Outline

- Objective for today:
  - Replacement policies

Policies for Paged Virtual Memory

The OS tries to minimize page fault costs incurred by all processes, balancing fairness, system throughput, etc.

1. fetch policy: When are pages brought into memory?
   - prepage: reduce page faults by bring pages in before needed
   - on demand: in direct response to a page fault

2. replacement policy: How and when does the system select victim pages to be evicted/discard from memory?

3. placement policy: Where are incoming pages placed?
   - Which frame?

4. backing storage policy:
   - Where does the system store evicted pages?
   - When is the backing storage allocated?
   - When does the system write modified pages to backing store?
   - Clustering: reduce seeks on backing storage

Page Replacement

- When there are no free frames available, the OS must replace a page (victim), removing it from memory to reside only on disk (backing store), writing the contents back if they have been modified since fetched (dirty).
- Replacement algorithm - goal to choose the best victim, with the metric for "best" (usually) being to reduce the fault rate.
Terminology

- Each thread/process/job utters a stream of page references.
  - Model execution as a page reference string: e.g., "abcabcabcabce...
- The OS tries to minimize the number of faults incurred.
  - The set of pages (the working set) actively used by each job changes relatively slowly.
  - Try to arrange for the resident set of pages for each active job to closely approximate its working set.
- Replacement policy is the key.
  - Determines the resident subset of pages.

Page Reference String

<table>
<thead>
<tr>
<th>Addr trace</th>
<th>Page str</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>1 0 1</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1 0 1</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1 0 1</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1 0 1</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1 0 1</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1 0 1</td>
</tr>
</tbody>
</table>

Example (Artificially Small Pagesize)

```c
#define dim
int A[dim][dim], B[dim][dim];
int main()
{ int i, j, k;
  for (i=0; i<dim; i++)
    for (j=0; j<dim; j++)
      B[i][j] = A[i][j];
  exit;
}
```
Example Page Reference String

...0 { 1 2 { 2 3 4 5 15 5 20 5 6}m 2 6
( 2 3 4 5 16 5 21 5 6)"m 2 6
( 2 3 4 5 17 5 22 5 6)"m 2 6
...
( 2 3 4 5 19 5 24 5 6)"m 2 6 7 )"n

Assessing Replacement Algs

✓ Model program execution as a reference string
✓ Metric of algorithm performance is fault rate
  • Comparison to base line of **Optimal Algorithm.**
  • For a specific algorithm: What is the information needed? How is that information gathered? When is it acted upon?
    • At each memory reference
    • At fault time
    • At periodic intervals

Managing the VM Page Cache

1. Pages are typically referenced by page table entries. Must invalidate before reusing the frame.
2. Reads and writes are implicit; the TLB hides them from the OS.
   How can we tell if a page is dirty?
   How can we tell if a page is referenced?
3. Page cache manager must run policies periodically, sampling page state.
   Continuously push dirty pages to disk to "launder" them.
   Continuously check references to judge how "hot" each page is.
   Balance accuracy with sampling overhead.
Replacement Algorithms
Assume fixed number of frames in memory assigned to this process:
• Optimal - baseline for comparison - future references known in advance - replace page used furthest in future.
• FIFO
• Least Recently Used (LRU) stack algorithm - don’t do worse with more memory.
• LRU approximations for implementation
  Clock, Aging register

Example: Optimal

FIFO
• No extra hardware assistance needed,
  No per-reference overhead (we have no information about actual access pattern)
• At fault time: maintain a first-in first-out queue of pages resident in physical memory
• Replace oldest resident page
• Why it might make sense - straight-line code, sequential scans of data
• Belady’s anomaly - fault rate can increase with more memory
**LRU**

- At fault time: replace the resident page that was last used the longest time ago
- Idea is to track the program’s temporal locality
- To implement exactly: we need to order the pages by time of most recent reference (per-reference information needed $\rightarrow$ HW, too $\mathcal{S}$)
  - timestamp pages at each ref, stack operations at each ref
- Stack algorithm - doesn’t suffer from Belady’s anomaly -- if $i > j$ then set of pages with $j$ frames is a subset of set of pages with $i$ frames.
LRU Approximations for Paging

- Pure LRU and LFU are prohibitively expensive to implement.
  - most references are hidden by the TLB
  - OS typically sees less than 10% of all references
  - can’t tweak your ordered page list on every reference
- Most systems rely on an approximation to LRU for paging.
  - Implementable at reasonable cost
  - periodically sample the reference bit on each page
    - visit page and set reference bit to zero
    - run the process for a while (the reference window)
    - come back and check the bit again
  - reorder the list of eviction candidates based on sampling

Approximations to LRU

- HW support: usage/reference bit in every PTE - set on each reference.
  - Set in TLB entry
  - Written back to page table (in memory) on TLB replacement
- We now know whether or not a page has been used since the last time the bits were cleared.
  - Algorithms differ on when & how that’s done.
  - Unless it’s on every reference, there will be “ties”

A Page Table Entry (PTE)

- Valid bit: OS sets this to tell MMU that the translation is valid.
- Write-enable bit: OS can enable or disable write access for this mapping.
- Dirty bit: set when a store is completed to the page (page is modified).
- Reference bit: set when a reference is made through the mapping.
**Clock Algorithm**

- Maintain a circular queue with a pointer to the next candidate (clock hand).
- At fault time: scan around the clock, looking for page with usage bit of zero (that's your victim), clearing usage bits as they are passed.
- We now know whether or not a page has been used since the last time the bits were cleared.

**Example Reference String**

\[ \downarrow \downarrow \downarrow \downarrow \downarrow \]
\[ \ldots 0 \{ 1 \ 2 \{ 2 \ 3 \ 4 \ 5 \ 15 \ 5 \ 20 \ 5 \ 6 \} \} \downarrow \downarrow \downarrow \downarrow \downarrow \]
\[ \{ 2 \ 3 \ 4 \ 5 \ 16 \ 5 \ 21 \ 5 \ 6 \} \downarrow \downarrow \downarrow \downarrow \downarrow \]
\[ \{ 2 \ 3 \ 4 \ 5 \ 17 \ 5 \ 22 \ 5 \ 6 \} \downarrow \downarrow \downarrow \downarrow \downarrow \]

**Approximating a Timestamp**

- Maintain a supplemental data structure (a counter) for each page.
- Periodically (on a regular timer interrupt) gather info from the usage bits and zero them.
  ```c
  for each page i {
    if (used[i]) counter[i] = 0;
    else counter[i]++;
    used[i] = 0;
  }
  ```
- At fault time, replace page with largest counter value (time intervals since last use).
Practical Considerations

• Dirty bit - modified pages require a writeback to secondary storage before frame is free to use again.
• Variation on Clock tries to maintain a healthy pool of clean, free frames
  – on timer interrupt, scan for unused pages, move to free pool, initiate writeback on dirty pages
  – at fault time, if page is still in frame in pool, reclaim; else take free, clean frame.

A Page Table Entry (PTE)

PFN

Valid bit: OS sets this to tell MMU that the translation is valid.
Write-enable: OS touches this to enable or disable write access for this mapping.
Dirty bit: set when a store is completed to the page (page is modified).
Reference bit: set when a reference is made through the mapping.

The Paging Daemon

• Most OS have one or more system processes responsible for implementing the VM page cache replacement policy.
  – A daemon is an autonomous system process that periodically performs some housekeeping task.
• The paging daemon prepares for page eviction before the need arises.
  – Wake up when free memory becomes low.
  – Clean dirty pages by pushing to backing store.
    • prewrite or pageout
  – Maintain ordered lists of eviction candidates.
  – Decide how much memory to allocate to file cache, VM, etc.
MMU Games

Vanilla Demand Paging
- Valid bit in PTE means non-resident page. Resulting page fault causes OS to initiate page transfer from disk.
- Protection bits in PTE means page should not be accessed in that mode (usually means non-writable)

What else can you do with them?

A Page Table Entry (PTE)

- Valid bit: OS sets this to tell MMU that the translation is valid.
- Write-enable: OS touches this to enable or disable write access for this mapping.
- Reference bit: set when a reference is made through the mapping.
- Dirty bit: set when a store is completed to the page (page is modified).

Simulating Usage Bits

- Turn off both valid bit and write-protect bit
- On first reference - fault allows recording the reference bit information by OS in an auxiliary data structure. Set it valid for subsequent accesses to go through HW.
- On first write attempt - protection fault allows recording the dirty bit information by OS in aux. data structure.
Copy-on-Write

- Operating systems spend a lot of their time copying data.
  - particularly Unix operating systems, e.g., `fork()`
  - cross-address space copies are common and expensive
- Idea: defer big copy operations as long as possible, and hope they can be avoided completely.
  - create a new shadow object backed by an existing object
  - shared pages are mapped read-only in participating spaces
  - read faults are satisfied from the original object (typically)
  - write faults trap to the kernel
    - make a (real) copy of the faulted page
    - install it in the shadow object with writes enabled

Variable / Global Algorithms

- Not requiring each process to live within a fixed number of frames, replacing only its own pages.
- Can apply previously mentioned algorithms globally to victimize any process’s pages
- Algorithms that make number of frames explicit.

Variable Space Algorithms

- Working Set
  Tries to capture what the set of active pages currently is.
  The whole working set should be resident in memory for the process to bother running. WS is set of pages referenced during window w of time (now - w, now).
  - Working Set Clock - a hybrid approximation
- Page Fault Frequency
  Monitor fault rate, if pff > high threshold, grow # frames allocated to this process, if pff < low threshold, reduce # frames.
  Idea is to determine the right amount of memory to allocate.
**Working Set Model**

- Working set at time $t$ is the set of pages referenced in the interval of time $(t-w, t)$ where $w$ is the working set window.
  - Implies per-reference information captured.
  - How to choose $w$?
- Identifies the "active" pages. Any page that is resident in memory but not in any process's working set is a candidate for replacement.
- Size of the working set can vary with locality changes.

---

**Example Reference String**

```
A B i j ...
{ 1 2 3 4 5 15; 5 20 5 6} m 2 6
{ 2 3 4 5 16; 5 21 5 6} m 2 6
{ 2 3 4 5 17; 5 22 5 6} m 2 6
...
{ 2 3 4 5 19; 5 24 5 6} m 2 6 7}
```

---

**Clock Algorithm**

- Maintain a circular queue with a pointer to the next candidate (clock hand).
- At fault time: scan around the clock, looking for page with usage bit of zero (that's your victim), clearing usage bits as they are passed.
- We now know whether or not a page has been used since the last time the bits were cleared.
**Working Set Model**

- Working set at time t is the set of pages referenced in the interval of time \((t-w, t)\) where \(w\) is the working set **window**.
  - Implies per-reference information captured.
  - How to choose \(w\)?
- Identifies the "active" pages. Any page that is resident in memory but not in any process's working set is a candidate for replacement.
- Size of the working set can vary with locality changes

**WSClock**

- The implementable approximation
- At fault time: scan usage bits of resident pages. For each page \(i\):
  
  \[
  \begin{align*}
  \text{if (used)} & \text{ (time of ref} = v_{\text{owner}[i]} / \text{virtual time of owning process} / \text{used} = 0) \\
  \text{else if } \text{(} | v_{\text{owner}[i]} - \text{time of ref} | >= w \text{)} & \text{ replaceable; else still in "working set"}
  \end{align*}
  \]

<table>
<thead>
<tr>
<th>(\text{vt} )</th>
<th>(\text{rt} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
</tr>
</tbody>
</table>

**WSClock**

```
{a,b,c}  {x,y}  {a,f}  {y,z}  W=2
```

```
rt=0  rt=3  rt=6  rt=10  rt=13
```
Thrashing

- Page faulting is dominating performance
- Causes:
  - Memory is overcommitted - not enough to hold locality sets of all processes at this level of multiprogramming
  - Lousy locality of their programs
  - Positive feedback loops reacting to paging I/O rates

- Load control is important (how many slices in frame pie?)
  - Admission control strategy (limit # processes in load phase)

Pros and Cons of VM

- Demand paging gives the OS flexibility to manage memory...
  - programs may run with pages missing
    - unused or "cold" pages do not consume real memory
    - improves degree of multiprogramming
    - program size is not limited by physical memory
    - program size may grow (e.g., stack and heap)
- ...but VM takes control away from the application.
  - With traditional interfaces, the application cannot tell how much memory it has or how much a given reference costs.
  - Fetching pages on demand may force the application to incur I/O stalls for many of its references.

Motivation:
How to exploit remote memory?

- Low latency networks
  - make retrieving data across the network faster than local disk
  - VM and file caching

- "IO bottleneck"
Distributed Shared Memory (DSM)

Allows use of a shared memory programming model (shared address space) in a distributed system (processors with only local memory)

DSM Issues

- Can use the local memory management hardware to generate fault when desired page is not locally present or when write attempted on read-only copy.
- Locate the page remotely - current "owner" of page (last writer) or "home" for page.
- Page sent in message to requesting node (read access makes copy; write migrates)
- Consistency protocol - invalidations or broadcast of changes (update)
  - directory kept of "caches" holding copies

DSM States

Forced faults are key to consistency operations

- Invalid local mapping, attempted read access
  - data copied from most recent writer, set write-protect bit for all copies.
- Invalid local mapping, attempted write access
  - migrate data, invalidate all other copies.
- Local read-only copy, write-fault
  - invalidate all other copies
False Sharing

• Page bouncing sets in when there are too frequent coherency operations.
• One cause: false sharing
  Half used by one proc.
  Other half by another
• Subpage management?

Packaged & managed as one page