Administrivia

• Hope everyone had a good break

• First good news: NachOS project extension to thursday
  – Please use digital drop box to hand in assignment
  – Grading by appointment starting Monday (to give chance for those taking slack days)

• New NachOS assignment available (most of it—still fiddling with cross compiler, but that shouldn’t matter)
  – Pretty tough assignment; will give some help in class on Thursday

• Exam 2: Tentatively on Tax day 4/15

• Homework assigned on Thursday, due 4/10 to prepare for exam
• Huge disconnect with my illness then spring break

• We were discussing memory management; some important concepts
  – Multiple processes sharing memory
    * Requirements for memory management: relocation, protection and sharing
  – Logical Address Space vs. Physical address space
    * Dynamic Run-time translation: code uses logical addresses, which dynamically get translated to physical addresses
    * Need Translation and Protection with very memory access –
How to allocate memory among the processes

- Contiguous allocation (one memory chunk per process):
  - fixed partitions
  - dynamic partitions
  - buddy system

- Non-contiguous allocation (multiple memory chunks per process):
  - paging and segmentation
Paging

• Now we relax the restriction of contiguous allocation

• Idea:
  – Divide physical memory into equal size small parts called **frames**
  – Divide each process’ logical memory into pages (same size as frame)
  – To run a program of size $n$ pages, need to find $n$ free frames and load the program (not necessarily contiguous)

• This is the memory management model in NachOS
Size of the Frame

- Paging is like fixed partitions (one size), but you allow each process to have multiple partitions
  - Again, if you were printing a single bill, which bill would you print?
- If the size is too large, we risk internal fragmentation
- If it is too small, the overhead to manage it is big
- The page size is a power of 2 (simplifies translation), usually in the range 1k to 64k
Paging Hardware

Logical address

physical address

CPU

p d

page table

f d

physical memory

f pages

f0000...0000

f1111...1111
Example

logical memory

page0

page1

page2

page3

physical memory

page table

1
4
3
7

page 0

page 2

page 1

page 3
<table>
<thead>
<tr>
<th></th>
<th>Logical Memory</th>
<th>Physical Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>e</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>g</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>h</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>i</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>k</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>l</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>m</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>n</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>o</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>p</td>
<td></td>
</tr>
</tbody>
</table>

- Memory is 32-bytes, page is 4-bytes
• Free frames are a property of the physical memory; one table called the frame table can keep track of the state of the memory
  – Frame table keeps track of whether frames are full or empty
  – In NachOS you are using the Bitmap object for that
• How does paging provide protection?
• The page table for each process is managed by the OS; user cannot modify it directly
• When the OS allocate pages, it marks the entries in the page table. OS sets valid bit
• As long as the logical address is valid, the physical address will be (trap to OS if invalid page)
• Keep track of the size of the page table
Discussion

• Paging is similar to fixed partitions, but allows many partitions for each process
  – Advantages: Much less internal fragmentation; no external fragmentation; will later allow us to only keep a portion of the process memory resident; easy to protect and share
  – Disadvantages: more complex to implement than contiguous allocation

• Steps for memory access:
  1. From logical address find page number
  2. Access the page table to get the corresponding frame
  3. Physical address = frame number : offset

• How does it do in terms of our 3 requirements (relocation, protection and sharing)?

• How is the page table implemented? In memory?
• It is possible to share one or more pages (by having them appear in more than one page table)

• Do you share data or code? Why?
  – Code can be shared if it is **re-entrant**; it is not self-modifying
  – Does the code have to appear in the same logical pages for each process?
Page Table Organization

• What is involved in address translation?
  – From the address find the page number
  – Look in the page table to find the appropriate entry
    * how should the page table be implemented?
  – Get the physical frame number and generate the physical address

• How big is the page table? Example: assume an address space of 4 gigabytes \(2^{32}\) and a page size of 1kbyte \(2^{10}\)

• Where should the page table be kept? In hardware? In memory (real or virtual)?

• Many instructions needed for every memory reference?? This is inefficient!
Making it Real

• To Recap, we have two problems:
  1. Inefficient to have several instructions, and two accesses to memory just to read the page table entry
     – This is needed even if the value is in the cache?
  2. The size of the page table can be very big; how to store it?
     – This is especially true when we get to virtual memory
Hardware Support

• Evaluate based on: access time; cost when we context switch; hardware/memory overhead required

• Implement the page table in a special register set? Good or bad?

• Page table in memory, have a page-table base register (PTBR) – better?
  – Still need two memory accesses to get the value

• Use a cache for the page table: Translation Lookaside Buffer (TLB)
  – A fully associative cache; usually 32-64 entries
  – Idea: most recently used entries will be in the TLB quick translation
  – If not in TLB (TLB miss), get it from the page table in memory (using the PTBR scheme above)
- Cache organization – associative caches most expensive, but have highest hit-ratio
- If the TLB hit ratio is high enough, we almost eliminate the cost of the page-table access
  - Almost the same performance as the page-table in registers scheme but cheaper and without limitations on page table size
- TLB solves memory access time problem; we will delay solutions for page table size until we discuss virtual memory basics
TLB Management

• TLB Misses
  – Miss occurs when a reference is generated to a page whose page table entry is not in the TLB
  – Must get the page table entry from memory, place the entry in the TLB and translate
    * Software TLB: TLB miss generates a trap to the OS – OS handles the miss.
    * Hardware TLB: CPU handles the miss in hardware
  – Software TLB most common; another example of the hardware/software tradeoff

• What happens on a context switch?
  – TLB entries are no longer valid (they refer to the old processes page entries)
  – Two options: (1) clear the TLB, or (2) save/restore the TLB entries into the PCB as part of the context switch
• Memory-management scheme that supports user view of the memory
• A program is a collection of segments
• Segments are “variable size pages”
• No internal fragmentation (external possible)
• Similar to dynamic partitioning, except each process may have more than one partition
• Must keep track of length of segment (not just frame number)
Segmentation Implementation

• Logical address consists of a two tuple (segment-number, offset)

• Each process has a segment table
  – Each segment has a base (physical address where it starts) and a limit (length of the segment)

• Hardware support
  – Segment-Table Base Register (STBR) keeps the location of the table in memory
  – Segment-Table Length Register (STLR) keeps the number of segments used by the program (used for protection)

• which is bigger, a page table entry or a segment table entry?

• which is bigger, a page table or a segment table?
Segmentation Hardware

CPU → s d → limit base → physical address → Physical Memory

- yes: physical address
- no: trap; memory protection error

Segmentation Hardware

CPU → s d → limit base → physical address → Physical Memory

- yes: physical address
- no: trap; memory protection error
Example Segmentation

logical address space/User’s view

<table>
<thead>
<tr>
<th>limit</th>
<th>base</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1400</td>
</tr>
<tr>
<td>400</td>
<td>6300</td>
</tr>
<tr>
<td>400</td>
<td>4300</td>
</tr>
<tr>
<td>1100</td>
<td>3200</td>
</tr>
<tr>
<td>1000</td>
<td>4700</td>
</tr>
</tbody>
</table>

segment 0
segment 3
segment 2
segment 4
segment 1

Sqrt
symbol table
main program

sub-routine
stack
Sharing with Segmentation

Logical Memory

process P1

editor
segment 0

data 1
segment 1

Logical Memory

process P2

editor
segment 0

data 2
segment 1

<table>
<thead>
<tr>
<th>limit</th>
<th>base</th>
</tr>
</thead>
<tbody>
<tr>
<td>25286</td>
<td>43062</td>
</tr>
<tr>
<td>4425</td>
<td>68384</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>limit</th>
<th>base</th>
</tr>
</thead>
<tbody>
<tr>
<td>25286</td>
<td>43062</td>
</tr>
<tr>
<td>8850</td>
<td>90003</td>
</tr>
</tbody>
</table>

43062 editor
68348 data 1
72773
90003 data 2
98553
Discussion

• Which is better paging or segmentation?
• Segments are nice from the users perspective
  – Can associate properties with the segment (protection, sharing, etc..) rather than arbitrary pages
• External fragmentation
  – May force the long-term scheduler to schedule a low-priority process that fits over a high-priority process that doesn't
• If the segments are too small
  – Overhead for managing them becomes too high; big segment tables, expensive search to find holes
• What if we make the whole process a segment?
• What if we make each byte a segment?
• Just use paging?
Combining Segmentation and Paging

- Segments are nice in that they correspond to user view

- Pages are nice in that no external fragmentation and simple management

- Several CPU’s/OS’s support combined segmentation and paging (including Pentium)

- Each process has several segments

- Each segment is paged (consists of several pages), and has its own page table

- Tricky, but concept is simple
Segmentation with Paging