

Virtual Memory

- What is virtual memory, and why do we need it?
- Demand paging
 - » Description
 - » Performance
- Page replacement: when it doesn't all fit into memory
 - » Algorithms
 - » Performance
- Allocating frames to processes
- Performance issues
 - » Thrashing
 - » Page size selection
 - » Writing efficient programs
- Virtual memory and segments

What is Virtual Memory?

- Virtual memory separates logical memory from physical memory
 - » Keep only the “active” code & data for a program in memory
 - » Keep the remainder of the pages on disk, swapping them in and out as necessary
 - » Allow a logical address space much larger than the available physical memory
 - » Allows more programs to run by keeping inactive pieces of code & data on disk
- Virtual memory can be used with
 - » Paging
 - » Segmentation

Demand Paging

- Simplest form of paging is demand paging: bring a page into memory only when it's actually referenced
 - » Requires less I/O (don't get pages until they're used)
 - » Requires less memory (not wasted on unused pages)
 - » More users (less memory per process)
 - » Faster response on process startup (no need to load entire program before running)
- On reference to page not in memory:
 - » If not in memory (but on disk), fetch into memory
 - » If invalid reference (dereferencing NULL pointer), abort process

Valid Bit in Page Table

- Associate a valid bit with each page table entry
 - » 1 => page in memory
 - » 0 => page not in memory
- Set all valid bits to 0 initially
- During address translation, check valid bit
 - » 1 => translation proceeds normally
 - » 0 => page fault exception (instruction doesn't finish)

frame #	valid bit
0x41f	1
0x241	1
0x03e	1
	0
0x12a	1
⋮	
	0
	0
0x3e0	1

How a Page Fault Works

- On reference to invalid page ($\text{valid} == 0$), cause an exception called a page fault
- During a page fault, OS must decide if reference is valid
 - » Use another bit from the page table entry (not the valid bit)
 - » Check the “frame number” to see if the page is on disk (for example, 0 == page not on disk either)
- OS allocates an empty page frame
- Read the page from disk into the just-allocated frame
- Modify the page table
 - » Insert the real frame number into the page table entry
 - » Change the valid bit from 0 to 1
- Restart the instruction
 - » Can be difficult with complex instructions!

What If There's No Free Frame?

- After a system has been running for some time, all frames will be allocated if there's no way to free them
 - » Exiting processes free their frames
 - » How can the OS claim frames from still-running processes?
- Page replacement
 - » Find a page in memory that's not in active use
 - » Swap it out (move it to disk)
 - » Mark the page table entry as invalid again
 - » A given page may be fetched into memory multiple times
- Page replacement algorithms
 - » Decide which page in memory to swap to disk
 - » Critical for good performance: OS wants to minimize page faults

Demand Paging Performance

- Assume the following numbers
 - » Page fault rate p ($0 \Rightarrow$ no page faults)
 - » Page fault time f , composed of
 - Exception overhead ($\sim 1-10$ us)
 - Time to swap the old page out (~ 10 ms)
 - Time to swap the new page in (~ 10 ms)
 - Instruction restart overhead ($\sim 1-10$ us)
 - » Memory access time t
- Effective access time for memory is then:
$$\text{EAT} = (1-p) * t + p * f$$

Demand Paging Performance: Example

- Basic performance numbers:
 - » Memory access time = 100 ns
 - » Disk access time = 10 ms
 - » Page being replaced is modified 40% of the time
 - Only needs to be written to disk if it's modified
 - » Page fault overhead is 20 us (excluding disk I/O time)
 - » Page faults occur every 100,000 instructions
- Page fault time
 - » $10000 \text{ us} * (1 + 40\% * 1) + 20 \text{ us} = 14020 \text{ us}$
- Effective access time
 - » $\text{EAT} = p * 14020 + (1-p) * 0.100$
 - » $\text{EAT} = 10^{-5} * 14020 + (1 - 10^{-5}) * 0.100 = 0.24 \text{ us} = 240 \text{ ns}$

Page Replacement

- Modify page fault handler to include page replacement
 - » Prevents over-allocation of memory
 - » Centralizes code in one place
- Use *modified (dirty) bit* to keep track of changed pages
 - » Set bit to 0 each time page is swapped into memory
 - » Set bit to 1 each time page is written to
 - » Only swap to disk when page is “dirty” (dirty bit == 1)
- Store dirty bit in the page table entry
 - » TLB entries include the dirty bit and other bits used by paging
 - » Changes to status bits (such as dirty bit) occur only in TLB entry
 - » TLB entries written back to page table when the entry is replaced in the TLB

Page Replacement Algorithms

- A page replacement algorithm chooses the page to be swapped out of memory
- Goal: get the lowest page fault rate
 - » Lower page fault rates mean better performance
 - » Compare algorithm's page fault against “optimal” algorithm
- Evaluate the algorithm by running it on a particular sequence of memory (page) references
 - » Compute the number of page faults on that sequence
 - » Compare the page fault rate with other algorithms, including the optimal algorithm
- For these examples, we'll use the reference string
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Optimal Algorithm

- Optimal algorithm is the goal all other algorithms are measured against
- Rule: replace page that will not be used for the longest period of time
 - » Impossible to do in real OS - requires future knowledge
 - » Other algorithms attempt to figure out which page will be used furthest in the future, but may guess wrong
- Useful as a baseline for other algorithms' performance

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

1	4
2	
3	
4	5

First-In-First-Out (FIFO) Algorithm

- Rule: replace the page that has been in memory the longest
- Simple to implement
 - » Keep a circular list of frames
 - » Update a pointer to the next frame to be replaced
- Belady's Anomaly: more frames can result in more (not fewer) page faults!
 - » May be present for FIFO replacement
 - » Can't be present for LRU replacement

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

1	4	5
2	1	3
3	2	4

9 page faults

1	5	4
2	1	5
3	2	
4		3

10 page faults

Least Recently Used (LRU) Algorithm

- Rule: replace the page that was used least recently
- Implementation
 - » Keep a counter in each page table entry
 - » Copy the current clock into the PTE when a reference is made
 - » Search PTEs for the lowest clock value to find the page to replace
- Can be slow in real OS
 - » Searching through all those PTEs takes time!
 - » Updating PTEs with the current clock is slow

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

1			5
2			
3	5		4
4		3	

Better LRU Implementation?

- Address problems with previous LRU implementation
 - » Searching for frame to replace is slow (requires scan of all PTEs)
 - » Updating PTE with current clock can be slow w/o hardware help
- Use a doubly-linked list of page numbers, where pages at top of list have been used recently
 - » Move page to top of list when it's referenced
 - » No search for replacement
 - » May be slow: 6 pointers changed on page reference
- Better still: combine the two methods
 - » Use clock for entries in TLB
 - » Update list in memory only when PTE is replaced in TLB

Approximating LRU

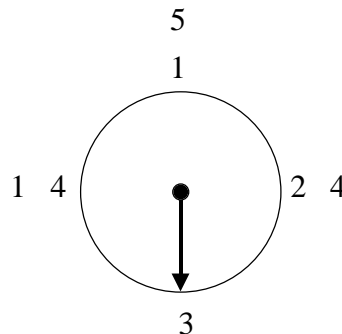
- Problem: LRU can be slow and needs hardware help
- Use a reference bit in each PTE
 - » Reference bit initially set to 0
 - » Reference bit set to 1 when page is referenced
 - » Replace a page with bit set to 0 if such a page exists
 - May not be the least recently used
- Use the reference bit to approximate LRU
 - » Second chance replacement
 - » Clock algorithm

Clock Algorithm

- Keep a circular list of page frames in memory
- Keep a pointer to a location in the circular list
- Rule:

```
while (not done) {  
  if (ref bit == 1) {  
    replace this frame  
  } else {  
    set ref bit = 0  
    leave page in memory &  
    advance to next frame  
  }  
}
```
- Set ref bit = 1 when page is referenced

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



Page Replacement in Unix

- Problem with clock algorithm: slow replacement (dirty page must be written out before new page brought in)
- Keep a circular list of frames as with clock algorithm
 - » Go through list of frames at a fixed rate
 - If frame's reference bit is 0
 - Write it to disk (if dirty) and clear the dirty bit
 - Place it into a pool of "available" pages & set valid bit in PTE to 0
 - If frame's reference bit is 1, set the bit to 0
- On a page fault
 - » Search available pool for the desired page - if found, mark PTE as valid and remove page from available pool
 - » If not found, replace any page in available pool with page fetched from disk

Counting-Based Algorithms

- Keep a counter of the number of references that have been made to each page
- LFU algorithm: replace page with the smallest count
- MFU algorithm: replace page with largest count, since page with smallest count may have just been brought in and will be used more in the future
- Problem: counts can only increase
- Solution: periodically go through PTEs and reduce counters
 - » Set counters back to 0
 - » Reduce counters by a factor of n

Allocating Frames to Processes

- Each process needs a minimum number of pages
 - » A single instruction might access more than one page
 - » Example:
VAX instruction add.w 70000(r1), 70000(r2), 70000(r3)
 - Instruction might span 2 pages
 - Each operand might span 2 pages
 - Total page faults = $2 + 3 * 2 = 8$ page faults!
 - » System must make sure that a process can execute such an instruction if necessary
- Two basic allocation schemes:
 - » Fixed allocation
 - » Priority allocation

Fixed Allocation of Frames

- Equal allocation: allocate frames to processes evenly
 - » If there are 256 frames and 8 processes, allocate 32 frames per process
- Proportional allocation: allocate frames according to the size of the process (bigger process => more frames)
 - » Add up the total number of frames required by all running processes
 - » Divide by the total number of available frames for paging
 - » Divide each process's total frames by the result to compute the total allocation for that process (but allocate minimum #)
 - » Example: p1 -> 100 frames, p2 -> 20 frames, p3 -> 40 frames
40 frames available
p1 gets $100 * 40 / (100 + 40 + 20) = 25$ frames
p2 gets $20 * 40 / 160 = 5$ frames
p3 gets $40 * 40 / 160 = 10$ frames

Priority Allocation of Frames

- Allocate frames to processes proportionally using priorities rather than fixed allocation (size)
 - » Priority = number of recent page faults
 - » Priority = more “important” process
- If process P_i generates a page fault
 - » Select one of its own frames for replacement
 - » Select a frame from a process with a lower priority

Local vs. Global Frame Allocation

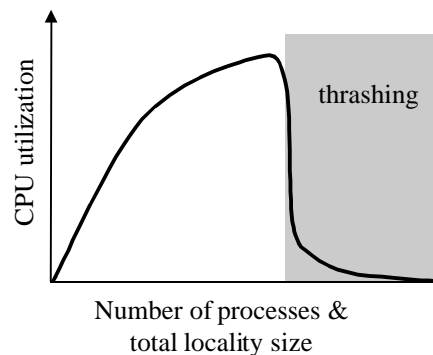
- Local replacement
 - » Replacement algorithm is run only on frames allocated to the process
 - » More “suitable” frames in other processes are ignored
 - » Easier bookkeeping
 - » Penalizes processes that behave poorly (too many page faults)
- Global replacement
 - » Replacement algorithm run over entire set of frames in all processes
 - » Can be slower than local replacement
 - » Usually leads to globally better behavior
 - » Individual processes may be slowed down unnecessarily

Thrashing in Virtual Memory

- Not “enough” pages leads to very high page fault rates
 - » Low CPU utilization - CPU always waiting for pages from disk
 - » Potential OS problems
 - OS sees low CPU utilization
 - OS adds another process to better utilize CPU
 - Thrashing gets even worse as there are more processes fighting over too little memory
- Thrashing describes a process (or system) that spends all of its time swapping pages in and out
- How few is “too few” pages for a process?

Characterizing Thrashing

- Paging works because of locality
 - » Locality is the pages “currently” in use by a process
 - » Process moves from one locality to another
 - » Localities may overlap
- Thrashing occurs when the current localities for all processes don’t fit into physical memory
 - » CPU utilization drops
 - » I/O rate increases



Avoiding Thrashing: Working Sets

- Working set is the group of pages “currently” in use by a given process
 - » “Currently” means the page was accessed within the past n instructions by this process
 - » The n instruction window is called the working set window
- WSS_i (P_i 's working set) is all of the pages accessed in the most recent n instructions
 - » If window is too small, it won't encompass the current locality
 - » If window is too large, it will encompass several localities and possibly waste space
 - » If window is infinitely large, it will include the whole program
- Goal: keep the sum of the sizes of WSS_i below the total available physical memory
- If all WSS_i won't fit into memory, suspend a process

Calculating Working Set Size

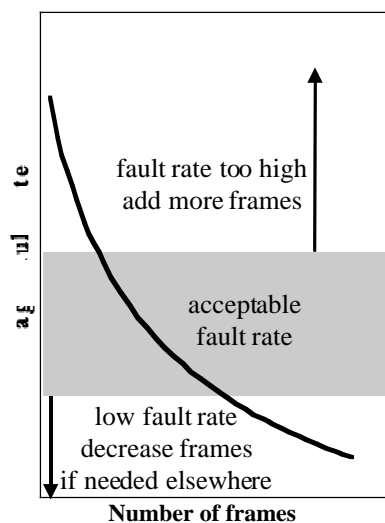
- Exact calculation of working set size is difficult
- Approximate calculation with an interval timer, reference bit per page, and history of reference bits for all pages
 - » Current reference bit set to 1 on page reference
 - » History (values at previous interrupts) also stored in PTE
- For example, working set window is 100,000 instructions
 - » Timer interrupts every 100,000 instructions
 - » For each page in the process
 - Shift history bits left by 1 position
 - Insert the value of the current reference bit
 - Set the current reference bit to 0
 - » If any history bits are 1, page is in current working set

Issues With Working Sets

- Method for calculating working set has problems
 - » May not be completely accurate
 - » May be somewhat slow
- Interrupt more often
 - » Better accuracy
 - » Slower
- Keep more history bits
 - » Better accuracy
 - » More space required
- Use clock-based methods (discussed earlier) to approximate working set algorithms
- Use page-fault frequency to figure out working set size

Using Page Fault Frequency

- Each process has an acceptable page fault rate
 - » If actual rate is higher, add more frames
 - » If actual rate is lower, remove frames
- Don't remove frames unless other processes need them!
 - » Low page fault rate is good!
 - » Allow processes to keep pages until they're needed elsewhere



Prefetching Pages

- Demand paging is good, but may result in excessive delay
 - » User must wait while process fetches needed pages
 - » Solution: fetch pages *before* they're needed
- Prefetch pages that may be needed soon
 - » Pages near the current page (1-2 pages ahead or behind)
 - » Pages the program says it will need soon
- Advantage: less delay seen by process & user (need pages already in memory)
- Disadvantage: may replace useful pages with pages that will never be used

Selecting a Page Size

- Consider many issues when selecting a page size
 - » May not be a single “optimal” page size
 - » Tradeoffs between different factors
- Fragmentation
 - + Smaller pages have less internal fragmentation
- Page table size
 - + Larger pages require smaller page tables
- I/O overhead
 - + Larger pages have less I/O overhead per byte
 - + Smaller pages require less I/O for a single page fault
- Locality
 - + Larger pages require fewer fetches for a given locality
 - + Smaller pages include less “inactive” memory

User Programs and Virtual Memory

- Program code interacts with virtual memory system

- » Interaction hidden from user
- » Performance problems *not* hidden!

- Program to add two 2-D arrays together

- » Arrays stored in C order
- » Page size is 4 KB
- » Only 2 data frames available to process

- Loop order makes a *huge* difference in performance!

```
int X[1024][1024], Y[1024][1024];
```

```
for (j=0; j<1024; j++) {  
    for (k=0; k<1024; k++) {  
        X[j][k] += Y[j][k];  
    }  
}
```

X: 1024 page faults

Y: 1024 page faults

```
for (k=0; k<1024; k++) {  
    for (j=0; j<1024; j++) {  
        X[j][k] += Y[j][k];  
    }  
}
```

X: 1024 x 1024 page faults

Y: 1024 x 1024 page faults

Demand Segmentation

- Some systems can support segments but not paging
 - » Trap when unloaded segment is accessed
 - » No address translation for paging
- Use methods similar to those used in paging
 - » Valid bit in segment descriptor to indicate a segment loaded into memory
 - » Segment fault if segment not in memory
 - Segment loaded into memory (space found by looking through list of memory holes)
 - Segment table updated to show valid segment
 - » Deallocate segments no longer in use