Virtual memory & memory hierarchy

Hung-Wei Tseng
Recap: What happens when we access data

- Processor sends load request to L1-$
  - if read hit — return data
  - if write hit — set dirty and update in the block
- if miss
  - Select a victim block
    - If the target “set” is not full — select an empty/invalidated block as the victim block
    - If the target “set is full — select a victim block using some policy
      - LRU is preferred — to exploit temporal locality!
  - If the victim block is “dirty” & “valid”
    - Write back the block to lower-level memory hierarchy
  - Fetch the requesting block from lower-level memory hierarchy and place in the victim block
    - If write-back or fetching causes any miss, repeat the same process
Recap: causes of $ misses

- Compulsory miss
  - Cold start miss. First-time access to a block
- Capacity miss
  - The working set size of an application is bigger than cache size
- Conflict miss
  - Required data block replaced by block(s) mapping to the same set
  - Similar collision in hash — if the conflict miss doesn’t go away even though you made the cache fully-associative — it’s a capacity miss
Recap: optimizations

• Software
  • Data layout — capacity miss, conflict miss, compulsory miss
  • Blocking — capacity miss, conflict miss
  • Loop fission — conflict miss — when $ has limited way associativity
  • Loop fusion — capacity miss — when $ has enough way associativity
  • Loop interchange — conflict/capacity miss

• Hardware
  • Increase block size — compulsory miss, but increase miss penalty
  • Increase way associativity — conflict miss, but increase hit time
  • Increase capacity — capacity miss, but $$$
Recap: Virtual memory
# Let’s dig into this code

```c
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sched.h>
#include <sys/syscall.h>
#include <time.h>

double a;

int main(int argc, char *argv[])
{
    int i, number_of_total_processes=4;
    number_of_total_processes = atoi(argv[1]);
    // Create processes
    for(i = 0; i < number_of_total_processes-1 && fork(); i++);
    // Generate rand see
    srand((int)time(NULL)+(int)getpid());
    a = rand();
    fprintf(stderr, "\nProcess %d is using CPU: %d. Value of a is %lf and address of a is %p\n", getpid(), a, &a);
    sleep(10);
    fprintf(stderr, "\nProcess %d is using CPU: %d. Value of a is %lf and address of a is %p\n", getpid(), cpu, a, &a);
    return 0;
}
```
Consider the following code ...

```c
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sched.h>
#include <sys/syscall.h>
#include <time.h>
double a;

int main(int argc, char *argv[])
{
    int i, number_of_total_processes=4;
    number_of_total_processes = atoi(argv[1]);
    for(i = 0; i< number_of_total_processes-1 && fork(); i++);
    srand((int)time(NULL)+(int)getpid());
    fprintf(stderr, "%Process %d is using CPU: %d. Value of a is %lf and address of a is %p\n", getpid(), cpu, a, &a);
    sleep(10);
    fprintf(stderr, "%Process %d is using CPU: %d. Value of a is %lf and address of a is %p\n", getpid(), cpu, a, &a);
    return 0;
}
```

- Consider the case when we run multiple instances of the given program at the same time on modern machines, which pair of statements is correct?
  1. The printed “address of a” is the same for every running instances
  2. The printed “address of a” is different for each instance
  3. All running instances will print the same value of a
  4. Some instances will print the same value of a
  5. Each instance will print a different value of a

A. (1) & (3)
B. (1) & (4)
C. (1) & (5)
D. (2) & (3)
E. (2) & (4)
Consider the case when we run multiple instances of the given program at the same time on modern machines, which pair of statements is correct?

① The printed “address of a” is the same for every running instances
② The printed “address of a” is different for each instance
③ All running instances will print the same value of a
④ Some instances will print the same value of a
⑤ Each instance will print a different value of a

A. (1) & (3)  
B. (1) & (4)  
C. (1) & (5)  
D. (2) & (3)  
E. (2) & (4)
Consider the following code ...

```c
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sched.h>
#include <sys/syscall.h>
#include <time.h>

double a;

int main(int argc, char *argv[])
{
    int i, number_of_total_processes=4;
    number_of_total_processes = atoi(argv[1]);
    for(i = 0; i< number_of_total_processes-1 && fork(); i++);
    srand((int)time(NULL)+(int)getpid());
    fprintf(stderr,
"Process %d is using CPU: %d. Value of a is %lf and address of a is %p\n", getpid(), cpu, a, &a);
    sleep(10);
    fprintf(stderr,
"Process %d is using CPU: %d. Value of a is %lf and address of a is %p\n", getpid(), cpu, a, &a);
    return 0;
}
```

Consider the case when we run multiple instances of the given program at the same time on modern machines, which pair of statements is correct?

1. The printed “address of a” is the same for every running instances
2. The printed “address of a” is different for each instance
3. All running instances will print the same value of a
4. Some instances will print the same value of a
5. Each instance will print a different value of a

A. (1) & (3)  
B. (1) & (4)  
C. (1) & (5)  
D. (2) & (3)  
E. (2) & (4)

If you still don’t know why — you need to take CSE120
If there is no abstraction between the processor and memory, the processor/cache needs to directly use main memory’s byte address to read/write data. How many of the following would be happening?

① The program’s memory footprint, including instructions/data, cannot exceed the capacity of the installed DRAM
② There is no guarantee the compiled program can execute on another machine if both machines have the same processor but different memory capacities
③ Two programs cannot run simultaneously if they use the same memory addresses
④ One program can maliciously access data from other concurrently executing programs

A. 0  
B. 1  
C. 2  
D. 3  
E. 4
If we can only use physical memory ...

- If there is no abstraction between the processor and memory, the processor/cache needs to directly using main memory’s byte address to read/write data. How many of the following would be happening?
  ① The program’s memory footprint, including instructions/data, cannot exceed the capacity of the installed DRAM
  ② There is no guarantee the compiled program can execute on another machine if both machine have the same processor but different memory capacities
  ③ Two programs cannot run simultaneously if they use the same memory addresses
  ④ One program can maliciously access data from other concurrently executing programs

A. 0
B. 1
C. 2
D. 3
E. 4
If there is no abstraction between the processor and memory, the processor/cache needs to directly use main memory’s byte address to read/write data. How many of the following would be happening?

① The program’s memory footprint, including instructions/data, cannot exceed the capacity of the installed DRAM
② There is no guarantee the compiled program can execute on another machine if both machine have the same processor but different memory capacities
③ Two programs cannot run simultaneously if they use the same memory addresses
④ One program can maliciously access data from other concurrently executing programs

A. 0  
B. 1  
C. 2  
D. 3  
E. 4
If we expose memory directly to the processor (I)

<table>
<thead>
<tr>
<th>Program</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0f00bb27 509cbd23 00005d24 000bd24 2ca422a0 130020e4 00003d24 2ca4e2b3</td>
<td>00c2e800 00000008 00c2f000 00000008 00c2f800 00000008 00c30000 00000008</td>
</tr>
</tbody>
</table>

What if my program needs more memory?
If we expose memory directly to the processor (II)

What if my program runs on a machine with a different memory size?
If we expose memory directly to the processor (III)

What if both programs need to use memory?
If we can only use physical memory ...

- If there is no abstraction between the processor and memory, the processor/cache needs to directly using main memory’s byte address to read/write data. How many of the following would be happening?

  1. The program’s memory footprint, including instructions/data, cannot exceed the capacity of the installed DRAM
  2. There is no guarantee the compiled program can execute on another machine if both machine have the same processor but different memory capacities
  3. Two programs cannot run simultaneously if they use the same memory addresses
  4. One program can maliciously access data from other concurrently executing programs

A. 0  
B. 1  
C. 2  
D. 3  
E. 4
Virtual memory

Program

Memory

Program

Virtual Memory Space

Virtual Memory Space

Instruction

Data

Instruction

Data
Virtual memory

- An abstraction of memory space available for programs/software/programmer
- Programs execute using virtual memory address
- The operating system and hardware work together to handle the mapping between virtual memory addresses and real/physical memory addresses
- Virtual memory organizes memory locations into “pages”
The virtual memory abstraction

Processor Core
Registers

load 0x0009

Page table

Main memory (DRAM)

Virtual Memory Space

Page #1
Demo revisited

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sched.h>
#include <sys/syscall.h>
#include <time.h>

#define _GNU_SOURCE

double a;

int main(int argc, char *argv[])
{
    int i, number_of_total_processes = 4;
    number_of_total_processes = atoi(argv[1]);
    for(i = 0; i < number_of_total_processes - 1 && fork(); i++);
    srand((int)time(NULL) + (int)getpid());
    fprintf(stderr, "Process %d is using CPU: %d. Value of a is %lf and address of a is %p\n", getpid(), cpu, a, &a);
    sleep(10);
    fprintf(stderr, "Process %d is using CPU: %d. Value of a is %lf and address of a is %p\n", getpid(), cpu, a, &a);
    return 0;
}
```

Process A

Process B

&a = 0x601090

Process A's Page Table

Process B's Page Table
Address translation

- Processor receives virtual addresses from the running code, main memory uses physical memory addresses.
- Virtual address space is organized into “pages”.
- The system references the page table to translate addresses.
  - Each process has its own page table.
  - The page table content is maintained by OS.
Demand paging

• Treating physical main memory as a “cache” of virtual memory
• The block size is the “page size”
• The page table is the “tag array”
• It’s a “fully-associate” cache — a virtual page can go anywhere in the physical main memory
Assume that we have 64-bit virtual address space, each page is 4KB, each page table entry is 8 Bytes, what magnitude in size is the page table for a process?

A. MB — $2^{20}$ Bytes
B. GB — $2^{30}$ Bytes
C. TB — $2^{40}$ Bytes
D. PB — $2^{50}$ Bytes
E. EB — $2^{60}$ Bytes
Size of page table

• Assume that we have 64-bit virtual address space, each page is 4KB, each page table entry is 8 Bytes, what magnitude in size is the page table for a process?
  A. MB — $2^{20}$ Bytes
  B. GB — $2^{30}$ Bytes
  C. TB — $2^{40}$ Bytes
  D. PB — $2^{50}$ Bytes
  E. EB — $2^{60}$ Bytes
Size of page table

• Assume that we have 64-bit virtual address space, each page is 4KB, each page table entry is 8 Bytes, what magnitude in size is the page table for a process?

A. MB — $2^{20}$ Bytes  
B. GB — $2^{30}$ Bytes  
C. TB — $2^{40}$ Bytes  
D. PB — $2^{50}$ Bytes  
E. EB — $2^{60}$ Bytes

\[
\frac{2^{64} \text{ Bytes}}{4 \text{ KB}} \times 8 \text{ Bytes} = 2^{55} \text{ Bytes} = 32 \text{ PB}
\]

If you still don’t know why — you need to take CSE120
Do we really need a large table?

Your program probably never uses this huge area!

If you still don’t know why — you need to take CSE120.
Do we really need a large table?

Dynamic allocated data: `malloc()`
Address translation in x86-64

<table>
<thead>
<tr>
<th>63:48 (16)</th>
<th>47:39 (9 bits)</th>
<th>38:30 (9 bits)</th>
<th>29:21 (9 bits)</th>
<th>20:12 (9 bits)</th>
<th>11:0 (12 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SignExt</td>
<td>L4 index</td>
<td>L3 index</td>
<td>L2 index</td>
<td>L1 index</td>
<td>page offset</td>
</tr>
</tbody>
</table>

X86 Processor

CR3 Reg.

512 entries

512 entries

512 entries

512 entries

11:0 (12 bits)

physical page #

page offset
When we have virtual memory...

• If an x86 processor supports virtual memory through the basic format of the page table as shown in the previous slide, how many memory accesses can a `mov` instruction that access data memory once incur?

A. 2  
B. 4  
C. 6  
D. 8  
E. 10
When we have virtual memory...

- If an x86 processor supports virtual memory through the basic format of the page table as shown in the previous slide, how many memory accesses can a `mov` instruction that access data memory once incur?
  
  A. 2  
  B. 4  
  C. 6  
  D. 8  
  E. 10
### Address translation in x86-64

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:48</td>
<td>SignExt</td>
</tr>
<tr>
<td>47:39</td>
<td>L4 index</td>
</tr>
<tr>
<td>38:30</td>
<td>L3 index</td>
</tr>
<tr>
<td>29:21</td>
<td>L2 index</td>
</tr>
<tr>
<td>20:12</td>
<td>L1 index</td>
</tr>
<tr>
<td>11:0</td>
<td>Page offset</td>
</tr>
</tbody>
</table>

- **SignExt**: Sign extension
- **L4, L3, L2, L1**: Levels of page tables
- **Page offset**: Offset within a page

---

**X86 Processor**

- **CR3 Reg.**: Contains the base address of the page tables

- **512 entries** at each level

---

May have 10 memory accesses for a "MOV" instruction! — 5 for instruction fetch and 5 for data access
When we have virtual memory...

• If an x86 processor supports virtual memory through the basic format of the page table as shown in the previous slide, how many memory accesses can a `mov` instruction that access data memory once incur?
  
  A. 2  
  B. 4  
  C. 6  
  D. 8  
  E. 10
Avoiding the address translation overhead
• TLB — a small SRAM stores frequently used page table entries
• Good — A lot faster than having everything going to the DRAM
• Bad — Still on the critical path
TLB + Virtual cache

- L1 $ accepts virtual address — you don’t need to translate

- Good — you can access both TLB and L1-$ at the same time and physical address is only needed if L1-$ misses

- Bad — it doesn’t work in practice
  - Many applications have the same virtual address but should be pointing different physical addresses
  - An application can have “aliasing virtual addresses” pointing to the same physical address

You really need “physical address” to judge if that’s what you want
Can we find physical address directly in the virtual address — Not everything — but the page offset isn’t changing!

Can we indexing the cache using the “partial physical address”?
— Yes — Just make set index + block set to be exactly the page offset

Virtually indexed, physically tagged cache

![Diagram of Virtually Indexed, Physically Tagged Cache]

- Virtual address
- Page table
- Physical address
- Tag
- Valid
- Set
- Block offset
- Virtual page number
- Page offset
- Physical page number
- Block set
- Index
Virtually indexed, physically tagged cache

memory address: $0 \times 0 \quad 8 \quad 2 \quad 4$

virtual page # index offset

memory address: $0b0000100000100100$

V virtual page # physical page #

<table>
<thead>
<tr>
<th>V</th>
<th>virtual page #</th>
<th>physical page #</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x29</td>
<td>0x45</td>
</tr>
<tr>
<td>1</td>
<td>0xDE</td>
<td>0x68</td>
</tr>
<tr>
<td>1</td>
<td>0x10</td>
<td>0xA1</td>
</tr>
<tr>
<td>0</td>
<td>0x8A</td>
<td>0x98</td>
</tr>
</tbody>
</table>

VD tag data

1 1 $0x00$ AABBCDDEEGGFFHH
1 1 $0x10$ IIJKKLLMMNNOOPP
1 0 $0xA1$ QQRSSSTTUUVVVwWX
0 1 $0x10$ YYZZAABBCDDEEFF
1 1 $0x31$ AABBCDDEEGGFFHH
1 1 $0x45$ IIJKKLLMMNNOOPP
0 1 $0x41$ QQRSSSTTUUVVVwWX
0 1 $0x68$ YYZZAABBCDDEEFF

hit?
Virtually indexed, physically tagged cache

- If page size is 4KB —

\[
lg(B) + lg(S) = lg(4096) = 12
\]

\[
C = ABS
\]

\[
C = A \times 2^{12}
\]

if \( A = 1 \)

\[
C = 4\text{KB}
\]
Virtual indexed, physical tagged cache limits the cache size

- If you want to build a virtual indexed, physical tagged cache with 32KB capacity, which of the following configuration is possible? Assume the operating system use 4K pages.
  A. 32B blocks, 2-way
  B. 32B blocks, 4-way
  C. 64B blocks, 4-way
  D. 64B blocks, 8-way
Virtual indexed, physical tagged cache limits the cache size

- If you want to build a virtual indexed, physical tagged cache with 32KB capacity, which of the following configuration is possible? Assume the operating system use 4K pages.
  
  A. 32B blocks, 2-way  
  B. 32B blocks, 4-way  
  C. 64B blocks, 4-way  
  D. 64B blocks, 8-way
If you want to build a virtual indexed, physical tagged cache with 32KB capacity, which of the following configuration is possible? Assume the operating system use 4K pages.

A. 32B blocks, 2-way
B. 32B blocks, 4-way
C. 64B blocks, 4-way
D. 64B blocks, 8-way

Exactly how Core i7 configures its own cache

\[
lg(B) + lg(S) = lg(4096) = 12
\]

\[
C = ABS
\]

\[
32KB = A \times 2^{12}
\]

\[
A = 8
\]
Announcements

• Assignment #4 is up on the website
• Regarding final exam —
  • 9/4 8am—6pm — any consecutive, non-stop 3-hour slot you pick
  • Open books, open notes, but it’s going to be twice longer than the midterm
  • Lockdown browser (considering)
• No zoom, no response to piazza posts for fairness
Now playing — Just an illusion (Julie Zahra)