Outline for Today's Lecture

- Administrative:
  - Demo 4 signups?
  - Next week plans – guest lectures by Amin Vahdat (Tues) and Darrell Anderson (Thurs)
- Objective for today:
  - Placement policy – where the research is now.
  - Replacement policy

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?
   - Prepaging: 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/discarded 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

Placement Policy

Which free frame to chose?
Are all frames in physical memory created equal?

- Yes, only considering size. Fixed size.
- No, if considering
  - Cache performance, conflict misses
  - Access to multi-bank memories
  - Multiprocessors with distributed memories

Power Management (RAMBUS memory)

Read or write

Again, the v.a. → p.a mapping could exploit page coloring ideas
Power Management (RAMBUS memory)

- Active: 300mW
- Powered down: +6ns
- Standby: 180mW

Read or write 60ns

Again, the v.a. -> p.a mapping could exploit page coloring ideas

Two Dimensions to Control Energy

- Hardware control
- Software control

Dual-state (Static) HW Power State Policies

- All chips in one base state
- Individual chip Active while pending requests
- Return to base power state if no pending access
Quad-state (Dynamic) HW Policies

- Downgrade state if no access for threshold time
- Independent transitions based on access pattern to each chip
- Competitive Analysis
  - rent-to-buy
  - Active to nap 100's of ns
  - Nap to PDN 10,000 ns

- Access
- STBY
- PDN
- Nap

Time

Page Allocation Polices

- Random Allocation
  - Pages spread across chips
- Sequential First-Touch Allocation
  - Consolidate pages into minimal number of chips
  - One shot
- Frequency-based Allocation
  - First-touch not always best
  - Allow movement after first-touch

Dual-state + Random Allocation*

- Active to perform access, return to base state
- Nap is best ~85% reduction in E*D over full power
- Little change in run-time, most gains in energy/power

Normalized Energy*Delay

Quad-state HW + Sequential Allocation*

- Base: Dual-state Nap Sequential Allocation
- Thresholds: 0ns A->S; 750ns S->N; 375,000 N->P
- Quad-state + Sequential 30% to 55% additional improvement over dual-state nap sequential
- Sophisticated HW not enough

Normalized Energy*Delay

*NT Traces

*SPEC
The Design Space

<table>
<thead>
<tr>
<th>Dual-state Hardware (static)</th>
<th>Sequential Allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nap is best dual-state policy</td>
<td>Additional 10% to 30% over Nap</td>
</tr>
<tr>
<td>Thresholds not obvious. Could equal to dual-state</td>
<td>Best Approach: 6% to 55% over dual-nap-seq, 99% to 80% over all active</td>
</tr>
</tbody>
</table>

Hot Spots

Problem of creating a “hot spot” at one of the node memories - essentially analogous to the cache conflict miss problem motivating the page coloring and bin hopping ideas. Reuse a good idea.

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?
   - prepaging: 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/discarded from memory?

3. **placement policy**: Where are incoming pages placed?

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.
**Backing Store = Disk**

- Modified (dirty) pages are pushed to backing store (swap) on eviction.
- Page-out pages are fetched from backing store when needed.
- Initial references to user stack and BSS are satisfied by zero-fill on demand.

**Rotational Media**

- Access time = seek time + rotational delay + transfer time
  - **Seek time** = 5-15 milliseconds to move the disk arm and settle on a cylinder
  - **Rotational delay** = 8 milliseconds for full rotation at 7200 RPM; average delay = 4 ms
  - **Transfer time** = 1 millisecond for an 8KB block at 8 MB/s

- Bandwidth utilization is less than 50% for any noncontiguous access at a block grain.
- Layout issues: clustering

**A Case for Large Pages**

- Page table size is inversely proportional to the page size
- Memory saved
- Transferring larger pages to or from secondary storage (possibly over a network) is more efficient
- Number of TLB entries are restricted by clock cycle time,
  - Larger page size maps more memory
  - Reduces TLB misses

**A Case for Small Pages**

- Fragmentation
  - Not that much spatial locality
  - Large pages can waste storage
  - Data must be contiguous within page
**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?
   - prepaging: 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/dumped 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

---

**Paged Virtual Memory**

- General notion of a cache:
  - copies of data temporarily moved into storage of faster, higher cost, lower capacity technology
  - Achieving a high hit ratio gives performance equivalent to having most of memory built using this $\$$ technology
- The paging system manages physical memory as a cache over a larger virtual address space.
  - Data “lives” on disk, and a copy is in physical memory only while in active use.
  - Hardware and OS software cooperate to maintain the illusion that the machine’s memory “looks like” the virtual memory.
  - The OS controls data placement/movement and establishes the set of translations in effect at any time.
  - The VM abstraction is (mostly) transparent to user code.

---

**The Page Caching Problem** (aka Replacement Policy)

- Each thread/process/job utters a stream of page references.
  - Model execution as a page reference string e.g., “abcabcabcabc”.
- 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.

---

**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**

```c
sw $0,16($ fp)
$L10:
lw $2,16($ fp)
bl $3,$5,;$ L13
sw $0,20($ fp)
$L11:
```
Assessing Replacement Algs

- Model program execution as a reference string
- Metric of algorithm performance is fault rate
- Comparison to baseline 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

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 Reference String

\[
\ldots 0 \{ 1 \ 2 \ \{ 2 \ 3 \ 4 \ 5 \ 15 \ 5 \ 20 \ 5 \ 6 \}^m 2 \ 6
\{ 2 \ 3 \ 4 \ 5 \ 16 \ 5 \ 21 \ 5 \ 6 \}^m 2 \ 6
\{ 2 \ 3 \ 4 \ 5 \ 17 \ 5 \ 22 \ 5 \ 6 \}^m 2 \ 6
\ldots
\{ 2 \ 3 \ 4 \ 5 \ 19 \ 5 \ 24 \ 5 \ 6 \}^n 2 \ 6 \ 7 \}
\]

Example Reference String

\[
\ldots 0 \{ 1 \ 2 \ \{ 2 \ 3 \ 4 \ 5 \ 15 \ 5 \ 20 \ 5 \ 6 \}^m 2 \ 6
\{ 2 \ 3 \ 4 \ 5 \ 16 \ 5 \ 21 \ 5 \ 6 \}^m 2 \ 6
\{ 2 \ 3 \ 4 \ 5 \ 17 \ 5 \ 22 \ 5 \ 6 \}^m 2 \ 6
\ldots
\{ 2 \ 3 \ 4 \ 5 \ 19 \ 5 \ 24 \ 5 \ 6 \}^n 2 \ 6 \ 7 \}
\]
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

Example 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

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 -> HW, too $\mathcal{O}$)
  - 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.

Example 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

15 20 6
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.
  - 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

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

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.
  for each page i {if (used \(i\)) \(\text{counter}_i = 0\); else \(\text{counter}_i++\); \(\text{used}_i = 0\);}
- 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.
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.
  - Maintain ordered lists of eviction candidates.
  - Decide how much memory to allocate to UBC, VM, etc.

FIFO with Second Chance (Mach)

- **Idea**: do simple FIFO replacement, but give pages a “second chance” to prove their value before they are replaced.
  - Every frame is on one of three FIFO lists:
    - active, inactive and free
  - Page fault handler installs new pages on tail of active list.
  - “Old” pages are moved to the tail of the inactive list.
  - Paging daemon moves pages from head of active list to tail of inactive list when demands for free frames is high.
  - Clear the refbit and protect the inactive page to “monitor” it.
- Pages on the inactive list get a “second chance”.
  - If referenced while inactive, reactivate to the tail of active list.

Illustrating FIFO-2C

- Restock inactive list by pulling pages from the head of the active list; knock off the reference bit and inactivate.
- Inactive list scan:
  1. Page on inactive list has been referenced? Remove from tail of active list (inactivation).
  2. Page at head of inactive list has not been referenced? Page_protect and place on tail of free list.
  3. Dirty page on inactive list? Push to disk and return to inactive list tail.
- Consume frames from the head of the free list.
- If free shrinks below threshold, kick the paging daemon to start a scan.

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 of time (now-t, 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

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>15</th>
<th>20</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ 2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>16</td>
<td>5</td>
<td>21</td>
<td>5</td>
<td>6}</td>
</tr>
<tr>
<td>{ 2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>17</td>
<td>5</td>
<td>22</td>
<td>5</td>
<td>6}</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{ 2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>19</td>
<td>5</td>
<td>24</td>
<td>5</td>
<td>6}</td>
</tr>
</tbody>
</table>

WSClock

• The implementable approximation
  • At fault time: scan usage bits of resident pages. For each page i
    
    ```
    if (used_i) {time_of_ref_i = v_towner[i] /*virtual time of owning process*/; used_i = 0;}
    else if ( | v_towner[i] – time_of_ref_i | >= w ) replaceable; //else still in “working set"
    ```
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

Let \( w = 5 \)

Then this is no longer in working set of \( P_0 \)

Another problem: Sharing

- The indirection of the mapping mechanism in paging makes it tempting to consider sharing code or data - having page tables of two processes map to the same page, but...
- Interaction with caches, especially if virtually addressed cache.
- What if there are addresses embedded inside the page to be shared?
Paging and Sharing (difficulties with embedded addresses)

- Virtual address spaces still look contiguous.
- Virtual Address Space for Process 0 links foo into pink address region
- Virtual Address Space for Process 1 links bar into blue region
- Then along comes Process 2 ...

VAS\(_0\) wants to share both foo and bar.

Segmentation

- A better basis for sharing. Naming is by logical unit (rather than arbitrary fixed size unit) and then offset within unit (e.g. procedure).
- Segments are variable size
- Segment table is like a bunch of base/limit registers.

Sharing in Segmentation

- Naming is by logical objects.
- Offsets are relative to base of object.
- Address spaces may be sparse as well as being non-contiguous.

Combining Segmentation and Paging

- Sharing supported by segmentation. Programs name shared segments.
- Physical storage management simplified by paging of each segment.