Lecture Topics

- Today: Virtual Memory
  (Stallings, chapter 8.1-8.4)
- Next: continued

Announcements

- Self-Study Exercise #8
- Project #7 (due 11/1)
- Project #8 (due 11/15)
Exam #2

- Thursday, 11/8 during lecture
- 80 minutes, 18% of course grade
- Topics:
  - the memory hierarchy
  - cache memory
  - main memory
  - virtual memory
- Study suggestions on course website

OS Issues

- Resident set management
- Cleaning policy
- Load control
Resident Set Management

- Resident Set Size
  - Fixed allocation
  - Variable allocation

- Replacement Scope
  - Local replacement
  - Global replacement

Resident Set Size

Trade offs

- If fewer frames are allocated to one process, more processes can be resident in RAM; the pool of processes in the Ready state is larger.
- The page fault rate will be higher if too few frames are allocated to a process.
- Beyond a certain point, allocating more page frames will not lower the page fault rate for a process.
### Resident Set Management

<table>
<thead>
<tr>
<th></th>
<th>Local Replacement</th>
<th>Global Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fixed Allocation</strong></td>
<td>Number of frames allocated to a process is fixed.</td>
<td>Not possible.</td>
</tr>
<tr>
<td></td>
<td>Page to be replaced is chosen from among the frames</td>
<td></td>
</tr>
<tr>
<td></td>
<td>allocated to that process.</td>
<td></td>
</tr>
<tr>
<td><strong>Variable Allocation</strong></td>
<td>Number of frames allocated to a process may vary over time to maintain the working set of the process.</td>
<td>Page to be replaced is chosen from all available frames in main memory; this causes the size of the resident set of processes to vary.</td>
</tr>
<tr>
<td></td>
<td>Page to be replaced is chosen from among the frames</td>
<td></td>
</tr>
<tr>
<td></td>
<td>allocated to that process.</td>
<td></td>
</tr>
</tbody>
</table>

### Working Set Model

- **Resident set**: for a given process, the set of pages which are currently resident in RAM
- **Working set**: the set of pages which have been referenced recently (specific time delta usually given)
- Working set changes over time
The working set of a process grows when it first starts executing.

It then stabilizes (locality of reference).

It grows again when the process enters a new locality (transition period), eventually contains pages from two different localities.

It then decreases after sufficient time is spent in the new locality.
Working Set Strategy

Strategy:

1. Monitor working set of each process

2. Periodically remove pages from resident set if they are no longer in working set

3. Process can only be in Ready state if its working set is contained in its resident set

Working Set Strategy

Problems:

1. Size and membership of working set will change over time

2. Impractical to keep track of working set for each process (time stamp each page reference?)
Page Fault Frequency Algorithm

- Approximate working set strategy by focusing on the page fault rate of a process
- Page fault rate too high: allocate frame
- Page fault rate too low: deallocate frame
- Doesn't handle transitional periods very well, but variations on the algorithm can account for those spikes in the page fault rate

Cleaning Policy

- Pages which have been modified must be copied to disk (or changes will be lost)
- Demand cleaning: page is copied to disk when it has been chosen as the victim
- Precleaning: pages copied to disk periodically so that they can be copied in blocks
Cleaning Policy

- Problem with demand cleaning: process with page fault has to wait for two disk transfers (modified page from RAM to disk, required page from disk to RAM)

- Problem with precleaning: many pages are copied to disk and then modified again (same pages are copied to disk repeatedly)

Cleaning Policy

- Page buffering can improve performance

- Two free frame lists: unmodified and modified pages

- Unmodified pages do not need to be copied to disk

- Modified pages can be copied to disk, then moved to the other list
Load Control

- Multiprogramming level: number of processes which are resident in RAM

![Graph showing processor utilization vs. multiprogramming level]

Multiprogramming Level

- Initially, as the multiprogramming level increases, CPU utilization increases (there is less chance that all processes are blocked)

- However, beyond a certain point, the resident set for each process will become too small and there will be many page faults

- Suspend (swap out) one or more processes to lower the multiprogramming level
Multiprogramming Level

- Denning: adjust the multiprogramming level so that the mean time between page faults equals the mean time to process a fault
- Empirical studies show that CPU utilization is at a maximum with that approach
- Other strategies are similar: keep disk drive used for paging busy 50% of the time

Kernel Memory

- OS needs some page frames to hold the kernel and associated data structures
- Those page frames are usually "locked": not available to user processes, not subject to being paged out
- Memory allocation within OS page frames may be handled differently for efficiency (variation on buddy system often used)
Swap Space

- Pages selected for replacement need to be copied to secondary storage (disk)

- For performance reasons, the normal file system is often bypassed and parts of virtual memory (typically data pages) are stored in a special area on the disk called the swap space (accessed directly via disk sectors)

Example

- RAM has 4 pages, disk has room for 8 pages

- Four processes in the system: three partially in RAM, one completely swapped out
Summary: Storage Management

- Kernel memory is managed separately
- User processes are initially loaded from disk to RAM (text and data sections), additional pages created for stack and heap
- User pages may be evicted (replaced)
- User data pages must be copied to swap space when evicted
Reducing Page Table Size

- Consider a 32-bit address space and a page size of 4096 bytes
- Each process has $2^{20}$ page table entries; if each PTE is 4 bytes, that's 4,194,304 bytes
- Since each process needs a page table, too much memory would be used for page tables

The problem is worse for a 64-bit address space: $2^{52}$ page table entries, 8 bytes

Conclusion: page tables must be smaller

Processes usually do not use the full address space; exploit the fact that many (most) of the page table entries are not needed by using a hierarchical page table structure
Reducing Page Table Size

- Text and data sections copied from object code file (a.out)
- Stack and heap created during execution
- Most of address space is not used by typical process

Two general approaches:

- Inverted page table – use hashing to locate page number in page table (one entry in page table for each frame allocated to process)

- Hierarchical page table – subdivide page number into fields, use each field as an index into a specific level in the hierarchy
Reducing Page Table Size

Inverted page table:

Two-level page table:
Reducing Page Table Size

- Linear vs. two-level page table

- Cost: two-level hierarchical page table requires two memory accesses to retrieve the information about a given page

- Benefit: page table uses significantly less memory if the page table is sparsely populated
Example

- **System characteristics:**
  - 14-bit virtual addresses
  - 64-byte pages

- **Process:**
  - 2 pages for text
  - 2 pages for heap
  - 2 pages for stack

Example (2)

- Assume page table entries require 4 bytes:
  - control bits
  - frame number

- Linear page table would require 16 pages:
  - 256 PTEs x 4 bytes = 1024 bytes
  - 1024 bytes / 64 bytes = 16 pages
Example (3)

- Virtual addresses viewed as three fields:

- Two-level page table: page table directory and two page tables each stored in a separate page (3 pages total)

Example (4)

<table>
<thead>
<tr>
<th>Page Directory</th>
<th>Page of PT (@PFN:100)</th>
<th>Page of PT (@PFN:101)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PFN</td>
<td>valid?</td>
<td>PFN</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>—</td>
<td>0</td>
<td>23</td>
</tr>
<tr>
<td>—</td>
<td>0</td>
<td>—</td>
</tr>
<tr>
<td>—</td>
<td>0</td>
<td>—</td>
</tr>
<tr>
<td>—</td>
<td>0</td>
<td>80</td>
</tr>
<tr>
<td>—</td>
<td>0</td>
<td>59</td>
</tr>
<tr>
<td>—</td>
<td>0</td>
<td>—</td>
</tr>
<tr>
<td>—</td>
<td>0</td>
<td>—</td>
</tr>
<tr>
<td>—</td>
<td>0</td>
<td>—</td>
</tr>
<tr>
<td>—</td>
<td>0</td>
<td>—</td>
</tr>
<tr>
<td>—</td>
<td>0</td>
<td>—</td>
</tr>
<tr>
<td>—</td>
<td>0</td>
<td>—</td>
</tr>
<tr>
<td>—</td>
<td>0</td>
<td>—</td>
</tr>
<tr>
<td>—</td>
<td>0</td>
<td>—</td>
</tr>
<tr>
<td>—</td>
<td>0</td>
<td>—</td>
</tr>
<tr>
<td>—</td>
<td>0</td>
<td>—</td>
</tr>
<tr>
<td>—</td>
<td>0</td>
<td>—</td>
</tr>
<tr>
<td>—</td>
<td>0</td>
<td>—</td>
</tr>
<tr>
<td>—</td>
<td>0</td>
<td>—</td>
</tr>
<tr>
<td>—</td>
<td>0</td>
<td>—</td>
</tr>
<tr>
<td>—</td>
<td>0</td>
<td>—</td>
</tr>
<tr>
<td>—</td>
<td>0</td>
<td>—</td>
</tr>
<tr>
<td>—</td>
<td>0</td>
<td>—</td>
</tr>
<tr>
<td>101</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>101</td>
<td>1</td>
<td>—</td>
</tr>
</tbody>
</table>
Example (5)

Linux Virtual Memory (32 bit addresses)

Memory layout:
- 3 GB in user space
- 1 GB in kernel space

Loaded from file:
- machine language
- static data

Created during execution:
- stack
- heap
Linux Address Translation

Linux uses a two-level page table for 32-bit addresses (additional levels for 64-bit addresses)

- Page size: 4096 bytes
- Upper 10 bits used to identify page directory
- Middle 10 bits used to identify page table
- Lower 12 bits are the page offset
- Page table entries are 4 bytes, so 1024 PTEs per page (same with PDEs)
Linux Address Translation

Two-level page table efficient:

- virtual memory space is sparsely populated (large sections unused)
- many PDEs will be unused
- top-level table stored in one page
- each second-level table stored in one page
- page faults take two memory accesses, but relatively rare

4-level page table for 64-bit addresses:
Recap: Virtual Memory

- Provides illusion of very large main memory
  - sum of memory for all processes can be larger than physical memory
  - address space of one process can be larger than physical memory
- Allows main memory to be well utilized
- Simplifies memory management (relocation, protection and sharing)
- Exploits memory hierarchy to keep average access time low