Virtual Memory Management

Memory Management

- Previous solutions
  - Need to know the memory requirement for the process
    - Fixed partitioning, dynamic partitioning, simple paging
    - Not easy to do so
  - Need to allocate a contiguous memory space for a process
    - Fixed partitioning, dynamic partitioning
    - So that addressing on the fly can be done efficiently
    - But has fragmentation problem
      - Even there is sufficient memory, allocation may not be possible
  - ⇒ Paging is better
    - Simple paging requires a process to know #pages it uses in advance ⇒ But not difficult to allocate new pages when they are needed
    - How about needing more pages the memory can offer?
    - How about page table overhead

Virtual Memory

- What is the size limit of a process
  - A process can be large as it can address
    - Can be much larger than the physical memory size
  - The limit is set by the address space
    - 32 bits address → 4G address space
      - Even if it is big enough for each process, it is too small to address modern physical memory (beyond 4GB)
    - 64 bit address space → 16 exa bytes
  - Not all parts of the address space of a process have to be in memory at the same time
    - For regular programs ⇒ A lot of the address space is empty
    - For huge programs ⇒ Only those parts that are needed during current and near future execution need to be in memory
  - The "memory" for each process is "virtual"

Virtual Memory and Demand Paging

- Virtual Memory
  - Virtual addresses are used for memory accesses
    - Virtual addresses are mapped to physical addresses on the fly
    - The same in all memory management schemes
  - The size of each process is as large as the address space
    - Only true in virtual memory paradigm
- Demand paging
  - Address space is divided into pages
  - A page is brought into physical memory only when it is needed ⇒ This is called demand paging
  - But where are the remaining pages? ⇒ Disk of course

Virtual Memory and Demand Paging

- Demand paging
  - Swap space
    - All pages (that contain something) are stored on the disk
    - In the swap space allocated specifically for this purpose
      - Generally OS allow admin to set up the swap space size
  - If a needed page is not in memory
    - A **page fault** occurs
    - OS brings the page from swap space into memory
    - …
- ⇒ Memory hierarchy for memory content
    - At least need disk to realize virtual memory, demand paging
    - Cache for improving the access performance
      - For page table as well as content

Memory Hierarchy

- Register
- Cache
- Memory
- Disk (swap space, not files)
  - Many new research consider flash memory as another layer

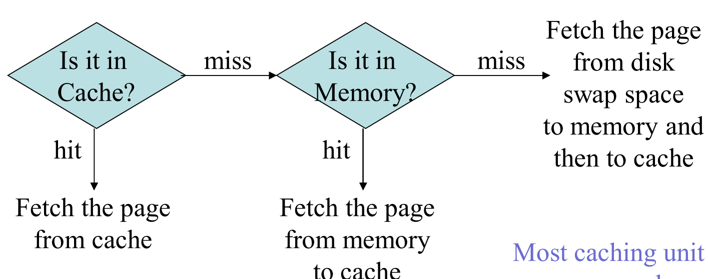
- Down the hierarchy
  - Decreasing cost
  - Increasing capacity and access time

Performance: Memory Hierarchy

- Memory hierarchy characteristics
  - Register: ~ 1 ns or less
  - Cache access time: ~ 1 ns to 10 ns
    - (for i7) L1 cache: ~4 cycles, L2: ~12 cycles, L3: ~40 cycles
  - Memory access time: ~ 50 ns
    - (overall time, not just the memory CAS latency, which is ~10ns)
  - Disk access time: ~ 0.1 ms
  - Cache hit rate
    - Probability that a word is in cache
  - Memory hit rate
    - Probability that a word is not in cache but in memory

Performance: Memory Hierarchy

- Memory access
  - Consider cache, memory, disk
  - Frequently used pages are stored in the cache
  - For a memory access



Most caching unit is not a page, much smaller

Memory Hierarchy (3 Levels)

- Average memory access latency

  $0.95 * 10$ ns +

  $[(1 – 0.95) * 0.999] * (100+10)$ ns +

  $[1 – (0.95 + (1 – 0.95) * 0.999)] * (100000+100+10)$ ns

  $= (9.5 + \sim 5.5 + \sim 5)$ ns $\quad = 1 * 10\text{ns} + .05 * 100\text{ns} + .00005 * 100000\text{ns}$

  $= 20$ ns

Cache	Memory	Disk
Access time = 10 ns	Access time = 100 ns	Access time = 100μs
Hit rate = 95%	Hit rate = 99.9%	

Virtual Memory Issues

- Allocation
  - Different issue in paging schemes, not about where, but …
  - How many pages should be loaded for each process
- Addressing
  - How to map a logical address to a physical address?
  - How to know which frame each page is mapped to?
  - Page table
    - Need a page table to keep track of the pages of a process
      - In cache, or memory, or disk
    - How to store the page table?
    - How to efficiently search the page table?

Virtual Memory Issues

- Replacement
  - A page that has been brought into memory may reside there till it is replaced
  - When a new page is to be brought into memory and there is no free frames
    - Need to replace an existing page
    - How to choose the process and the page to replace?
- Working set management
  - Working set: the set of pages required during execution
  - What is the best working set size?

Virtual Memory Issues

- Protection
  - Protect the address space of one process from being accessed by another process
- Performance metrics
  - No longer consider fragmentation
  - Time for each memory access ≈ Time for page table lookup
  - Number of page faults in a time unit

Addressing: Page Table

- Where to put the page table
  - In main memory, at least
    - Cache is too small, PT on disk will yield ridiculous performance
  - How many pages each process has?
    - 32-bit system: each process has 4GB/8KB = 0.5M pages
    - 64-bit system?
  - ⇒ Page table only stores the pages that are not empty
    - What should the data structure for the page table be so that search can be done efficiently? ⇒ Need a good design!!!
      - Array with binary search? Hash table with linear probing?
      - Potentially more than 1 extra references to M for each M access
    - No matter how good the data structure is, it will be at least one extra memory reference
  - ⇒ Memory hierarchy for page table

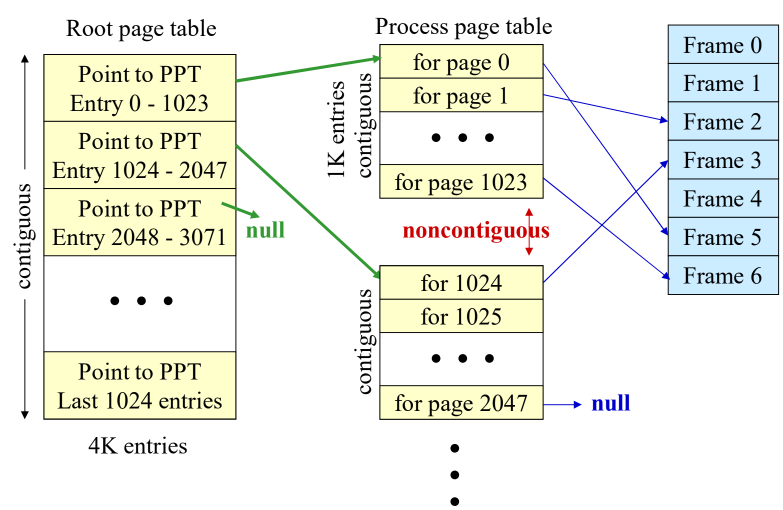
Page Table for Virtual Memory

- Page table designs
  - Process page table
    - Full indexing for each process
    - Can map to memory and disk
  - Inverted Page Table (IPT)
    - Maps memory frames to pages in processes
    - Only address pages in memory
  - Translation lookaside buffer (TLB)
    - Only address pages in memory

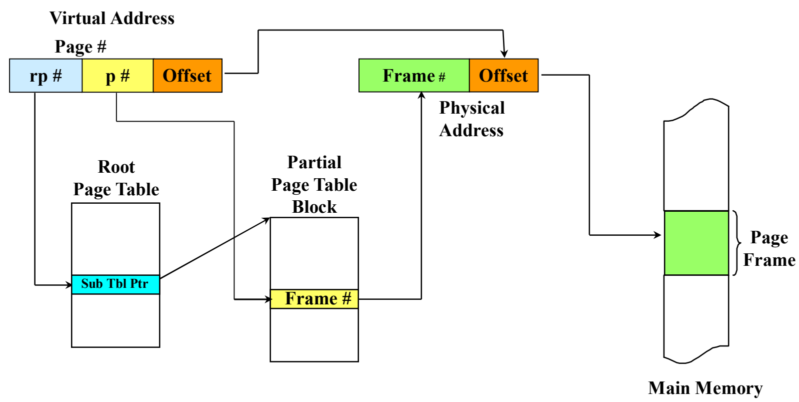
Two-Level Process Page Table

- How to search process page table efficiently
  - Contiguous page table
    - 4GB virtual address space (32 bits), assume 4KB page size
    - Page table has 1M entries for **each process**
  - Most of the entries in the table are empty
  - Non-contiguous
    - How to search? Efficiency will be a problem
  - Two level page table
    - Use another table to index the page table (**root page table**)
    - The low level ones are the **process page tables**
      - E.g., 1K entries in the RPT
      - Each entry in RPT points to the nonempty PPT block

Two-Level Page Table (for one process)

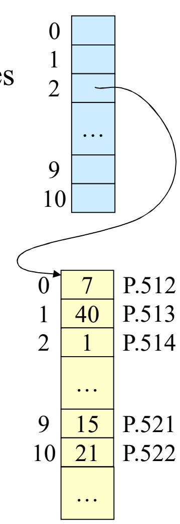


Two-Level Page Table (for one process)



Addressing in Two-Level PT

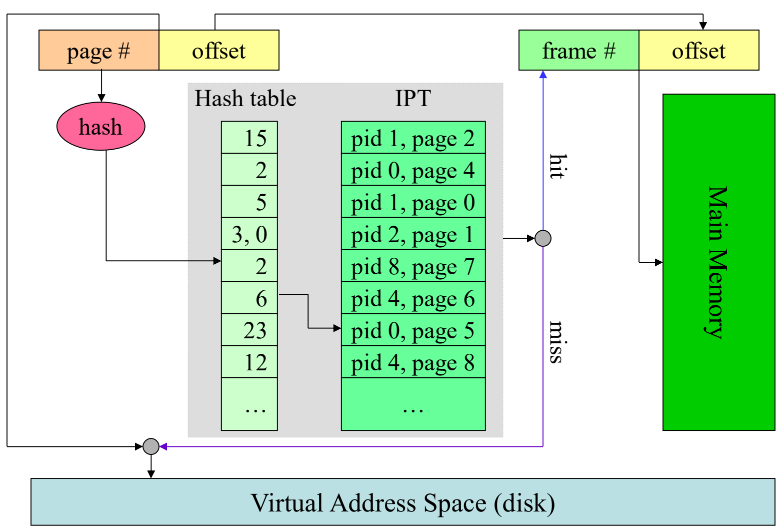
- Addressing
  - 4GB address space, 1KB per page ⇒ 4M pages
  - Root page table has 16K entries
    - Use 14 bits for addressing the root page table
  - Each partial page table block is 256 entries
    - Use 8 bits for addressing the page table block
  - 00000000000010000010100000101011
    - Root table page number = 10 = 2
    - Partial Page block index = 1010 = 10
    - Page offset = 101011
  - 00000000000000000101010000101011
    - Frame number = 21 = 10101



Inverted Page Table (IPT)

- Inverted: Map frame # to processe ID/page #
  - # IPT entries = # frames in physical memory
  - Indexed by physical page frames
  - Useful when a frame is modified
    - Easy to locate the corresponding process ID/page #
- Find page # from frame # can be very expensive
  - Use hash table to map page # to frame #
  - Require at least two extra memory accesses
    - One for hash table, one for IPT accesses
  - Collision may occur in the hash table
    - Need an additional field for chaining

Addressing in IPT



Addressing in IPT

- Addressing
  - Given (process#, page#) tuple
  - Locate the corresponding physical frame
- hash function h = process# $\otimes$ page#
- Example 1: Find frame number for (0,5)
  - $h(0, 5) = 0000 \otimes 0101 = 0101 = 5$
  - hashtable[5] = 6
  - IPT[6] = (0,5) $\Rightarrow$ Got the match
  - (process 0, page 5) is in memory frame 6

Addressing in IPT

- Example 2: Find frame number for (1, 2)
  - $h(1, 2) = 0001 \otimes 0010 = 0011 = 3$
  - hashtable[3] = 3, 0
  - IPT[3] = (2,1) $\Rightarrow$ does not match
  - IPT[0] = (1,2) $\Rightarrow$ Got the match
  - (process 1, page 2) is in memory frame 0

Addressing in IPT

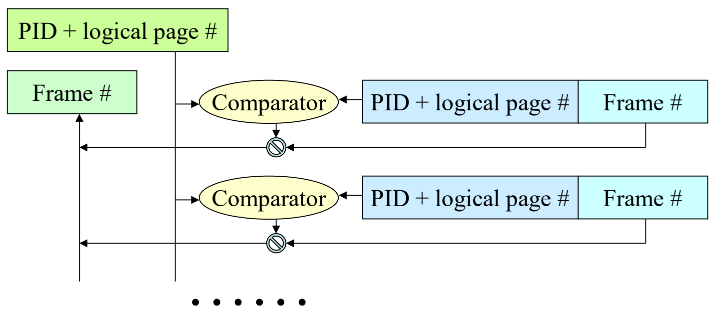
- Example 3: Find frame number for (8, 9)
  - $h(8, 9) = 1000 \otimes 1001 = 0001 = 1$
  - hashtable[1] = 2
  - IPT[2] = (1,0) $\Rightarrow$ does not match
  - (process 8, page 9) is not in memory
  - Issue a page fault

Translation Lookaside Buffer

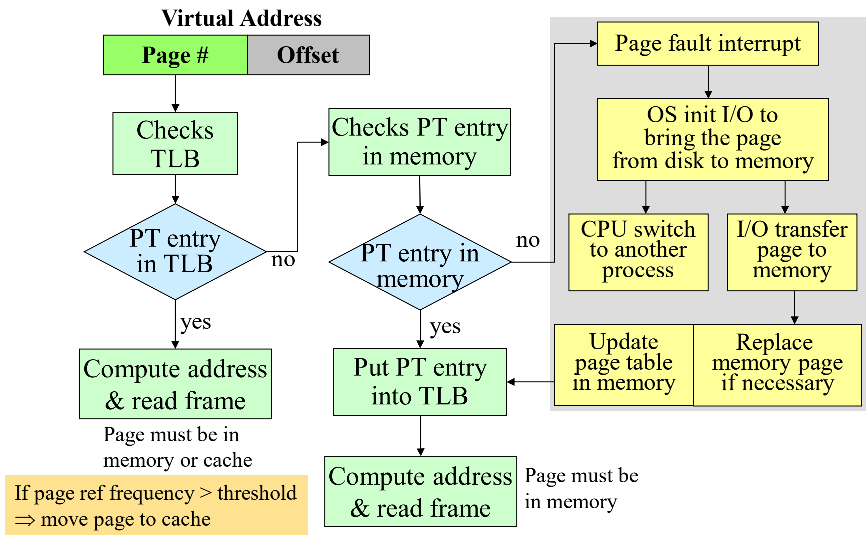
- For every memory access
    - Need to search the page table in memory and map the logical page number to physical frame number
  - $\Rightarrow$ Too expensive
  - Could cache the page table
    - Still relatively expensive
    - How to index the selected pages in the cache (non-contiguous $\Rightarrow$ cannot be indexed $\Rightarrow$ need to search sequentially)
  - Solution
    - Store the page-to-frame mappings in a fast cache
    - Use associative memory to enable single cycle parallel search
    - This is called the translation lookaside buffer (TLB)

Translation Lookaside Buffer

- TLB
  - Use process ID and logical page number as the key
  - Search for the entry in parallel to get the frame number
  - Latency ~ 5-10 ns (not really register speed, ~ cache speed)
  - The page currently referenced is definitely in TLB, for locality



Addressing -- Obtain Frame Number



Working Set Management

- Working set (resident set)
  - The set of pages that a process needs within a time interval
- Working set size for each process
  - Depend on the time interval considered
  - If it is small
    - Higher level of multiprogramming
    - Better for time-sharing systems
    - May result in **thrashing**
      - Page faults occur every few instructions
  - If it is too big
    - May have lower level of concurrency

Working Set Management

int A[128,128], B[128,128];
int i, j, x, y;
for (i=0; i < 128; i++)
  for (j=0; j < 128; j++)
   { $B[i,j] = A[i,j] + x + y$;
    x = (x * i) % 128; y = (y * j) % 128; }

- Memory
  - Page size = 1K bytes; size(int) = 1 word = 4 bytes
  - Program and x, y, i, and j are stored in one page
  - A[0,0]-A[0,127], A[1,0]-A[1,127] in a page
    B[0,0]-B[0,127], B[1,0]-B[1,127] in a page

Working Set Management

- When to load the working set
  - Load on demand
    - A page will be loaded when needed
    - Suitable for new jobs
    - May take some initialization time for a process to get all its needed pages
  - After being swapped out and swapped in
    - Load back pages on demand will incur a big overhead
    - Preload the working set before process starts to run
- Variable working-set size
  - More page faults $\Rightarrow$ Larger working set size
  - Pages that are not used for a time period $\Rightarrow$ Remove the page and reduce the working set size

Page Replacement Policies

- Principle of locality
  - A program that references a location *l* at some point of time is likely to reference the same location *l* and locations in the immediate vicinity of *l* in the near future
    - 90% of the execution time is spent on loops
  - Most of the time, a program is executed sequentially
  - A program tends to favor a subset of its pages during a time interval

Page Replacement Policies

- Replacement policies
  - When a working set of a process is full and a new page is needed $\Rightarrow$ replace an existing page
  - Which page to replace
- Global versus process based
  - Replacement for each process
    - Each process has a working set size
  - Global
    - There is no need to consider working set size for each process
    - May replace any page in the memory

Page Replacement Policies

- Replacement policies
  - When a working set of a process is full and a new page is needed $\Rightarrow$ replace an existing page
  - Which page to replace
- Policies
  - First-in-first-out policy (FIFO)
  - Least recently used policy (LRU)
  - Clock policy
  - Modified clock policy
  - Aging policy

Page Replacement Policies

- **First-in-first-out policy (FIFO)**
  - Simple (use a pointer points to the oldest page)
  - The oldest page may be used more recently
- **Least recently used policy (LRU)**
  - Replace the least recently used page
  - Comply with the principle of locality
  - High implementation overhead
    - Hard to keep track of all references
    - Use a stack for each process

Clock Policy

- Pointer *mrp*
  - points to the most recently **loaded** page
- Flag *used*
  - indicates whether a page has been used recently
- Page replacement
  - Scan and find the **first page with *used* = 0**
    - *mrp* points to the most recently loaded page
    - Scan from the page that is after where *mrp* is pointing to
  - During scanning, **set *used* → 0** if it was 1
  - If no page with *used* = 0 in the first round → will definitely get one during second round

Modified Clock Policy

- Same as before: *mrp* and *used*
- Flag *dirty*
  - Indicate whether a page has been modified
  - *dirty*=1 → the page have to be written out before clear the flag to 0
- Principle
  - Same as clock
  - First try to find a *used* = 0 page
  - Among them, *dirty* = 0 should be considered first

Modified Clock Policy

- Page replacement
  - Scan from *mrp*
  - First scan
    - Find the **first page with *used* = 0 ∧ *dirty* = 0**
    - No change to the *used* and *dirty* bits
  - Second scan, if no suitable pages found
    - Find the **first page with *used* = 0 ∧ *dirty* = 1**
    - During scan, **set *used* → 0** if it was 1
    - Never change dirty bit
  - Third scan, if still no suitable pages found
    - Find the **first page with *used* = 0 ∧ *dirty* = 0**
  - Fourth scan, if still no suitable pages found
    - Find the **first page with *used* = 0 ∧ *dirty* = 1**

Will definitely find one now

Example for Page Replacement Policies

- Page use pattern: 0 1 5 0 1 4 0 1 2 0 2 3 0 6 3
- Working set size: 3
- FIFO (10 page faults)

0 1 5 | 0
0 1 5 | 1
0 1 5 | 4
1 5 4 | 0
5 4 0 | 1
4 0 1 | 2
0 1 2 | 0

0 1 2 | 0
0 1 2 | 2
0 1 2 | 3
1 2 3 | 0
2 3 0 | 6
3 0 6 | 3
3 0 6 |

Example for Page Replacement Policies

- Page use pattern: 0 1 5 0 1 4 0 1 2 0 2 3 0 6 3
- LRU (7 page faults)
  - Minimal page faults required: 7

0 1 5 | 0
1 5 0 | 1
5 0 1 | 4
0 1 4 | 0
1 4 0 | 1
4 0 1 | 2
0 1 2 | 0

0 1 2 | 0
1 2 0 | 2
1 0 2 | 3
0 2 3 | 0
2 3 0 | 6
3 0 6 | 3
0 6 3 |

Least recently used ← Most recently used

Example for Page Replacement Policies

- Page use pattern: 0 1 5 0 1 4 0 1 2 0 2 3 0 6 3
- Clock (9 page faults)
  - * represents used bit, ↓ next to the last loaded pages

0*	1	4* 1 5	0	2* 0* 1	3
0* 1*	5	4* 0* 5	1	2* 0 3*	0
0* 1* 5*	0	4* 0* 1*	2	2* 0* 3*	6
0* 1* 5*	1	2* 0 1	0	6* 0 3	3
0* 1* 5*	4	2* 0* 1	2	6* 0 3*	

Example for Page Replacement Policies

- Page use pattern: 0 1 5 0 1 4 0 1 2 0 2 3 0 6 3
- Clock (9 page faults)
  - * represent ... aded pages

Scan and find page with (used = 0)
While scanning, change (used → 0)
⇒ 0 1* 5* | 4
⇒ 0 1 5* | 4
⇒ 0 1 5 | 4
Finish one round, no page selected
Start again, page 0 selected
⇒ 4* 1 5 |

0*	1		
0* 1*	5		
0* 1* 5*	0		
0* 1* 5*	1		0	6* 0 3	3
0* 1* 5*	4	2* 0* 1	2	6* 0 3*	

Example for Page Replacement Policies

Scan and find page with (used = 0)
Page 1 selected
⇒ 4* 0* 5 |

) 1 4 0 1 2 0 2 3 0 6 3

- * represents us[...], ↓ next to the last loaded pages

0* | 1	4* 1 5 | 0	2* 0* 1 | 3
0* 1* | 5	4* 0* 5 | 1	2* 0 3* | 0
0* 1* 5* | 0	4* 0* 1* | 2	2* 0* 3* | 6
0* 1* 5* | 1	2* 0 1 | 0	6* 0 3 | 3
0* 1* 5* | 4	2* 0* 1 | 2	6* 0 3* |

Example for Page Replacement Policies

- Page use pattern: 0 1 5 0 1 4 0 1 2 0 2 3 0 6 3
- Clock (9 page faults)
  - * represents used bit, ↓ ne[...]

Scan and find page with (used = 0)
While scanning, change (used → 0)
⇒ 4 0 1 | 2
Finish one round, no page selected
Start again, page 0 selected
⇒ 2* 0 1 |

0* | 1	4* 1 5 | 0	
0* 1* | 5	4* 0* 5 | 1	
0* 1* 5* | 0	4* 0* 1* | 2	2* 0* 3* | 6
0* 1* 5* | 1	2* 0 1 | 0	6* 0 3 | 3
0* 1* 5* | 4	2* 0* 1 | 2	6* 0 3* |

Example for Page Replacement Policies

- Page use pattern: 0 1 5 (
- Clock (9 page faults)
  - * represents used bit,

Scan and find page with (used = 0)
While scanning, change (used → 0)
⇒ 2* 0 1 | 3
Page 1 selected
⇒ 2* 0 3* |

0* | 1
0* 1* | 5
0* 1* 5* | 0
0* 1* 5* | 1
0* 1* 5* | 4

4* 1 5 | 0
4* 0* 5 | 1
4* 0* 1* | 2
2* 0 1 | 0
2* 0* 1 | 2

2* 0* 1 | 3
2* 0 3* | 0
2* 0* 3* | 6
6* 0 3 | 3
6* 0 3* |

Example for Page Replacement Policies

- Page use/write pattern
  0 1w 5 0 1w 4w 0 2 0 1w 2w 4
- Modified Clock Policy (x represents modified)

0* | 1w
0* 1*x | 5
0* 1*x 5* | 0
0* 1*x 5* | 1w

0* 1*x 5* | 4w
4*x 1x 5 | 0
4*x 1x 0* | 2
4x 2* 0* | 0

4x 2* 0* | 1w
1*x 2* 0 | 2w
1*x 2*x 0 | 4
1*x 2*x 4* |

Scan and find page with (used=0, dirty=0)
⇒ 0* 1*x 5* | 4w → no page selected
2nd scan, find page with (used=0, dirty=1)
While scanning, change used bit
⇒ 0 1x 5 | 4w → still no page selected
Scan and find page with (used=0, dirty=0)
Page 0 selected
⇒ 4*x 1x 5 |

Ex... ment Policies

- P... 2w 4
- M... modified)

0* | 1w
0* 1*x | 5
0* 1*x 5* | 0
0* 1*x 5* | 1w

0* 1*x 5* | 4w
4*x 1x 5 | 0
4*x 1x 0* | 2
4x 2* 0* | 0

4x 2* 0* | 1w
1*x 2* 0 | 2w
1*x 2*x 0 | 4
1*x 2*x 4* |

Example for Page Replacement Policies

Scan and find page with (used=0, dirty=0)
Page 5 selected
⇒ 4*x 1x 0* |

- Page use...
  0 1... 4
- Modifie... lified)

0* | 1w
0* 1*x | 5
0* 1*x 5* | 0
0* 1*x 5* | 1w

1*x 5* | 4w
4*x 1x 5 | 0
4*x 1x 0* | 2
4x 2* 0* | 0

4x 2* 0* | 1w
1*x 2* 0 | 2w
1*x 2*x 0 | 4
1*x 2*x 4* |

Example for Page Replacement Policies

- Page use/wr
  0 1w 5
- Modified C

Scan and find page with (used=0, dirty=0)
⇒ 4*x 1x 0* | 2 → no page selected
2nd scan, find page with (used=0, dirty=1)
While scanning, change used bit
⇒ 4x 1x 0* | 2 → page 1 selected
⇒ 4x 2* 0* | → write page 1 to disk

0* | 1w	x 5* | 4w	4x 2* 0* | 1w
0* 1*x | 5	4 1x 5 | 0	1*x 2* 0 | 2w
0* 1*x 5* | 0	4*x 1x 0* | 2	1*x 2*x 0 | 4
0* 1*x 5* | 1w	4x 2* 0* | 0	1*x 2*x 4* |

Example for Page Replacement Policies

- Page use/write pattern
  0 1w 5 0 1w 4v
- Modified Clock Policy

Scan and find page with (used=0, dirty=0)
⇒ 4x 2* 0* | 1w → no page selected
2nd scan, find page with (used=0, dirty=1)
While scanning, change used bit
⇒ 4x 2* 0 | 1w → page 4 selected
Write page 4 back to disk, replace it
⇒ 1*x 2* 0 |

0* | 1w	0* 1*x 5* | 4w	4x 2* 0* | 1w
0* 1*x | 5	4*x 1x 5 | 0	1*x 2* 0 | 2w
0* 1*x 5* | 0	4*x 1x 0* | 2	1*x 2*x 0 | 4
0* 1*x 5* | 1w	4x 2* 0* | 0	1*x 2*x 4* |

Aging Policy

- Aging policy (global page replacement)
  - Increase used field whenever it is accessed
    - Set the leftmost bit of the age to 1
  - System periodically scan the memory and reduce the "used" value
    - Right shift the age value
  - A page is to be removed if it reaches 0
    - Does not need to actually remove till receiving a replacement request
    - Replace a page with the lowest age number
  - Special hardware
    - Special memory for the age vector, able to shift, set, select min

Aging Policy

- Aging policy
  - Example
    - 3 pages in memory, page use pattern: 0 1 5 0 1 4 0 1 2 0 2 3 0 6 3
    - 8-bit age vector, scan every 1 sec (each sec has 1 page accesses)
  - Result
    - Start: 00000000 (-) for all frames
    - Access 0: 10000000 (0), 00000000 (-), 00000000 (-)
    - Access 1: 01000000 (0), 10000000 (1), 00000000 (-)
    - Access 5: 00100000 (0), 01000000 (1), 10000000 (5)
    - Access 0: 10010000 (0), 00100000 (1), 01000000 (5)
    - Access 1: 01001000 (0), 10010000 (1), 00100000 (5)
    - Access 4: 00100100 (0), 01001000 (1), 10000000 (4)
      - Choose the page with the lowest age to replace

Other Page Replacement Considerations

- Global page replacement
  - Consider all processes together
  - E.g., Linux page replacement policy
    - Aging policy with 8-bit aging vector
- Load Control
  - How many processes should reside in memory
    - In global policy, this cannot be controlled by working set size
  - Control the level of multiprogramming
  - Too many processes → high frequency of page faults
  - Too few processes → reduced level of concurrency

Other Issues in VM

- Frame Locking
  - Lock pages that cannot be replaced
    - e.g. OS code and data
  - When a page is just brought in and before it is used
    - Process A is switched out when having a page fault
    - When the page is brought in, A may not be scheduled to run

Other Issues in VM

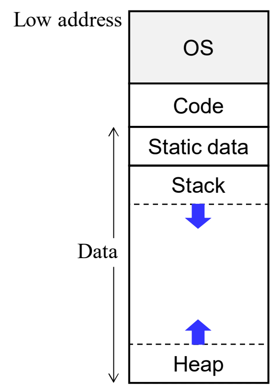
- Page sharing
  - In many situations, multiple address spaces may have exactly the same pages
    - E.g., library functions, kernel functions
    - Only one copy is needed in physical memory
    - Page tables of all the processes for the shared page point to the same memory frame
  - Copy on write
    - When a process writes to the shared page, then the shared page is copied and a private copy is given to the process
    - Save physical memory space as well as copying time

Other Issues in VM

- Page sharing
  - Fork: child processes initially have exactly the same copy as the parent process
  - No need to do actual copy unless there is a write
    - What needs to be copied?
  - Frequently exec follows fork, and all the copying after fork is wasted
  - With copy on write, the saving can be significant

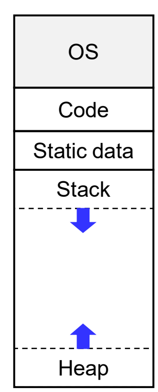
Other Issues in VM

- Protection
  - How to ensure that each process accesses its own memory
    - Is it a problem in virtual memory?
    - Why there are access violations?
  - Memory layout
    - Stack: call stack
    - Heap: dynamic references
      - E.g., those allocated by malloc
    - OS: in the address space of every process
      - No context switch for kernel mode
      - Share the same frames by all processes
      - Use copy on write when data differs
  - Frame metadata
    - Accessed, modified, valid, read only, mode (user/kernel)



Other Issues in VM

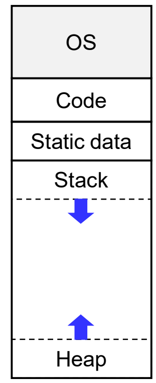
- Protection
  - Is the entire address space mapped in page table?
    - The empty region between stack and heap is not
      - No page for it (neither in memory nor in swap space)
    - OS allocate a certain size of memory for stack
      - Can be changed, but generally are fixed
    - Heap is requested page by page
      - When malloc, if no space in existing heap page, system call sbrk/mmap is invoked to increase the heap space, generally it is page aligned
      - A marker marks the boundary of the heap space sbrk/mmap moves it
      - sbrk/mmap can cause a new entry (or new entries) being inserted to the page table



Other Issues in VM

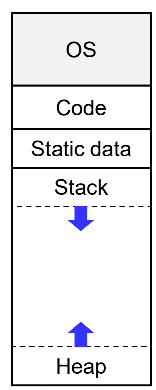
- Protection
  - Why there are access violations?
    - If accessing OS region
      - Page table points to shared OS frames
        - The mode bit for these frames are set to "kernel mode"
      - These OS frames can only be accessed in kernel model
        - Set the mode register to kernel mode
        - Only allowed in system calls, exception handlers, etc.
    - If accessing the empty region
    - If accessing address 0
      - Address 0 is frequently reserved specifically in the address space for null
    - Even though the above are in the address space of the process!!!

Why put OS code in user space?
⇒ Avoid context switch
⇒ Need to switch to kernel mode



Other Issues in VM

- Protection
  - Call stack
    - When there is a procedure call, the in/out parameters, the local data, and a return address are put into stack
  - Stack buffer overflow
    - A procedure with a local buffer is put into stack
    - Caller provides input longer than the buffer size
    - May put malicious code in the buffer, longer input overwrites the return address, causing branching into the malicious code
    - Can attack the system, but only if the process being attacked by buffer overflow has the privilege



Other Issues in VM

- Protection
  - Zeroing pages
    - When a page is brought into memory for a process, the entire frame is zeroed to ensure that one process does not get to read another process's previous memory content
      - Because zeroing is expensive, it can be done in a lazy way

Readings

- Sections 1.5, 1.6
- Sections 8.1, 8.2