Memory Management

The Virtual Memory Abstraction

Physical Memory
- Unprotected address space
- Limited size
- Shared physical frames
- Easy to share data

Virtual Memory
- Programs are isolated
- Arbitrary size
- All programs loaded at "0"
- Sharing is possible

Address spaces: Physical and Virtual

Physical address space consists of the collection of memory addresses supported by the hardware

Virtual address space consists of the collection of addresses that the process can "touch"

Note: CPU generates virtual addresses

Address Translation

A function that maps \langle pid, virtual address \rangle into physical address

Advantages:
- Protection
- Relocation
- Data sharing
- Multiplexing
Protection

At all times, the functions used by different processes map to disjoint ranges.

Relocation

The range of the function used by a process can change over time.

Relocation

The range of the function used by a process can change over time.

Data Sharing

Map different virtual addresses of different processes to the same physical address.
Contiguity

Contiguous addresses in the domain need not map to contiguous addresses in the codomain.

Multiplexing

The domain (set of virtual addresses) that map to a given range of physical addresses can change over time.
Multiplexing

The domain (set of virtual addresses) that map to a given range of physical addresses can change over time.

One idea, many implementations

- Base and limit
- Segment table
  - maps variable-sized ranges of contiguous VAs to a range of contiguous PAs
- Page table
  - maps fixed-size ranges of contiguous VAs to fixed sized ranges of contiguous PAs
- Paged segmentation
- Multilevel page tables
- Inverted page table

It's all just a lookup...

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>30940</td>
</tr>
<tr>
<td>1</td>
<td>56abe03</td>
</tr>
<tr>
<td>10</td>
<td>240421</td>
</tr>
<tr>
<td>unmapped</td>
<td></td>
</tr>
<tr>
<td>unmapped</td>
<td></td>
</tr>
<tr>
<td>unmapped</td>
<td></td>
</tr>
<tr>
<td>FFFFF</td>
<td>d82a04</td>
</tr>
</tbody>
</table>
### Base & Limit

- CPU Logical addresses ≤ yes
- Memory Exception
- physical address space
- 500
- 1000
- Limit Register
- Base Register

**Implementation**
- HW: Add base and bound registers to CPU
- SW: Add base and bound registers to PCB
  - Context switch, change B&B
  - Privileged

**Contiguous Allocation**
- contiguous virtual addresses are mapped to contiguous physical addresses
- Protection is easy, but sharing is hard
- Two copies of emacs: want to share code, but have data and stack distinct...
- Managing heap and stack dynamically is hard
  - We want them as far as possible in virtual address space, but...

### On Base & Limit

### Contiguous allocation:
- multiple variable partitions
  - OS keeps track of empty blocks ("holes")
  - Initially, one big hole!
  - Over time, a queue of processes (with their memory requirements) and a list of holes
  - OS decides which process to load in memory next
  - Once process is done, it releases memory

### Strategies for Contiguous Memory Allocation

- **First Fit**
  - Allocate first big-enough hole

- **Next Fit**
  - As first fit, but start to search where you previously left off

- **Best Fit**
  - Allocate smallest big-enough hole
**Fragmentation**

- External fragmentation
  - Unusable memory between units of allocation

- Internal fragmentation
  - Unusable memory within a unit of allocation

**Eliminating External Fragmentation: Compaction**

- Relocate programs to coalesce holes

- Problem with I/O
  - Pin job in memory while it is performing I/O
  - Do I/O in OS buffers
Eliminating External Fragmentation: Swapping

- Preempt processes and reclaim their memory
- Move images of suspended processes to backing store

From a user's perspective, a process is a collection of distinct logical address spaces

We call these logical address spaces segments

E Pluribus Unum

Implementing Segmentation

Segment table generalizes base & limit
On Segmentation

- Sharing a segment is easy!
- Protection bits control access to shared segments
- External fragmentation...
- Each process maintains a segment table, which is saved to PCB on a context switch
- Fast?
- How do we enlarge a segment?

Paging

- Allocate VA & PA memory in fixed-sized chunks (pages and frames, respectively)
  - memory allocation can use a bitmap
  - typical size of page/frame: 4KB to 16KB
- Gives illusion of contiguity...
  - ...but adjacent pages in VA need not map to contiguous frames in PA
- Of course, now internal fragmentation...

Virtual address

- Two components
  - page number
  - offset within page

Virtual address

- Two components
  - page number - how many pages in the VA
  - offset within page - how large is a page?
- To access a piece of data
  - extract page number
  - extract offset
  - translate page number to frame number
  - access data at offset in frame
Virtual address

- Two components
  - page number - how many pages in the VA
  - offset within page - how large is a page?

- To access a piece of data
  - extract page number
  - extract offset
  - translate page number to frame number
  - access data at offset in frame

Basic Paging Implementation

- Page table

- CPU

- Logical addresses

- Frame size

- Physical addresses

- Memory Exception

Speeding things up

- TLB hit

- TLB miss

- EAT: \((1+\epsilon)\alpha + (2+\epsilon)(1-\alpha)\) = \(2+\epsilon-\alpha\) (\(\alpha\): hit ratio)

Sharing

- Processes share pages by mapping virtual pages to the same memory frame
  - code segments of processes running same program can share pages with executables

- Fine tuning using protection bits (rwx)
Memory Protection

Used valid/invalid bit to indicate which mappings are active

<table>
<thead>
<tr>
<th>Protection bits</th>
<th>Memory Frames</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

What happens on a TLB miss?

Can be handled in software or hardware

Software
- TLB generates trap
- Switch to kernel mode
- OS does translation
- OS loads new TLB entry and returns from trap

On Context Switch
- Flush TLB
- add PID tag to TLB
- add a CPU register
- change PID register on context switch

Hardware
- HW includes PTB register
- HW follows pointer and does look up in page table
- Exception handler invoked only if no/bad mapping/permission

On Context Switch
- change value stored in PTB register
- Flush TLB

Space overhead

Two sources
- data structure overhead (the page table)
- fragmentation
  - How large should a page be?

Overhead for paging:

\[(\text{#entries} \times \text{sizeofEntry}) + (\text{#"segments"} \times \text{pageSize}/2) = (\text{VA Size}/\text{pagesize}) \times \text{sizeofEntry} + (\text{#"segments"} \times \text{pageSize}/2)\]

Size of entry
- enough bits to identify physical page (log2 (PA Size / page size))
- should include control bits (valid, modified, referenced, etc)
- usually word or byte aligned

Computing paging overhead

1 MB maximum VA, 1 KB page, 3 segments (program, stack, heap)

\[ ((2^{20} / 2^{10}) \times \text{sizeofEntry}) + (3 \times 2^3) \]

- If I know PA is 64 KB then sizeofEntry = 6 bits (2^6 frames) + control bits
- if 3 control bits, byte aligned size of entry: 16 bits
Oops...

What is the size of the page table for a machine with 64-bit addresses and a page size of 4KB?

Good news

Much of the space is unused

Use a smarter data structure to capture page table

Tree!

Examples

Two-level paging

Outer page table fits in a page

Rest of page table allocated in page-size chunks

Examples

Two-level paging

64-bit address space

4Kb pages

4 bytes PTE

32-bit address space

4Kb pages

4 bytes PTE

Multi-level Paging

Structure virtual address space as a tree

Virtual address of a SPARC

0

256

16384

4096

1024

128

16

4K

8K

16K
Examples

Two level paging
- Outer page table fits in a page
- Rest of page table allocated in page-size chunks
- Internal fragmentation (where?)
- Increased TLB miss time

The Challenge of Large Address Spaces

With large address spaces (64-bits) page tables become cumbersome
- 5/6 levels of tables
A new approach—make tables proportional to the size of the physical, not the virtual, address space
- Virtual address space is growing faster than physical

Page Registers (a.k.a. Inverted Page Tables)

For each frame, a register containing
- Residence bit
  - Is the frame occupied?
- Page # of the occupying page
- Protection bits

An example
- 16 MB of memory
- Page size: 4k
- # of frames: 4096
- Used by page registers (8 bytes/register): 32 KB
- Overhead: 0.2%
- Insensitive to size of virtual memory

Examples

64-bit VA; 2K page; 4 byte/entry

How many levels?
- Each page table includes 512 entries ($2^9$)
- Number of pages = $2^{64}/2^{11}$
- Number of levels = $53/9 = 6$ (rounded up)
Basic Inverted Page Table Architecture

CPU → pid → p → offset → Physical Memory

Inverted Page Table

Where have all the pages gone?

- Searching 32KB of registers on every memory reference is not fun.
- If the number of frames is small, the page registers can be placed in an associative memory—but...
- Large associative memories are expensive.
  - hard to access in a single cycle.
  - consume lots of power.

The BIG picture

CPU → vaddr → Translator Box → Physical memory

The BIG picture

CPU → vaddr → Translator Box

- if no match
- if match

Virtually addressed cache

- if no match
- if match

Physically addressed cache

Segment and page table lookup

- if no match
- if match

PTBR (per process) → vpage → dictionary

Main memory
**Time Overhead**

- Average Memory Access Time (AMAT)
  \[ \text{AMAT} = T_{L1} + (P_{L1\text{miss}} \times T_{L1\text{miss}}) \]

\[ T_{L1\text{miss}} = T_{TLB} + (P_{TLB\text{miss}} \times T_{TLB\text{miss}}) + T_{L2} + (P_{L2\text{miss}} \times T_{\text{mem}}) \]

\[ T_{TLB\text{miss}} = \#\text{references} \times (T_{L2} + P_{L2\text{miss}} \times T_{\text{mem}}) \]

**Demand Paging**

- Code pages are stored in a file on disk
  - some are currently residing in memory–most are not
- Data and stack pages are also stored in a file
- OS determines what portion of VAS is mapped in memory
  - this file is typically invisible to users
  - file only exists while a program is executing
- Creates mapping on demand

**Page-Fault Handling**

- References to a non-mapped page (i in page table) generate a page fault

Handling a page fault:
1. Processor runs interrupt handler
2. OS blocks running process
3. OS finds a free frame
4. OS schedules read of unmapped page
5. When read completes, OS changes page table
6. OS restarts faulting process from instruction that caused page fault
Taking a Step Back

- Physical and virtual memory partitioned into equal-sized units (respectively, frames and pages)
- Size of VAS decoupled to size of physical memory
- No external fragmentation
- Minimizing page faults is key to good performance

Page replacement

- Local vs Global replacement
  - Local: victim chosen from frames of faulty process
    - fixed allocation per process
  - Global: victim chosen from frames allocated to any process
    - variable allocation per process
- Many replacement policies
  - Random, FIFO, LRU, Clock, Working set, etc.
- Goal is minimizing number of page faults

FIFO Replacement

- First block loaded is first replaced
- Low overhead
- Commonly used

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>d</th>
<th>g</th>
<th>a</th>
<th>f</th>
<th>d</th>
<th>g</th>
<th>a</th>
<th>f</th>
<th>c</th>
<th>b</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F1</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F3</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

M M H M M M M M M M M M M M
**LRU Replacement**

- Replace block referenced least recently
- Reference stack
  - referenced block moved to top of stack
  - on page fault, block on bottom of stack is replaced and new block is placed on top of stack
- Difficult to implement

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a</th>
<th>d</th>
<th>g</th>
<th>a</th>
<th>f</th>
<th>d</th>
<th>g</th>
<th>a</th>
<th>f</th>
<th>c</th>
<th>b</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>d</td>
<td>g</td>
<td>a</td>
<td>f</td>
<td>d</td>
<td>g</td>
<td>a</td>
<td>f</td>
<td>c</td>
<td>b</td>
</tr>
<tr>
<td>F1</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>d</td>
<td>g</td>
<td>a</td>
<td>f</td>
<td>d</td>
<td>g</td>
<td>a</td>
<td>f</td>
<td>c</td>
<td>b</td>
</tr>
<tr>
<td>F2</td>
<td>b</td>
<td>d</td>
<td>g</td>
<td>a</td>
<td>f</td>
<td>d</td>
<td>g</td>
<td>a</td>
<td>f</td>
<td>c</td>
<td>b</td>
<td>g</td>
<td>a</td>
</tr>
<tr>
<td>F3</td>
<td>b</td>
<td>a</td>
<td>d</td>
<td>g</td>
<td>a</td>
<td>f</td>
<td>d</td>
<td>g</td>
<td>a</td>
<td>f</td>
<td>c</td>
<td>b</td>
<td>g</td>
</tr>
</tbody>
</table>

**Clock Replacement**

- First-In-Not-Used - First-Out replacement
- Like FIFO, but add a “used” bit (*) for each queue entry and make queue circular
- Clock hand points to orange frame

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a</th>
<th>d</th>
<th>g</th>
<th>a</th>
<th>f</th>
<th>d</th>
<th>g</th>
<th>a</th>
<th>f</th>
<th>c</th>
<th>b</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>F1</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>F2</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td>F3</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
</tr>
</tbody>
</table>

**Optimal Replacement**

- Replace block referenced furthest in future
- Minimum number of faults
- Impossible to implement

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a</th>
<th>d</th>
<th>g</th>
<th>a</th>
<th>f</th>
<th>d</th>
<th>g</th>
<th>a</th>
<th>f</th>
<th>c</th>
<th>b</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>F1</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>F2</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td>F3</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
</tr>
</tbody>
</table>

**Working Set Replacement**

- Global replacement policy
- \( WS_t = \) set of pages referenced in \((t-T+1, t)\)
- A page is replaced at \( t \) if it does not belong to \( WS_t \)
- pages not necessarily replaced at page fault time!
- adapts allocation to changes in locality

\( T = 4 \)

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a</th>
<th>d</th>
<th>g</th>
<th>a</th>
<th>f</th>
<th>d</th>
<th>g</th>
<th>a</th>
<th>f</th>
<th>c</th>
<th>b</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>F1</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td>F2</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
</tr>
<tr>
<td>F3</td>
<td>g</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>M</td>
<td>M</td>
<td>H</td>
<td>M</td>
<td>M</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>M</td>
<td>M</td>
<td>M</td>
</tr>
</tbody>
</table>
Thrashing

- If too much multiprogramming, pages tossed out while needed
- One program touches 50 pages
  - With enough pages, 100ns/ref
  - If too few and faults every 5th reference
    - 10ms for disk IO
    - One reference now costs 2ms: 20,000 times slowdown