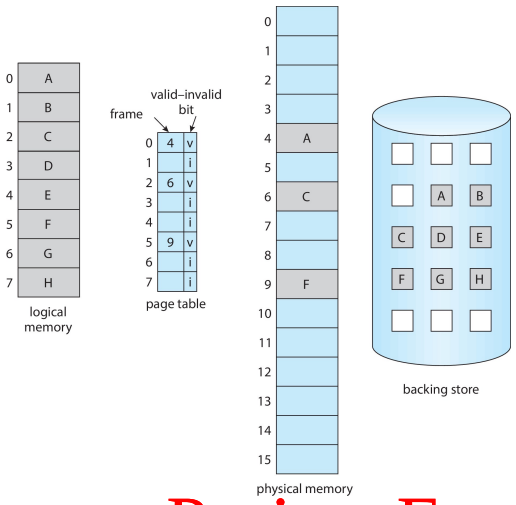
Valid-Invalid Bit

- With each page table entry, a valid–invalid bit is associated (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:



- During MMU address translation, if valid–invalid bit in page table entry is **i** ⇒ **page fault**

Page Table When Some Pages Are Not in Main Memory



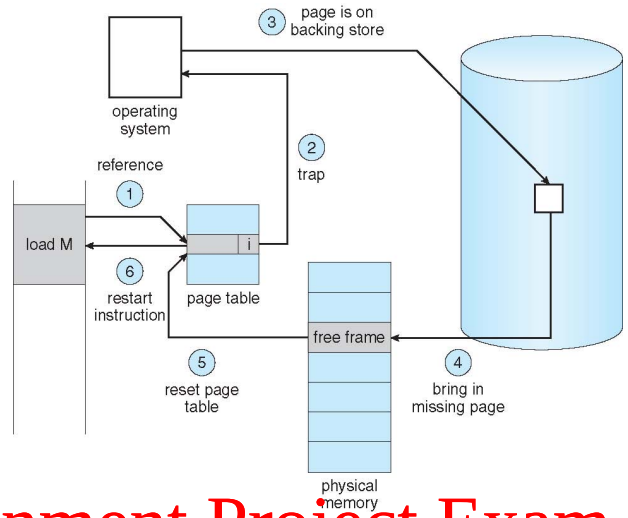
11

Steps in Handling Page Fault

1. If there is a reference to a page, first reference to that page will trap to operating system
   - Page fault
2. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory
3. Find free frame
4. Swap page into frame via scheduled disk operation
5. Reset tables to indicate page now in memory --> set validation bit = **v**
6. Restart the instruction that caused the page fault

12

Steps in Handling a Page Fault (Cont.)



13

Free-Frame List

- When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory.
- Most operating systems maintain a **free-frame list** -- a pool of free frames for satisfying such requests.

- Operating system typically allocate free frames using a technique known as **zero-fill-on-demand** -- the content of the frames zeroed-out before being allocated.
- When a system starts up, all available memory is placed on the free-frame list.

14

Performance of Demand Paging

- Three major activities
  - Service the interrupt – careful coding means just several hundred instructions needed
  - Read the page – lots of time
  - Restart the process – again just a small amount of time
- Page Fault Rate $0 \le p \le 1$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

EAT = (1 – *p*) x memory access + *p* (page fault overhead + swap page out + swap page in)

Demand Paging Example

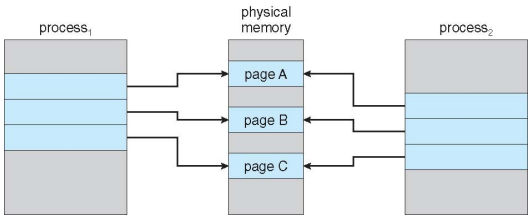
- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 + p (8 milliseconds)
  = ((1 – p) x 200) + (p x 8,000,000)
  = 200 + (p x 7,999,800)
- If one access out of 1,000 causes a page fault (p=0.001), then
  EAT = 8.2 microseconds.
  This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 x p
    20 > 7,999,800 x p
  - p < 0.0000025
  - p < one page fault in every 400,000 memory accesses

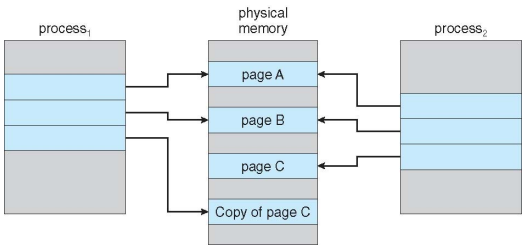
Copy-on-Write

- **Copy-on-Write** (COW) 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
- COW allows more efficient process creation as only modified pages are copied
- `vfork()` variation on `fork()` system call has parent suspend and child using copy-on-write address space of parent
  - Designed to have child call `exec()`
  - Very efficient

17

Before Process 1 Modifies Page C





18

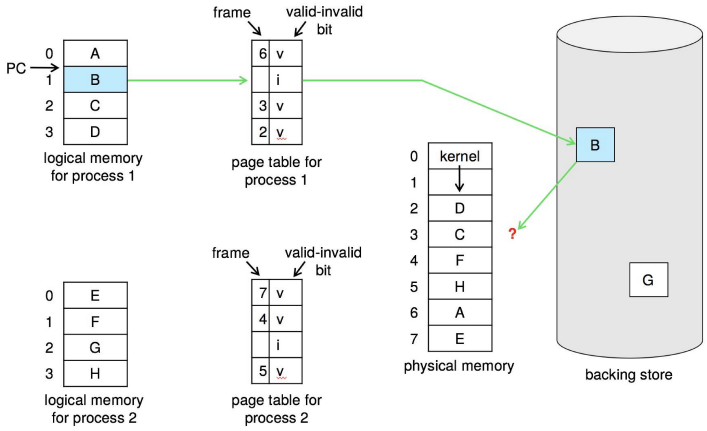
What Happens if There is no Free Frame?

- Used up by process pages
- Also, in demand from the kernel, I/O buffers, etc.
- Algorithm – terminate? swap out? replace the page?
  - Page replacement – find some page in memory, but not really in use, page it out
- Performance – want an algorithm which will result in minimum number of page faults
- How much to allocate to each?
- Same page may be brought into memory several times

Page Replacement

- Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement
- Use **modify** (**dirty**) **bit** to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

Need For Page Replacement



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

Note now potentially 2-page transfers for page fault – increasing EAT

Page Replacement

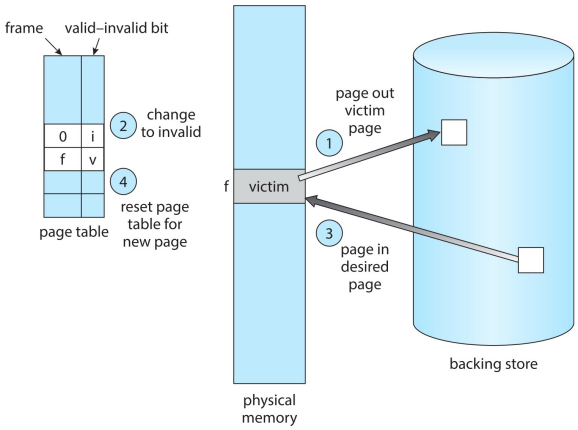


Figure 10.10 Page replacement

23

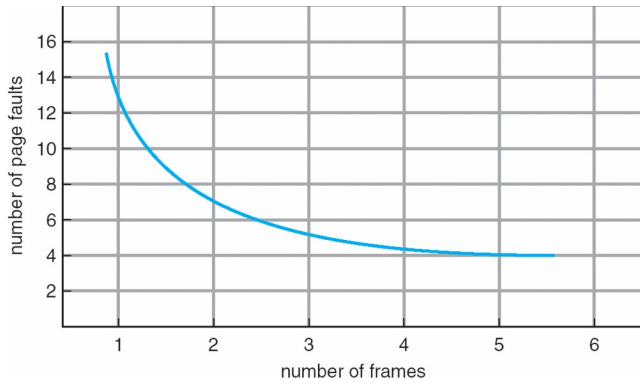
Page and Frame Replacement Algorithms

- **Frame-allocation algorithm** determines
  - How many frames to give each process
- **Page-replacement algorithm**
  - Which pages to replace
  - Want lowest page-fault rate on both first access and re-access
- 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
  - Repeated access to the same page does not cause a page fault
  - Results depend on number of frames available
- In all our examples, the **reference string** of referenced page numbers is

  7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

24

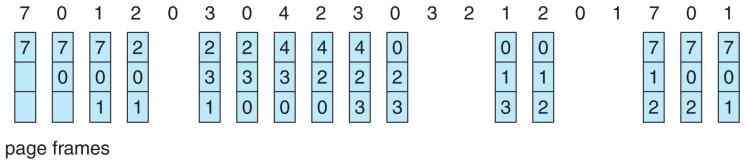
Graph of Page Faults Versus The Number of Frames (in general)



25

First-In-First-Out (FIFO) Algorithm

- 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)

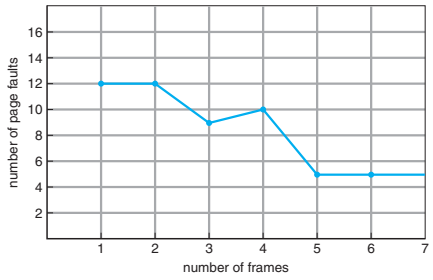


FIFO: 15 page-faults

26

FIFO Illustrating Belady's Anomaly

- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
  - Adding more frames can cause more page faults!
    - **Belady's Anomaly**

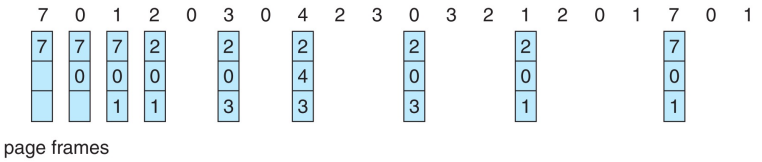


- How to track ages of pages?
  - Just use a FIFO queue

27

Optimal Algorithm

- Replace page that will not be used for longest period of time
  - 9 page-faults is optimal for the example
- How do you know this?
  - Can't read the future
- Used for measuring how well your algorithm performs

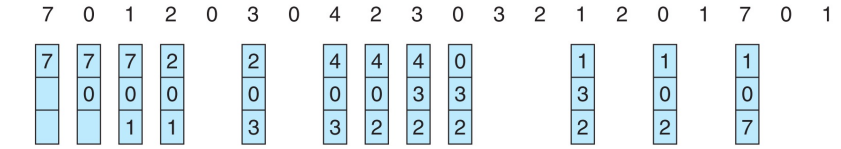


- 9 faults – absolute best.

28

Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page



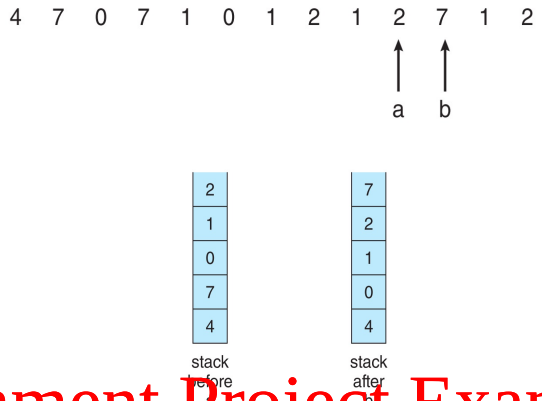
- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?

LRU Algorithm (Cont.)

- 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 and
    - A write to memory for each memory access
- 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
  - But each update more expensive
  - The tail pointer points to the bottom of the stack, which is the LRU page
  - No search for replacement

Use Of A Stack to Record Most Recent Page References

*LRU (like OPT) is a case of **stack algorithms** that doesn't't have Belady's Anomaly.*



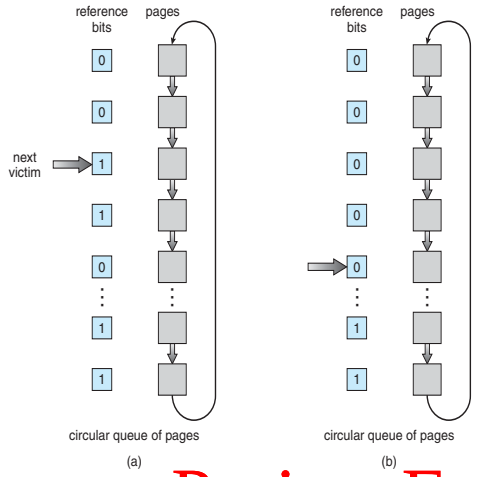
31

LRU Approximation Algorithms

- LRU needs special hardware and still slow
- **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, however
- **Second-chance algorithm**
  - Generally, FIFO, plus hardware-provided reference bit
  - **Clock** replacement
  - If page to be replaced has
    - Reference bit = 0 -> replace it
    - reference bit = 1 then:
      - set reference bit 0, leave page in memory
      - replace next page, subject to same rules

32

Second-Chance (clock) Page-Replacement Algorithm



33

Enhanced Second-Chance Algorithm

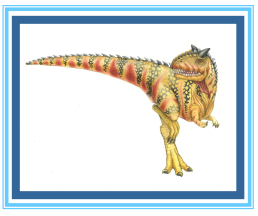
- Improve algorithm by using two bits: reference bit and modify bit (if available) in concert
- Take ordered pair (reference, modify):
  - (0, 0) neither recently used not modified – best page to replace
  - (0, 1) not recently used but modified – not quite as good, must write out before replacement
  - (1, 0) recently used but clean – probably will be used again soon
  - (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

34

Counting Algorithms

- Keep a counter of the number of references that have been made to each page
- **Least 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
- Not common – potential problems?
- One solution is to shift the counts right by 1 bit at regular intervals, forming an exponentially decaying average usage count.

Second Module



Add-on procedures – best practices

- Other procedures are often used in addition to a specific page-replacement algorithm:
  - Keep a pool of free frames, always
    - Then frame available when needed, not to be found at fault time
    - Read page into free frame and select victim to evict and add to free pool
    - When convenient, evict victim
  - Possibly, keep list of modified pages
    - When backing store idle, write pages there and set to non-dirty
  - Possibly, keep free frame contents intact and note what is in them
    - If referenced again before reused, no need to load contents again from disk
    - Generally useful to reduce penalty if wrong victim frame selected

37

Applications and Page Replacement

- All these algorithms have OS guessing about future page access
- Some applications have better knowledge – i.e., databases
- Memory intensive applications can cause double buffering
  - OS keeps copy of page in memory as I/O buffer
  - Application keeps page in memory for its own work
- Operating system can give direct access to the disk, getting out of the way of the applications
  - **Raw disk** mode
  - Bypasses buffering, locking, etc.

38

Allocation of Frames

- Each process needs ***minimum*** number of frames
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*
- ***Maximum*** of course is total frames in the system
- Two major allocation schemes
  - fixed allocation
  - priority allocation
- Many variations

Fixed Allocation

- Equal allocation – For example, if there are 93 frames (after allocating frames for the OS) and 5 processes, give each process 18 frames
  - Keep 3 frames as free frame buffer pool
- Proportional allocation – Allocate according to the size of process
  - Dynamic as degree of multiprogramming, process sizes change

– s_i = size of process p_i

– $S = \sum s_i$

– m = total number of frames

– a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$m = 64$

$s_1 = 10$

$s_2 = 127$

$a_1 = \frac{10}{137} \times 62 \approx 4$

$a_2 = \frac{127}{137} \times 62 \approx 57$

Global vs. Local Allocation

- Now that we know allocation, let's get back to page replacement:
  - **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
    - But then process execution time can vary greatly
    - But greater throughput so **more common**
  - **Local replacement** – each process selects from only its own set of allocated frames
    - More consistent per-process performance
    - But possibly underutilized memory

41

Priority Allocation

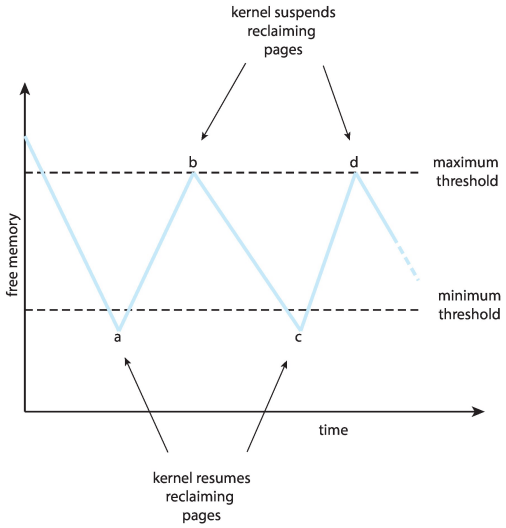
- Use a proportional allocation scheme but using priorities rather than size
- If process P_i generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority number

42

Reclaiming Pages

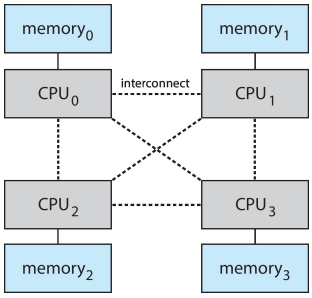
- A strategy to implement global page-replacement policy
- All memory requests are satisfied from the free-frame list
  - The free-frame list will not be lower/upper than a threshold
- Page replacement is triggered when the list falls below a certain threshold.
- This strategy attempts to ensure there is always enough free memory to satisfy new requests.

Reclaiming Pages Example (reaper procedure)



Non-Uniform Memory Access

- So far, all memory accessed equally
- Many systems are **NUMA** – speed of access to memory varies
  - Consider system boards containing CPUs and memory, interconnected over a system bus
- NUMA multiprocessing architecture



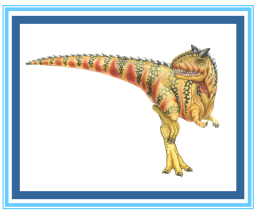
Non-Uniform Memory Access (Cont.)

- Optimal performance comes from allocating memory "close to" the CPU on which the thread is scheduled
  - And modifying the scheduler to schedule the thread on the same system board when possible
  - Solved by Solaris by creating **lgroups**
    - Structure to track CPU / Memory low latency groups
    - Used my schedule and pager
    - When possible, schedule all threads of a process and allocate all memory for that process within the lgroup

Thrashing

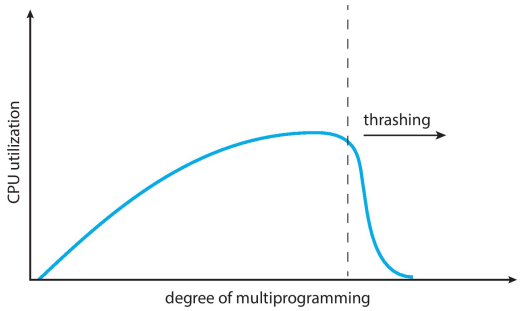
- A process is thrashing if it is spending more time paging than executing.
- If a process does not have "enough" pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process added to the system – the problem might get exacerbated.

Third Module



Thrashing (Cont.)

- **Thrashing.** A process is busy swapping pages in and out



49

Demand Paging and Thrashing

- Why does demand paging work?

  Locality model

  - Process migrates from one locality to another
  - Localities may overlap

- Why does thrashing occur?

  Σ size of locality > total memory size

- We can limit the effect by using local or priority page replacement

50

Locality In A Memory-Reference Pattern

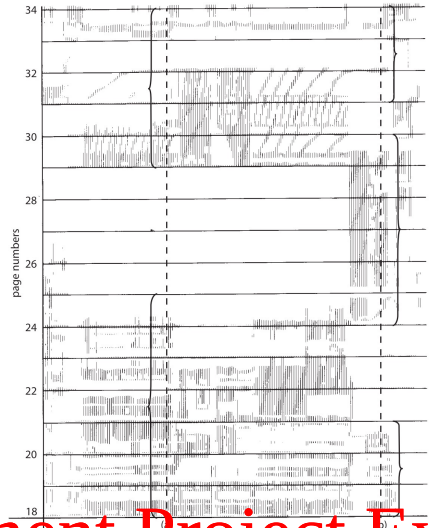


Figure 10.21 Locality in a memory-reference pattern.

51

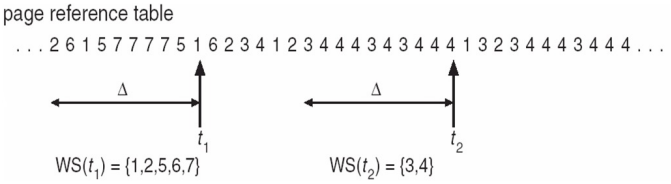
Working-Set Model

- The working-set model is based on the assumption of locality.
- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instructions
- WSS_i (working set of Process P_i) = total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \Sigma\ WSS_i \equiv$ total demand frames
  - Approximation of locality

52

Working-Set Model (Cont.)

- if $D > m \Rightarrow$ Thrashing
- Policy if $D >$ m, then suspend or swap out one of the processes
- Following picture, delta = 10



53

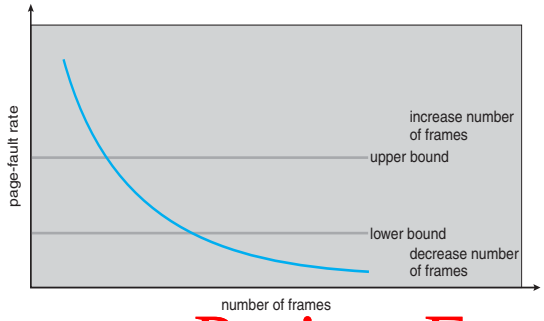
Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: Δ = 10,000
  - Timer interrupts after every 5000 time-units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time-units
  - Any problems?

54

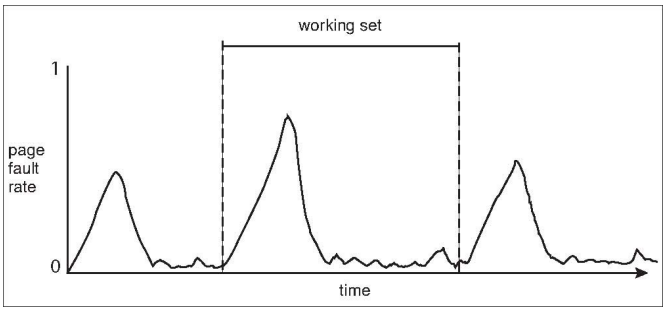
Page-Fault Frequency

- More direct approach than WSS
- Establish "acceptable" page-fault frequency (PFF) rate and use **local replacement policy**
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



Working Sets and Page Fault Rates

- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- Peaks and valleys over time



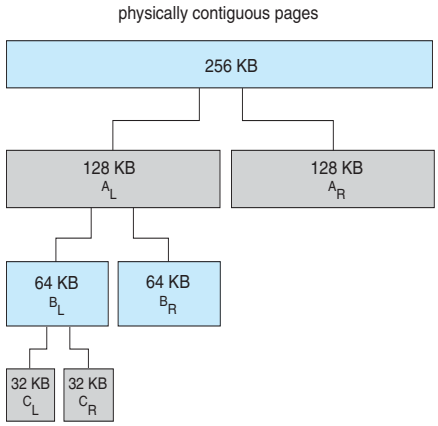
Allocating Kernel Memory

- Treated differently from user memory
  - Allocating and deallocating memory frequently
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - Some kernel memory needs to be contiguous
    - i.e., for device I/O
- Now, we examine two strategies for managing free memory that is assigned to kernel processes:
  - The “buddy system” and
  - The “slab allocation”

Buddy System

- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using **power-of-2 allocator**
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
    - Continue until appropriate sized-chunk become available
- For example, assume 256KB chunk available, kernel requests 21KB
  - Split into A_L and A_R of 128KB each
    - One further divided into B_L and B_R of 64KB
      - One further into C_L and C_R of 32KB each – one used to satisfy request
- Advantage – quickly **coalesce** unused chunks into larger chunk
- Disadvantage – ?
  - internal fragmentation
    - 50% wastage of memory

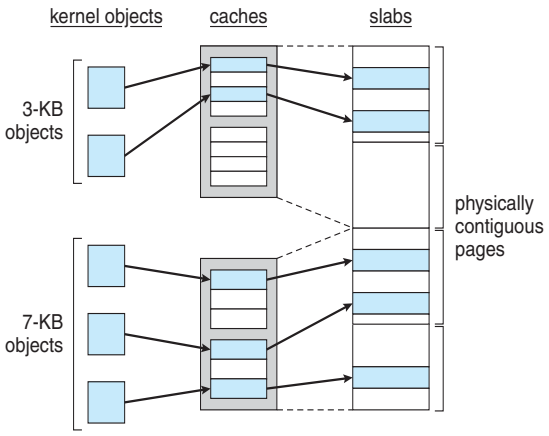
Buddy System Allocator



Slab Allocator

- Alternate strategy
- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure
  - Each cache filled with **objects** – instantiations of the data structure
- When cache created, filled with objects marked as **`free`**
- When structures stored, objects marked as **`used`**
- If slab is full of used objects, next object allocated from empty slab
  - If no empty slabs, new slab allocated
- Benefits include no fragmentation, fast memory request satisfaction

Slab Allocation



61

Slab Allocator in Linux

- For example, process descriptor is of type `struct task_struct`
- Approximately 1.7KB of memory
- New task -> allocate new struct from cache
  - Will use existing free `struct task_struct`
- Slab can be in three possible states
  1. Full – all used
  2. Empty – all free
  3. Partial – mix of free and used
- Upon request, slab allocator
  1. Uses free struct in partial slab
  2. If none, takes one from empty slab
  3. If no empty slab, create new empty

62

Slab Allocator in Linux (Cont.)

- Slab started in Solaris, now wide-spread for both kernel mode and user memory in various OSes
- Linux 2.2 had SLAB, now has both SLOB and SLUB allocators
  - SLOB for systems with limited memory, like embedded systems
    - Simple List of Blocks – maintains 3 list objects for small (256 byte), medium (less than 1024 byte), large objects
    - Memory requests are allocated from an object on the appropriate list using a first-fit policy.
  - SLUB, beginning at version 2.6.24, is performance-optimized SLAB
    - removes per-CPU queues (memory saving in multi-cpu systems)
    - metadata stored in page structure

63

Other Considerations

- Prepaging
- Page size
- TLB reach
- Program structure
- I/O interlock and page locking

64

Prepaging

- To reduce the large number of page faults that occurs at process startup
- Pre-page all or some of the pages a process will need, before they are referenced
- But if pre-paged pages are unused, I/O and memory was wasted
- Assume ***s*** pages are pre-paged and ***a%*** of the pages is used
  - Is cost of ***s * a*** save pages faults > or < than the cost of pre-paging ***s * (100 - a)*** unnecessary pages?
  - ***a*** near zero ⇒ pre-paging loses

Page Size

- Sometimes OS designers have a choice
  - Especially if running on custom-built CPU
- Page size selection must take into consideration:
  - Fragmentation
  - Page table size
  - Resolution
  - I/O overhead
  - Number of page faults
  - Locality
  - TLB size and effectiveness
- Always power of 2, usually in the range 2^{12} (4,096 bytes) to 2^{22} (4,194,304 bytes)
- On average, growing over time

TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
- TLB Reach = (TLB Size) X (Page Size)

- Ideally, the working set of each process is stored in the TLB
  - Otherwise, there is a high degree of TLB miss

- Increase the Page Size
  - This may lead to an increase in fragmentation as not all applications require a large page size

- Provide Multiple Page Sizes
  - This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation

67

Program Structure

- Program structure
  - `int[128,128] data;`
  - Each row is stored in one page of size 2^{12}
  - Program 1

```
for (j = 0; j <128; j++)
    for (i = 0; i < 128; i++)
        data[i,j] = 0;
```

128 x 128 = 16,384 page faults

  - Program 2

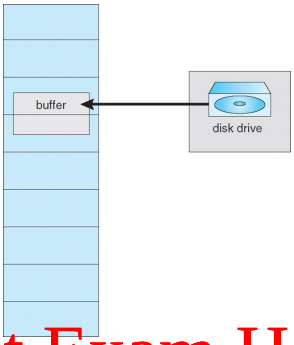
```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i,j] = 0;
```

128 page faults

68

I/O interlock

- **I/O Interlock** – Pages must sometimes be locked into memory
- Consider I/O - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm
- **Pinning** of pages to lock into memory



69

Operating System Examples

- Linux
- Solaris
- Windows

70

Windows

- 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
- A process may be assigned as many pages as possible 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

Performance of Demand Paging

- Stages in Demand Paging (worse case)

1. Trap to the operating system
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
5. Issue a read from the disk to a free frame:
   1. Wait in a queue for this device until the read request is serviced
   2. Wait for the device seek and/or latency time
   3. Begin the transfer of the page to a free frame
6. While waiting, allocate the CPU to some other user/program
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user/program
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 (process now in ready queue)
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

End of Chapter 10.