Memory Management
(Virtual Memory)

Mehdi Kargahi
School of ECE
University of Tehran
Spring 2008
Background

- It is not required to load all the program into main memory
- Virtual memory separates logical memory from physical memory
Demand Paging

- Pages are only loaded when they are demanded during program execution (using a *pager*).
- Swapper: manipulated entire processes.
- Pager: manipulates individual pages of a process.
Demand Paging

- If the pager guesses right the required pages at load time the execution can proceed normally.
Handling a Page Fault
Handling a Page Fault

- Whether are all instructions restartable?
  - IBM 370 → MVC with overlapping addresses
  - VAX & Motorola 68000 → MOV (R2)+, -(R3)
    - The microcode computes and attempts to access both ends of source and destination blocks
    - Using temporary registers to hold the values of overwritten locations

- Locality of reference for instructions

- The performance of demand paging can highly affect the feasibility of using paging in a system
Performance of Demand Paging

- Effective access time $= (1-p) \times ma + p \times \text{page fault time}$
- Major components of page fault service time
  - Service the page fault interrupt
  - Read in the page
  - Restart the process
Effective access time = \((1-p) \times ma + p \times \text{page fault time}\)

Computing the page fault time

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:
   a. Wait in a queue for this device until the read request is serviced.
   b. Wait for the device seek and/or latency time.
   c. Begin the transfer of the page to a free frame.
6. While waiting, allocate the CPU to some other user (CPU scheduling, optional).
7. Receive an interrupt from the disk I/O subsystem (I/O completed).
8. Save the registers and process state for the other user (if step 6 is executed).
9. Determine that the interrupt was from the disk.
10. Correct the page table and other tables to show that the desired page is now in memory.
11. Wait for the CPU to be allocated to this process again.
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction.
Performance of Demand Paging

- **Effective access time** = \((1-p) \times ma + p \times \text{page fault time}\)

- **Major components of page fault service time**
  - Service the page fault interrupt (1 to 100 \(\mu\)s)
  - Read in the page: 8 ms (ignoring device queueing time)
    - Hard disk latency: 3 ms
    - Seek time: 5 ms
    - Transfer time: 0.05 ms
  - Restart the process (1 to 100 \(\mu\)s)
  - Assuming \(ma = 200\) ns

\[
\text{effective access time} = (1 - p) \times 200 + p \times 8,000,000 \\
= 200 + 7,999,800 \times p.
\]

- \(P = 0.001\) \(\Rightarrow\) \(eat = 8.2\ \mu\)s \(\Rightarrow\) computer is slowed down by a factor of 40

- For 10% slow down: \(220 > 200 + 7,999,800 \times p\) \(\Rightarrow\) \(p < 0.0000025\)

M. Kargahi (School of ECE)
An additional aspect of demand paging is the handling and overall use of swap space. Disk I/O to swap space is generally faster than that to the file system. It is faster because swap space is allocated in much larger blocks, and file lookups and indirect allocation methods are not used (Chapter 12). The system can therefore gain better paging throughput by copying an entire file image into the swap space at process startup and then performing demand paging from the swap space. Another option is to demand pages from the file system initially but to write the pages to swap space as they are replaced. This approach will ensure that only needed pages are read from the file system but that all subsequent paging is done from swap space.

Some systems attempt to limit the amount of swap space used through demand paging of binary files. Demand pages for such files are brought directly from the file system. However, when page replacement is called for, these frames can simply be overwritten (because they are never modified), and the pages can be read in from the file system again if needed. Using this approach, the file system itself serves as the backing store. However, swap space must still be used for pages not associated with a file; these pages include the stack and heap for a process. This method appears to be a good compromise and is used in several systems, including Solaris and BSD UNIX.
Copy-On-Write
Need for Page Replacement
Basic Page Replacement

- Page fault and no free frame

M. Kargahi (School of ECE)
Basic Page Replacement

- Reducing the overhead
  - Modify (Dirty) bit: Page not modified and read-only pages need not be written to the disk

- Two important matters
  - How many frames should be allocated to each process?
  - Which page should be replaced to reduce the page fault probability?

- Reference string
Reference String

- Address sequence
  0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105

- Page size: 100 bytes
- Reference string:  1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1
- 3 frames for the process  ➔ 3 page faults
- 1 frame  ➔ 11 page faults

![Graph showing the relationship between number of frames and number of page faults]
Page Replacement Algorithms

- Sample reference string:
  
  7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

- **FIFO page replacement**

  Reference string:
  
  7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

  Page frames:
  
  7 7 7 2 2 2 4 4 4 0 0 0 0 3 3
  0 0 1 1 2 2 3 3
  2 2 1

- 15 page faults
Belady’s Anomaly and FIFO Page Replacement

- Sample reference string:
  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

- Page faults with 1, 2, 3, 4, 5, and 6 pages?
Optimal Page Replacement

- Reduces 15 page faults of FIFO to 9 page faults
- Difficult to implement
- Mainly used for comparison study
- Belady’s anomaly?
LRU Page Replacement

- **LRU: Least-Recently Used**

Reference string:

```
7  0  1  2  0  3  0  4  2  3  0  3  2  1  2  0  1  7  0  1
```

Page frames:

```
7  7  7  2  2  4  4  4  0  1  1  1
0  0  0  0  0  3  3  3  3  1  0  0
1  1  1  1  3  2  2  2  2  1  1  1
```

- **12 page faults**
- **Implementation:** according to the last time of access to each page!!!
LRU Implementation

- Hardware support is required
- Counters
  - A logical clock or counter is incremented on each memory access
  - This value is copied in the time-of-use field in the page table entry
- Problems
  - Search for the smallest time value
  - A write to memory for each memory access
  - Maintaining the information when the page table is changed due to CPU scheduling
  - Clock overflow
LRU Implementation

- Stack
- Stack is implemented as a doubly linked-list
- Each update requires changing 6 pointers
- The page number at the bottom of the stack is the LRU page
- Belady’s anomaly?
  - The set of pages in $n$ frames is a subset of the set of pages in $n+1$ frames

M. Kargahi (School of ECE)
LRU-Approximation Page Replacement

- The updating of clock fields or stack for every memory reference requires an interrupt!!
- Few computers provide sufficient hardware for implementing LRU
- A basic method
  - Using a reference bit
    - Hardware should set this bit on each access
    - This method does not specify the order of using the pages
Additional-Reference-Bits Algorithm

- Recording the reference bits at regular intervals
- Assume an 8-bit byte for each page in a table in memory
- At regular intervals, a timer interrupt activates the OS to shifts the reference bit into the high-order bit of each page
- The corresponding register contains the history of accessing the page for the last 8 time periods
- The smaller the unsigned value of the register, the LRU page

Problems
- Searching among the values
- Equal values will result an approximation of LRU
  - Replace all equal smallest values or according to FIFO order
Second-Chance Algorithm

- Degenerates to FIFO if all bits are set
Enhanced Second-Chance Algorithm

- Considering both reference-bit and modify-bit

1. (0, 0) neither recently used nor modified—best page to replace
2. (0, 1) not recently used but modified—not quite as good, because the page will need to be written out before replacement
3. (1, 0) recently used but clean—probably will be used again soon
4. (1, 1) recently used and modified—probably will be used again soon, and the page will need to be written out to disk before it can be replaced

- The circular queue may have to be scanned several times before finding a page to be replaced
Counting-Based Page Replacement

- LFU (Least-Frequently Used)
  - Problem: A page is used heavily during the initial phase but then is never used again
  - Solution: Shift the counts right by one bit at regular intervals

- MFU (Most-Frequently Used)
  - The page with the smallest count was probably just brought in and has yet to be used

- Properties
  - Expensive to implement
  - Weakly approximate OPT
Page-Buffering Algorithms

- Beside page-replacement algorithms
- Systems commonly keep a pool of frames
  - Page fault ➔ Selecting a victim frame ➔ Read in the page into a free frame before writing out the victim to restart the process sooner ➔ When the victim is later written out, the free frame is added to the pool
  - *Expansion*: Maintaining a list of modified pages ➔ Writing out a modified page whenever the paging device is idle and reset the modify bit ➔ Higher probability that a page will be clean.
Study Carefully

- Section 9.4.7
- Section 9.4.8
Frame Allocation

- The minimum number of frames for each process depends strongly on the system architecture:
  - A machine that all memory-reference instructions have only one memory address
    - Requires 2 pages, at least
  - A machine that all memory-reference instructions can have one indirect memory address
    - At least 3 pages is required
  - A machine that memory-reference instructions can have two indirect memory address
    - At least 6 pages is needed
  - What if the instructions can be of more than one word?
  - What about multiple levels of indirection? Any limit?
  - Minimum number of frames/process is defined by the architecture
  - Maximum number of frames/process is defined by the amount of available physical memory (If all memory is used for indirection)
Frame Allocation Algorithms

- $m$ frames, $n$ processes
- *Equal allocation*: $m/n$ frames per process
- *Proportional allocation*:
  - $s_i$: size of the virtual memory for process $p_i$
  - $S = \sum s_i$
  - $a_i = s_i/S \times m$
  - $a_i$ should be larger than minimum number of frames per process
- Priority can also be a parameter for frame allocation
Global vs. Local Allocation

- **Global replacement**: replaces a page among all pages in the memory
  - A process cannot manage its own page fault rate
  - Higher throughput

- **Local replacement**: replaces a page of itself
  - Does not use less used pages of memory
Thrashing

- A process is thrashing if it is spending more time paging than executing.
- If the number of frames of a process falls below the number of required frames by the computer architecture, it will thrash.
Example

1. Assume global page replacement
2. Assume that page faults result in some processes to wait on the paging device
3. CPU utilization decreases
4. Increasing the degree of multiprogramming
5. The new process requires page and encounters page fault, therefore waits on the paging device
6. CPU utilization decreases …
Using local page replacement, if one process starts thrashing, it cannot steal frames from other processes.

Thrashing results in increased waiting time on the paging device which affects the effective access time of memory even for processes which are not thrashing.

To prevent thrashing a process should have its required frames.

How do we know how many frames it needs?

- Working-Set Model
- Page-Fault Frequency
Locality in a Memory Reference Pattern
Locality in a Memory Reference Pattern

- If we allocate fewer frames than the size of the current locality, the process will thrash, since it cannot keep in memory all the pages that it is actively using.
Working-Set Model

- $\Delta$: Working-set window
- Working set: the most recent $\Delta$ page references
  - An approximation of the program’s locality
  - Ex.: $\Delta=10$

![Page reference table]

- Large $\Delta$: overlapping several localities
- Small $\Delta$: Doesn’t show the entire locality
Working-Set Model

- $WSS_i$: Working-set size of process $p_i$
- Total demand for frames $\Rightarrow D = \Sigma WSS_i$
- $D > m \Rightarrow$ Thrashing
- The OS will assign frames to each process according to its working-set size which can be obtained by looking at $\Delta$
- If free frames are ready $\Rightarrow$ another process can be initiated
- If $D > m \Rightarrow$ the OS should suspend some processes

M. Kargahi (School of ECE)
Page-Fault Frequency (PFF)

- Page-fault rate $> \text{upper bound} \implies$ increasing the frames of this process
- Page-fault rate $< \text{lower bound} \implies$ decreasing the frames of this process
Page-Fault Rate over Time
Memory-Mapped Files & I/O

- Mapping disk blocks to pages for memory-mapped files

- Memory-mapped I/O:
  - Setting ranges of memory addresses to special device registers
    - E.g., a few device registers are called I/O ports (serial and parallel)
    - Polling a control bit in the memory words to check if the I/O device has received the data: Programmed I/O
    - Using interrupts to do so: Interrupt-driven
Allocating Kernel Memory

- Kernel memory is often allocated from a free memory pool, in a different manner with respect to user-mode processes.
  - Kernel requests memory for data structures of varying size
    - Fragmentation should be minimized to prevent wasting
  - Due to the direct interaction of certain hardware devices with physical memory, usually contiguous memory is required (in spite of paging for user-mode processes)
Strategies for Managing Free Memory that is Assigned to Kernel Processes

- Buddy system
- Slab allocation
Buddy System

- Power-of-2 allocator
- Internal fragmentation
- Coalescing
Slab Allocation

- Slab: One or more physically contiguous pages
- Cache: One or more slabs
- There is a single cache for each unique kernel data structure, e.g., PCBs, file objects, semaphores, …
  - E.g., in Linux, a slab may be in one of the following three states:
    - Full: All objects in the slab are marked as used
    - Empty: All objects in the slab are marked as free
    - Partial: The slab consists of both used and free objects
Slab Allocation

- Benefits
  - No memory is wasted due to fragmentation
  - Memory requests can be satisfied quickly
- Slab allocator is used in Solaris 2.4 kernel as well as certain user-mode memory requests in Solaris
Other Considerations

- Prepaging
  - Prepaging $s$ pages
  - The fraction of $\alpha$ is used
  - The cost that have been saved with $\alpha \cdot s$ page faults should be greater than prepaging $(1-\alpha) \cdot s$ pages

- Page size
  - Page table size
  - Memory is better used with smaller pages
  - Time needed to read or write a page
  - Smaller page size improves locality and reduces I/O
  - The relationship between page size and sector size of the paging device
Other Considerations

- TLB Reach
  - TLB size * page size
  - Page size can affect the TLB reach
  - TLB in systems with more than one page size can be managed by software (Alpha, MIPS, UltraSPARC) or hardware (Pentium, PowerPC)

- Inverted page table
Other Considerations

- Program structure

```c
int i, j;
int[128][128] data;

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

```c
int i, j;
int[128][128] data;

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

- I/O interlock