Virtual Memory

- Describe the benefits of a virtual memory system
- Introduce demand paging, page replacement algorithms and frame allocation

Background

- 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.
  + Need to allow pages to be paged ("swapped") in and out
    - Performed by the pager
- Virtual memory can be implemented via:
  + Demand paging
  + Demand segmentation

Demand Paging

- Bring a page into memory only when it is needed
  + Less I/O needed
  + Less memory needed
  + Faster response
  + More users
- Page is needed ⇒ reference to it
  + Invalid reference ⇒ abort (trap to OS)
  + Not-in-memory ⇒ bring to memory

Lazy "swapping" / lazy paging

Justification of Demand Paging and Segmentation

Characteristics of Programs

- Programs have code used only in unusual situations, e.g., error management.
- Arrays, lists, and tables are allocated more memory than needed.

Benefits of having programs only partially in main memory:

- No constraints on the amount of physical memory available.
- Less memory required per program ⇒ more programs can execute concurrently ⇒ increase in throughput and CPU utilization (with no increase response time)
- Makes programming easier
- Benefits both the system and user

Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated
  (1 ⇒ in-memory, 0 ⇒ not-in-memory)
- Initially valid–invalid but is set to 0 on all entries.
- Example of a page table snapshot.

<table>
<thead>
<tr>
<th>Frame #</th>
<th>Valid-invalid bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- During address translation, if valid–invalid bit in page table entry is 0 ⇒ page fault

Agenda

- Background
- Demand Paging
- Performance of Demand Paging
- Page Replacement
- Allocation of Frames
- Thrashing
- Other Considerations
Page Fault

• If there is ever a reference to a page, first reference will trap to OS ⇒ page fault
• OS looks at another table to decide:
  + Invalid reference ⇒ abort.
  + Just not in memory.
• Get empty frame.
• Swap page into frame.
• Reset tables, validation bit = 1.
• Restart instruction: Least Recently Used
  + block move
  + auto increment/decrement location

Steps in Handling a Page Fault

What happens if there is no free frame?

• Terminate user process
  + Bad choice
• Swap out a process freeing all its frames
  + Decreases multiprogramming (feasible choice in certain conditions)
• Page replacement – find some page in memory, but not really in use, swap it out.
  + Algorithm
  + Performance – want an algorithm which will result in minimum number of page faults
• Same page may be brought into memory several times

Performance of Demand Paging

• Page Fault Rate 0 ≤ p ≤ 1.0
  + if p = 0, no page faults
  + if p = 1, every reference is a fault
• Effective Access Time (EAT)

\[ EAT = (1 - p) \times \text{memory access} + p \times (\text{swap page out} + \text{swap page in} + \text{restart overhead}) \]

Demand Paging Example

• Memory access time = 1 µs
  + 50% of the time the page that is being replaced has been modified and therefore needs to be swapped out.
• Swap Page Time = 10 ms

\[ EAT = (1 - p) \times 1 + p (10000 + 0.5 \times 10000) = 1 + 14999p \mu s \]

The Steps...

1. Trap to operating system
2. Save user registers and process state
3. Determine that interrupt was a page fault
4. Check that the page reference was legal, and determine the location of the page on disk
5. Initiate a disk read request
   a. Wait until read request is serviced
   b. Wait for the device seek and latency time
   c. Begin transfer of the page to free frame
6. While waiting, allocate CPU to other some other process
The Steps ... (Cont.)

7. Interrupt for the disk (I/O completion)
8. Save registers and process state for the new process (if step 6 was executed)
9. Determine interrupt was from disk
10. Update page table (and other internal structures)
11. Wait for CPU to be allocated to the process (again)
12. Restore user registers, process state, new page table. Then resume the interrupted transaction

The Simple View...

1. Service the page fault interrupt
2. Read in the page
3. Restart the process

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

General Page Fault Service Routine

1. Find the location of the desired page on the disk
2. Find a free frame
   a) If there is a free frame, use it
   b) Otherwise, use a page replacement algorithm to select a victim frame
   c) Write the victim page to disk; change page and frame tables
3. Read the desired page into (newly) free frame; change page and frame tables
4. Restart user process

Graph of Page Faults Versus The Number of Frames
First-In-First-Out (FIFO) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)
  - 1 1 4 5
  - 2 2 1 3
  - 3 3 2 4
- 9 page faults
- 4 frames
  - 1 1 5 4
  - 2 2 1 5
  - 3 3 2
  - 4 4 3
- More frames => less page faults

FIFO Replacement → Belady’s Anomaly

Optimal Algorithm

- Replace page that will not be used for longest period of time.
- 4 frames example
  - 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 6 page faults

Least Recently Used (LRU) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 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 determine which are to change.

LRU Algorithm (Cont.)

- 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
  - No search for replacement
- Both implementations require hardware support!
- LRU is a stack algorithm
  - An algorithm for which it can be shown that the set of pages in memory for n frames is always a subset of the set of pages that can be in memory with n+1 pages!
  - Cannot suffer from Belady’s anomaly.

LRU Approximation Algorithms

- Reference bit
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1.
  - Replace the one which is 0 (if one exists).
  - We do not know the order, however.
- Additional reference bits
  - E.g., 8 bits represent a history of references
  - The byte is shifted regularly
- Second chance
  - Need reference bit
  - Clock replacement
    - If page to be replaced (in clock order) has reference bit = 1, then:
      - Set reference bit 0
      - Leave page in memory
      - Replace next page (in clock order), subject to same rules

Second-Chance Page-Replacement Algorithm
LRU Approximation Algorithms (cont.)

- Enhanced second chance
  - Checks both reference bits and modify bits (ref, mod)
    - (0,0) = best page to replace (not used recently, not modified)
    - (0,1) = not recently used, but modified
    - (1,0) = recently used, but not modified
    - (1,1) = recently used, and modified (poor choice)

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.

Allocation of Frames

- Each process needs minimum number of pages.
- 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
- Two major allocation schemes.
  - fixed allocation
  - priority allocation

Fixed Allocation

- Equal allocation – e.g., if 100 frames (m) and 5 processes (n), give each process 20 pages (m/n).
- Proportional allocation – Allocate according to the size of process.

\[
\begin{align*}
  s_i & = \text{size of process } p_i \\
  S & = \sum s_i \\
  m & = \text{total number of frames} \\
  a_i & = \text{allocation for } p_i = \frac{s_i \times m}{S} \\
  a_1 & = \frac{10}{137} \times 64 = 5 \\
  a_2 & = \frac{127}{137} \times 64 = 59
\end{align*}
\]

Priority Allocation

- Use a proportional allocation scheme 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.

Global vs. Local Allocation

- Global replacement – process selects a replacement frame from the set of all frames; one process can take a frame from another.
- Local replacement – each process selects from only its own set of allocated frames.
Thrashing

- If a process does not have "enough" frames, the page-fault rate is very high. This leads to:
  + Low CPU utilization
  + Operating system thinks that it needs to increase the degree of multiprogramming.
  + Another process added to the system

- Thrashing • a process is busy swapping pages in and out.

Thrashing Diagram

- Why does 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

Working-Set Model

- ∆ ≡ working-set window ≡ a fixed number of page references
- Example: 10 000 instructions
- WSS (working set size of Process Pi) = total number of pages referenced in the most recent ∆ (varies in time)
  + if ∆ too small will not encompass entire locality.
  + if ∆ too large will encompass several localities.
  + if ∆ = ∞ will encompass entire program.
- D = Σ WSS approximately total demand frames
- if D > m ⇒ Thrashing
- Policy: if D > m, then suspend one of the processes

Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: ∆ = 10 000
  + Timer interrupts after every 5 000 time units.
  + Keep in memory 2 "history" pages for each page of the working set.
  + If one of the bits in memory = 1 ⇒ page in working set.
- Con: we don’t know where within the 5 000 time units, they were used.
- Improvement = 10 "history" bits and interrupt every 1 000 time units.

Page-Fault Frequency Scheme

- Establish "acceptable" page-fault rate.
  + If actual rate too low, process gains frame.
  + If actual rate too high, process loses frame.

Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by mapping a disk block to a page in memory
  + A file is initially read using demand paging. A page-sized portion of the file is read from the file system into a physical page.
  + Subsequent reads/writes to/from the file are treated as ordinary memory accesses.
  + Simplifies file access by treating file I/O through memory rather than read(), write() system calls
  + Also allows several processes to map the same file allowing the pages in memory to be shared
Memory Mapped Files

Other Issues – Prepaging

- Prepaging
  - To reduce the large number of page faults that occurs at process startup
  - Prepage all or some of the pages a process will need, before they are referenced
  - But if prepaged pages are unused, I/O and memory was wasted
  - Assume \( s \) pages are prepaged and \( \alpha \) of the pages is used
    - Is cost of \( s \times \alpha \) save pages faults > or < than the cost of prepaging \( s \times (1 - \alpha) \) unnecessary pages?
    - \( \alpha \) near zero \( \Rightarrow \) prepaging loses

Other Issues – Page Size

- Page size selection must take into consideration:
  - fragmentation
  - table size
  - I/O overhead
  - locality

Other Issues – TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
  - TLB Reach = (TLB Size) \times (Page Size)
  - Ideally, the working set of each process is stored in the TLB. Otherwise there is a high degree of page faults.
  - 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.

Other Issues – Program Structure

- Program structure
  - Page size 128 words
  - Each row is stored in one page

```java
int A[][] = new int[256][128];
for(int j = 0; j < A[0].length; j++)
  for(int i = 0; i < A.length; i++)
    A[i][j] = 0;
```

- 1/O interlock and addressing

- 256x128 = 32 768 page faults
- 256 page faults

Other Issues – 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.
Summary

- Logical address space > Physical address space
- Demand paging (pure demand paging)
- Page replacement algorithms
  - Belady's anomaly
  - Frame allocation
    - Locality
  - Thrashing

Recommended Reading and Exercises

- Reading:
  - [SGG7] Chapter 9 (9.10 optional)
  - Chapter 10 10.1-10.6, 10.8 (sixth edition)
- Exercises:
  - All