Frequently asked questions from the previous class survey

- Contents of page table entries in multilevel page table?
Topics covered in this lecture

- Demand Paging
- Performance of Demand Paging
- Page Replacement
- Belady’s anomaly
- Stack algorithms
Demand Paging: Basic concepts

- When a process is to be swapped in, **guess** which pages will be utilized by process
  - Before the process will be swapped out again

- **Avoid** reading unused pages
  - Better physical memory utilization
  - Reduced I/O
    - Lower swap times

Distinguishing between pages in memory and those on disk

- **Valid-Invalid** bits
  - Associated with entries in the page table

- **Valid**
  - Page is both legal and in memory

- **Invalid**
  1. Page is *not in logical address space* of process
  2. Valid BUT currently *on disk*
Distinguishing between pages in memory and those on disk

Handling Valid-invalid entries in the page table

- If process never attempts to access an invalid page?
  - No problems

- If process accesses page that is not memory resident?
  - Page fault
Handling page faults

1. Reference
2. Trap to the OS
3. Locate page on backing store
4. Bring in missing page
5. Reset page table
6. Restart instruction

Pure demand paging

- Never bring a page into memory unless it is required
- Execute process with no pages in memory
  - First instruction of process will fault for the page
- Page fault to load page into memory and execute
Potential problems with pure demand paging

- Multiple page faults per instruction execution
  - One fault for instruction
  - Many faults for data

- Multiple page faults per instruction are rare
  - Locality of reference

Hardware requirements to support demand paging

- Page Table
  
- Secondary memory
  - Section of disk known as swap space is used
Restarting instructions after a page fault

- Page faults occur at memory reference
- Use PCB to save state of the interrupted process
- Restart process in exactly the same place
  - Desired page is now in memory and accessible

Restarting processes after a page fault has been serviced

- If fault occurred during an instruction fetch
  - During restart, refetch the instruction
- If fault occurred while fetching operands
  - 1. Fetch and decode instruction
  - 2. Fetch the operand
Worst case example

- Add operands A and B
  - Place sum in C

- If we fault while storing C
  - Service page fault
  - Update page table
  - Restart instruction
    - Decode, fetch operand and perform addition

Problems when operations modify several different memory locations

- E.g. Move a block from one memory location to another
- {C1} Either block straddles page-boundary
- {C2} Page fault occurs

- Move might be partially done
  - Uh-oh …
Approaches to fault-proofing block transfers

1. Compute and access both ends of the block
   - If a page fault were to happen: it will at this point
     - Nothing has been partially modified
   - After fault servicing, block transfer completes

2. Use temporary registers
   - Track overwritten values

Can on-demand paging be applied anywhere without modifications?

- Paging is between CPU and physical memory
  - Transparent to user process

- Non-demand paging can be applied to any system

- Not so for demand paging
  - Fault processing of special instructions non-trivial
Effective access times

- **Without** page faults, effective access times are equal to memory access times
  - 200 nanoseconds approximately

- **With** page faults
  - Account for fault servicing with disk I/O
Calculating the effective access times with demand paging

\[ p \]: probability of a page fault
\[ ma \]: memory access time

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

Components of page-fault servicing

- Service interrupt
  - 1~100 \( \mu \)S
- Read in the page
  - Latency: 3 mS
  - Seek: 5 mS
- Restart process
  - 1~100 \( \mu \)S
Effective access times

- Effective access time =
  \[(1-p) \times ma + p \times \text{page-fault-time}\]
  
  \[(1-p) \times 200ns + p \times (8mS)\]
  
  \[(1-p) \times 200 + p \times (8,000,000)\]
  
  \[= 200 + 7,999,800 \times p\]

Effective access time directly proportional to page-fault rate

If performance degradation is to be less than 10%

\[220 > 200 + 7,999,800 \times p\]

\[20 > 7,999,800 \times p\]

\[p < 0.0000025\]

Fewer than 1 memory access out of 399,990 can page-fault
OTHER ISSUES IN DEMAND PAGING

Allocation of physical memory to I/O and programs is a challenge

- Memory used for holding program pages
- I/O buffers also consume a big chunk of memory
- Solutions:
  - Fixed percentage set aside for I/O buffers
  - Processes and the I/O subsystem compete
Demand paging and the limits of logical memory

- **Without demand paging**
  - All pages of process **must be** in physical memory
  - Logical memory **limited** to size of physical memory

- **With demand paging**
  - All pages of process **need not be** in physical memory
  - Size of logical address space is **no longer constrained** by physical memory

Demand paging is the OS’ attempt to improve CPU utilization and system throughput

- **Load pages into memory when they are referenced**
  - Increases degree of **multiprogramming**

- **Example**
  - 40 pages of physical memory
  - 6 processes each of which is 10 pages in size
    - Each process only needs 5 pages as of now
  - Run 6 processes with 10 pages to spare
Increasing the degree of multiprogramming can be tricky

- Essentially we are **over-allocating** physical memory
- Example
  - Physical memory = 40 pages
  - 6 processes each of which is of size 10 pages
    - But are using 5 pages each as of now
  - What happens if each process needs all 10 pages?
    - 60 physical frames needed

Coping with over-allocation of memory

- **Terminate** a user process
  - But paging should be transparent to the user

- **Swap out** a process
  - Reduces the degree of multiprogramming

- **Page replacement**
The two core problems in demand paging

- **Frame allocation**
  - How many frames to allocate to a process

- **Page replacement**
  - Select the frame(s) for replacement

- **Caveat:**
  - Disk I/O is expensive so inefficient solutions can weigh things down
Page replacement

- If no frame is free?
  - Find one that is not currently being used
    - Use it

Freeing a physical memory frame

- Write frame contents to swap space
- Change page table of process
  - To reflect that page is no longer in memory
- Freed frame can now hold some other page
Servicing a page fault

1. Reference
2. Trap to the OS
3. Locate page on backing store
4. Bring in missing page
5. Reset page table
6. Restart instruction

Page replacement is central to demand paging

1. Load M
2. Trap to the OS
3. Locate page on backing store
4. Bring in missing page
5. Reset page table
6. Restart instruction
Overheads for page replacement

- If no frames are free: 2 page transfers needed
  - Victim page out
  - New page in

- No free frames?
  - Doubles page-fault service time
  - Increases effective access time

Using the modify bit to reduce page replacement overheads

- Each page/frame has a **modify** bit
  - Set by hardware when page is written into
  - Indicates if page was modified
    - Since the last time it was read from disk

- During page replacement
  - If victim page not modified, no need to write it to disk
    - Reduces I/O time by **one-half**
Page replacement algorithms:

- What are we looking for?
  - Low page-fault rates

- How do we evaluate them?
  - Run algorithm on a string of memory references
    - Reference string
  - Compute number of page faults
The reference string:
Snapshot memory references

- We track page numbers
  - Not the entire address

- If we have a reference to a memory-resident page $p$
  - Any references to $p$ that follow will not page fault
    - Page is already in memory

---

The reference string: Example
Page size = 100 bytes

```
0100 0432 0101 0612 0102 0103 0101 0611 0102 0103
0104 0101 0610 0102 0103 0104 0101 0609 0102 0105
```

1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1
Factors involved in determining page faults

- Reference string of executing process
- Page replacement algorithm
- Number of physical memory frames available

Intuitively:
- Page faults reduce as the number of page frames increase

FIFO (First In First Out) Page Replacement Algorithm
FIFO page replacement algorithm: Out with the old; in with the new

When a page must be replaced
- Replace the **oldest** one

OS maintains list of all pages currently in memory
- Page at head of the list: Oldest one
- Page at the tail: Recent arrival

During a page fault
- Page at the head is removed
- New page added to the tail

FIFO example: 3 memory frames

Reference String

| 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |

Youngest

| 7 | 0 | 1 | 2 | 2 | 3 | 0 | 4 | 2 | 3 | 0 | 0 | 1 | 2 | 2 | 2 | 7 | 0 | 1 |

| 7 | 0 | 1 | 1 | 2 | 3 | 0 | 4 | 2 | 3 | 3 | 3 | 0 | 1 | 1 | 1 | 2 | 7 | 0 |

Oldest

| 7 | 0 | 0 | 1 | 2 | 3 | 0 | 4 | 2 | 2 | 2 | 3 | 0 | 0 | 0 | 1 | 2 | 7 | |

No page fault
Intuitively the greater the number of memory frames, the lower the faults

- Surprisingly this is **not always** the case

- In 1969 Belady, Nelson and Shedler discovered counter example* in FIFO
  - FIFO caused more faults with 4 frames than 3

- This strange situation is now called **Belady’s anomaly**

An anomaly in space-time characteristics of certain programs running in a paging machine.
Belady’s anomaly: FIFO
Same reference string, different frames

<table>
<thead>
<tr>
<th>Youngest</th>
<th>Oldest</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 0 1 4 4 4 2 3 3</td>
<td>0 1 2 3 0 0 1 4 4</td>
</tr>
<tr>
<td>0 1 2 3 0 1 1 1 4 2 2</td>
<td></td>
</tr>
<tr>
<td>Numbers in this color: No page fault</td>
<td></td>
</tr>
<tr>
<td>9 page faults with 3 frames</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Youngest</th>
<th>Oldest</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 0 1 4 0 1 2 3 4</td>
<td>0 0 0 1 2 3 4 0 1</td>
</tr>
<tr>
<td>0 1 2 3 3 3 4 0 1 2 3</td>
<td></td>
</tr>
<tr>
<td>0 1 1 1 3 3 4 0 1 2</td>
<td></td>
</tr>
<tr>
<td>10 page faults with 4 frames</td>
<td></td>
</tr>
</tbody>
</table>

Belady’s anomaly

- Led to a whole theory on paging algorithms and properties
- Stack algorithms
The Model

- There is an array $M$
  - Keeps track of the state of memory
- $M$ has as many elements as pages of virtual memory
- Divided into two parts
  - Top part: $m$ entries \{Pages currently in memory\}
  - Bottom part: $n-m$ entries
    - Pages that were referenced BUT paged out
The model

Properties of the model

- When a page is referenced
  - Move to the top entry of M

- If the referenced page is already in M
  - All pages above it moved down one position
  - Pages below it are not moved

- Transition from within box to outside of it?
  - Page eviction from main memory
The model

Properties of the model

- $M(m,r)$
  - The set of pages in the top part of $M$
  - $m$ page frames
  - $r$ memory references
A property that has some interesting implications

- $M(m, r)$ subset of $M(m+1, r)$

- Set of pages in the top part of $M$ with $m$ frames
  - Also included in $M$ with $(m+1)$ frames

What the subset relationship means

- Execute a process with a set of memory frames
- If we increase memory size by one frame and re-execute
  - At every point of execution all pages in the first execution are present in the second run
- Does not suffer from Belady’s anomaly
  - Stack algorithms
The contents of this slide-set are based on the following references