Frequently asked questions from the previous class survey

- Where is the page table stored?
- Does the TLB contain page-to-frame mappings?
- Is there always an eviction on a TLB miss?
Topics covered in this lecture

- Shared pages
- Page sizes
- Structure of Page tables
  - Hashed Page Tables
  - Inverted Page Tables

All who joy would win must share it. Happiness was born a Twin.

George Byron
Reentrant Code

- A computer program or subroutine is called **reentrant** if:
  - It can be *interrupted* in the middle of its execution and
  - Then safely called again ("re-entered") *before* its previous invocations complete execution

**Non-self-modifying**
- Does not change during execution

Two or more processes can:
  1. Execute same code at same time
  2. Will have different data

Each process has:
- Copy of registers and data storage to hold the data
Shared Pages

- System with N users
  - Each user runs a text editing program

- Text editing program
  - 150 KB of code
  - 50 KB of data space

- 40 users
  - Without sharing: 8000 KB space needed
  - With sharing: 150 + 40 x 50 = 2150 KB needed

Shared Paging
Shared Paging

- Other heavily used programs can be shared
  - Compilers, runtime libraries, database systems, etc.

- To be shareable:
  1. Code must be reentrant
  2. The OS must enforce read-only nature of the shared code

---

You might not write well every day, but you can always edit a bad page. You can't edit a blank page.

Jodi Picoult
Paging and page sizes

- On average, ½ of the final page is empty
  - Internal fragmentation: wasted space

- With \( n \) processes in memory, and a page size \( p \)
  - Total \( np/2 \) bytes of internal fragmentation

- Greater page size = Greater fragmentation

But having small pages is not necessarily efficient

- Small pages mean programs need more pages
  - Larger page tables
  - 32 KB program needs
    - 4 8-KB pages, but 64 512-byte pages

- Context switches can be more expensive with small pages
  - Need to reload the page table
Transfers to-and-from disk are a page at a time

- Primary Overheads: Seek and rotational delays
- Transferring a small page almost as expensive as transferring a big page
  - 64 x 15 = 960 msec to load 64 512-bytes pages
  - 4 x 25 = 100 msec to load 4 8KB pages

- Here, large pages make sense

Overheads in paging:
Page table and internal fragmentation

- Average process size = s
- Page size = p
- Size of each page entry = e
- Pages per process = s/p
  - s/e/p: Total page table space

- Total Overhead = s/e/p + p/2
  - Page table overhead
  - Internal fragmentation loss
Looking at the overhead a little closer

- Total Overhead = \( \frac{se}{p} + \frac{p}{2} \)
  - Increases if \( p \) is small
  - Increases if \( p \) is large

- Optimum is somewhere in between
- First derivative with respect to \( p \)
  \[-se/p^2 + \frac{1}{2} = 0 \]
  \[ p = \sqrt{2se} \]

Optimal page size: Considering only page size and internal fragmentation

- \( p = \sqrt{2se} \)
- \( s = 128 \text{KB} \) and \( e = 8 \) bytes per entry

- Optimal page size = 1448 bytes
  - In practice we will never use 1448 bytes
  - Instead, either 1K or 2K would be used
    - Why? Pages sizes are in powers of 2 i.e. \( 2^x \)
    - Deriving offsets and page numbers is also easier
Pages sizes and size of physical memory

- As physical memories get bigger, page sizes get larger as well
  - Though *not linearly*

- Quadrupling physical memory size rarely even doubles page size

---

All problems in computer science can be solved by another level of indirection. Except, of course, the problem of too many indirections!

—David Wheeler

**Structure of the Page Table**
Typical use of the page table

- Process refers to addresses through pages’ **virtual** address
- Process has page table
- Table has entries for pages that process uses
  - One slot for each page
    - Irrespective of whether it is valid or not
- Page table **sorted** by virtual addresses

Basic Paging Hardware

- Logical Address
- Page offset
- Page number
- CPU
- Logical Address
- Physical Address
- Page Table
- Frame f
- Physical Address
- $f000...000$
- $f111...111$
## Structure of the Page Table

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

## Hierarchical Paging

- Logical address spaces: $2^{32} \sim 2^{64}$
- Page size: $4\text{KB} = 2^x \times 2^{10} = 2^{12}$
- Number of page table entries?
  - Logical address space size/page size
  - $2^{32}/2^{12} = 2^{20} \approx 1 \text{ million entries}$
- Page table entry = 4 bytes
  - Page table for process = $2^{20} \times 4 = 4 \text{ MB}$
Issues with large page tables

- Cannot allocate page table **contiguously** in memory

- Solution:
  - Divide the page table into smaller pieces
  - Page the page-table

Two-level Paging

<table>
<thead>
<tr>
<th>Page number</th>
<th>Page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>12</td>
</tr>
</tbody>
</table>

32-bit logical address
Two-level Paging

<table>
<thead>
<tr>
<th>Outer Page</th>
<th>Inner Page</th>
<th>Page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>

32-bit logical address

Address translation in two-level paging

Actual Physical address

Physical memory frame

Track pages of page-table

Page of page table

P1

P2

\( d \)
Two-level Page tables:
The outer page table

4 GB address space split into 1024 chunks

Each entry represents 4 MB

Page size is 4 KB

Two-level Page tables:
Case where only a few entries are needed

4 GB address space split into 1024 chunks

Each entry represents 4 MB

Unused by program

Page size is 4 KB
Two-level Page tables

Address space has a million pages
But ONLY 4 page tables are actually needed

Computing number of page tables in hierarchical paging

• There is 1 outer table with $2^{11}$ entries
• Each outer table entry points to an inner page table
  — So, there are $2^{11}$ inner page tables
• Total number of page tables = 1 + $2^{11}$
• Total number of entries = $2^{11} + 2^{11} \times 2^{11}$
Let's try 2-level paging for a 64-bit logical address space

- Outer page table has $2^{42}$ entries!
- **Divide the outer page table** into smaller pieces?

Why hierarchical tables may strain 64-bit architectures

- In our previous example
  - There would be $2^{32}$ entries in the outer page table
- We could keep going
  - 4-level page tables …
- But all this results in a **prohibitive** number of memory accesses
Hashed page tables

- An approach for handling address spaces > $2^{32}$
- Virtual page number is hashed
  - Hash used as key to enter items in the hash table
- The value part of table is a linked list
  - Each entry has:
    1. Virtual page number
    2. Value of the mapped page frame
    3. Pointer to next element in the list
Searching through the hashed table for the frame number

- Virtual page number is hashed
  - Hashed key has a corresponding value in table
    - Linked List of entries

- Traverse linked list to
  - Find a matching virtual page number

Hash tables and 64-bit address spaces

- Each entry refers to several pages instead of a single page
- Multiple page-to-frame mappings per entry
  - Clustered page tables
- Useful for sparse address spaces where memory references are non-contiguous (and scattered)
Inverted page table

- Only 1 page table in the system
  - Has an entry for each memory frame

- Each entry tracks
  - Process that owns it (pid)
  - Virtual address of page (page number)
Inverted Page table

- CPU
- Logical Address
- Physical Address
- Page Table
- Frame i

Profiling the inverted page table

- **Decreases** the amount of memory needed
- **Search time** increases
  - During page dereferencing
- **Stored based on frames**, but searched on pages
  - Whole table might need to be searched!
Other issues with the inverted page table

- Shared paging
  - Multiple pages mapped to same physical memory

- Shared paging **NOT possible** in inverted tables
  - Only 1 virtual page entry per physical page
    - Stored based on frames

PAGING IN REAL-WORLD SYSTEMS
x86-64

- Intel: IA-64 Itanium
  - Not much traction
- AMD: x86-64
  - Intel adopted AMD’s x86-64 architecture
- 64-bit address space: $2^{64}$ (16 exabytes)
- Currently x86-64 provides
  - 48-bit virtual address [Sufficient for 256 TB]
  - Page sizes: 4 KB, 2 MB, and 1 GB
  - 4-level hierarchical paging

A typical paging scheme in the x86-64

<table>
<thead>
<tr>
<th>1st-level</th>
<th>2nd-level</th>
<th>3rd-level</th>
<th>4th-level</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>9-bits</td>
<td>9-bits</td>
<td>9-bits</td>
<td>9-bits</td>
<td>12-bits</td>
</tr>
</tbody>
</table>
Optimization to eliminate levels in the x86-64

- High-end servers routinely have 2 TB RAM
- With 48-bit addressing and 4-level page tables we can have some optimizations
- Each physical frame on the x86 is 4 KB
- Each page in the 4th level page table maps 2 MB
  - If the entire 2MB covered by that page table is allocated contiguously in physical memory?
    - Page table entry one layer up can be marked to point directly to this region instead of page table
- Also improves TLB efficiency

ARM architectures

- iPhone and Android systems use this
- 32-bit ARM
  - 4 KB and 16 KB pages
  - 1 MB and 16 MB pages
    - Termed sections

There are two levels for TLBs:
A separate TLB for data
Another for instructions
Segmentation with Paging

- **Multics**: Each program can have up to 256K independent segments
  - Each with 64K 36-bit words

- **Intel Pentium**
  - 16K independent segments
  - Each segment has $10^9$ 32-bit words (4GB)
  - Few programs need more than 1000 segments, but many programs need large segments
Segmentation with Paging

- **32-bit x86**
  - Virtual address space within a segment has a 2-level page table
    - First 10-bits top-level page table, next 10-bits second-level page table, final 12-bits are the offsets within the page

- **64-bit x86**
  - 48-bits of virtual addresses within a segment
  - 4-level page table
    - Includes optimizations to eliminate one or two levels of the page table
How we got here …

Contiguous Memory ➔ External Fragmentation ➔ Pure Paging ➔ Single Address space ➔ Segmentation

Low Degree of Multiprogramming ➔ Virtual Memory

Memory Management: Why?

- Main objective of system is to execute programs
- Programs and data must be in memory (at least partially) during execution
- To improve CPU utilization and response times
  - Several processes need to be memory resident
  - Memory needs to be shared
Requiring the entire process to be in physical memory can be limiting

- **Limits** the size of a program
  - To the size of physical memory

- BUT the entire program is not always needed

Situations where the entire program need not be memory resident

- Code to handle rare error conditions

- Data structures are often allocated more memory than they need
  - Arrays, lists …

- Rarely used features
What if we could execute a program that is partially in memory?

- Program is **not constrained** by amount of free memory that is available.
- Each program uses **less** physical memory.
  - So, more programs can run.
- **Less I/O** to swap programs back and forth.

**Logical view of a process in memory**

```
max

stack

heap

data

text

low
```

- `{Function parameters, return addresses, and local variables}`
- `{Memory allocated dynamically during runtime}`
- `{Global variables}`
- `{Program code}`
Logical view of a process in memory

- stack
- heap
- data
- text

Requires actual physical space ONLY IF heap or stack grows

Sparse address spaces

- Virtual address spaces with holes
- Harnessed by
  - Heap or stack segments
  - Dynamically linked libraries
Loading an executable program into memory

- What if we load the entire program?
  - We may not need the entire program

- Load pages *only* when they are needed
  - **Demand Paging**
Differences between the swapper and pager

- **Swapper**
  - Swaps the *entire program* into memory

- **Pager**
  - Lazy swapper
  - Never swap a page into memory *unless* it is actually *needed*

The contents of this slide-set are based on the following references

