Contents

Introduction to Virtual Memory

Pages and Address Translation

TLBs and TLB miss handling

Memory System Performance Example
Outline of Slide Set 10 for Lecture Section 01

Introduction to Virtual Memory

Pages and Address Translation

TLBs and TLB miss handling

Memory System Performance Example
Introduction to virtual memory

The problem solved by **caches** is:

DRAM is affordable for large memories, but is much too slow to keep up with a processor core. Small SRAM systems can keep up with a core, but larger SRAM systems cannot.

**Virtual memory solves totally different problems!**

Keeping this in mind helps reduce confusion between cache details and virtual memory details.
Problems solved by virtual memory

1. Main memory may not have room for all instructions and data of running programs. VM allows some instructions or data to “spill” to disk storage when main memory is full.

2. Each of many multiple running programs should have its own protected address space. Programs must not read or write each other's memory by accident.

3. Programs may need large regions of contiguous address space, for allocation of large arrays, but main memory may be fragmented.
Virtual memory: An overview

Page 1 of the lecture document called “Virtual Memory, The Big Picture” shows a simplified picture of address space management by 32-bit versions of operating systems such as Windows, Mac OS X, and Linux.

(The picture for 64-bit OS versions is mostly the same, but has wider addresses.)

Four programs are running: the OS kernel, and three processes.

Note: A running user program is called a process.
VM: The Big Picture

HARD DISK

MAIN MEMORY PHYSICAL ADDRESS SPACE

PROCESS 1 VIRTUAL ADDRESS SPACE

PROCESS 2 VIRTUAL ADDRESS SPACE

PROCESS 3 VIRTUAL ADDRESS SPACE

KERNEL stack, data, and text
Multitasking

Suppose the computer with the four running programs (kernel, three processes) has a single processor core.

Can there really be four programs running at the same time?
Virtual memory overview: An essential point

Physical memory (on the right in the diagram) and a hard disk (on the left in the diagram) are real, physical systems you could find inside a computer.

Virtual address spaces (middle of the diagram) exist only as concepts! Instructions and data of processes are really located in physical memory or on disk.

(In many computer systems with VM in 2017, flash memory plays the role that a hard disk does in the above discussion. It’s traditional to involve magnetic disks in explanation of VM, but it’s important to realize that VM can work with flash instead of disk.)
Outline of Slide Set 10 for Lecture Section 01

Introduction to Virtual Memory

Pages and Address Translation

TLBs and TLB miss handling

Memory System Performance Example
The memory belonging to a process is a collection of *pages*. A *page* is a group of words with consecutive addresses; we’ll see that there are constraints on the **size** and **base address** of a page.

The size of a page must be a power of two. In a simple VM system, all pages are the same size.

Textbook and lecture examples mostly use 4KB pages (so a page holds 4096 bytes, that is, 1024 32-bit words).
More about Pages

Let’s make a sketch of a single page.

The diagram in “Virtual Memory, The Big Picture” shows how the memory belonging to each process is organized in pages. How many pages is process 2 using for instructions, for its data segment, and for its stack?

A page will be either entirely in main memory or entirely on disk, not partly in one and partly in the other.
Address translation

Processes use **virtual addresses**, which have to be **translated** to **physical addresses** for accesses to caches and main memory.

Think about the addresses a process uses . . .

- The PC for a process contains a virtual address.
- Addresses encoded in MIPS `j` and `jal` instructions are virtual addresses.
- All pointer variables and arguments in a process contain virtual addresses.
- Array element addresses generated by a process are virtual addresses.
Where does address translation fit into instruction fetch and data memory access?

Example for a 32-bit system with one level of cache . . .
Address translation for 32-bit systems with 4 KB pages

A page contains $4 \times 2^{10}$ bytes $= 2^{12}$ bytes, so it takes 12 bits of an address to select a byte within a page. Bits 11–0 of an address are called the *page offset*.

Everything else in an address—bits 31–12—is called the *page number*.

*How are the page number and page offset used in address translation?*
Translation only works when . . .

For translation of a virtual address to a physical address to work, the virtual page being accessed must be in **physical memory**.

If a process tries to access a virtual page that is on **disk**, the **entire page** must first be copied into physical memory—only after that can address translation and memory access be completed.

(Useful to remember: Compared to just about every other component in a modern computer, a disk drive is horrifically slow. But compared to a human operating a keyboard and a mouse, a disk drive is insanely fast.)
Page tables

For every process, the OS kernel must maintain information about how many virtual pages the process has, and where these pages are physically located.

This information is stored in kernel memory in a data structure called a page table.

Typically many processes are running at the same time, so typically a kernel is simultaneously maintaining many page tables.
**Page Table Entries (PTEs)**

Information about a single virtual page is stored in a PTE. A PTE is typically one or two words in size.

A page table can be thought of as a big collection of PTEs.

*Let’s sketch an example PTE.*

There is no VPN (virtual page number) in a PTE! *So how is a VPN used in address translation?*
How often is translation needed?

Every instruction fetch by a process requires an address translation.

If the instruction is a load or a store, another translation is needed for the data address.

To keep things simple, we’ll pretend that when the kernel is running, untranslated physical addresses are used for instruction fetches and data access. (Addresses used by real kernels are sometimes physical, sometimes virtual.)
A simple proposal for address translation

On every instruction fetch by a process, the kernel could do the address translation by searching the page table belonging to the process.

The same kind of thing would happen a second time for the same instruction if the instruction were a load or a store.

*Would this lead to a reasonable design for a real computer system?*
Outline of Slide Set 10 for Lecture Section 01

Introduction to Virtual Memory

Pages and Address Translation

TLBs and TLB miss handling

Memory System Performance Example
TLBs (translation lookaside buffers)

TLBs are hardware circuits designed to make address translation fast.

A TLB contains some but not all of the information found in page tables. (So a TLB is like a cache for page tables.)

Most address translations are performed by a TLB without any kernel intervention.

The first diagram on the “TLB organization” lecture document shows where TLBs fit between the core and the caches.
TLBs fit between the core and the first level of caches

![Diagram showing TLBs between core and caches]
**TLB lookup**

A TLB can only hold a small fraction of the VPN-to-PPN translations belonging to a medium-size or large process—the process might have thousands of pages, but the TLB might have room for only 64 translations.

Lookup in a TLB to find a PPN is similar to lookup in an I-cache or D-cache to find an instruction or data word.

The role of a VPN in a TLB lookup is somewhat like the role of a complete instruction or data address in I-cache or D-cache lookup.
TLB lookup, continued

Many TLB designs are **fully-associative**—the entire VPN is used like a tag. The TLB is searched by simultaneously comparing the VPN from the instruction or data address against all of the VPNs in the TLB.

(Note: A TLB has a much smaller number of entries than an I-cache or a D-cache. So fully-associative lookup in a TLB is does not cause unreasonably high power use.)

Other TLB designs are **set-associative**, using part of the VPN as an index, and the rest of the VPN as a tag.
TLB hits and misses

A *TLB hit* occurs when a translation needed for instruction fetch or data access is found in the TLB. The needed PPN is generated *fast*—one clock cycle or so.

A *TLB miss* occurs when a needed translation is not in the TLB. **Attention:** A TLB miss causes an *exception!*

To review exceptions, see slides 75–86 in Set 7, and textbook Section 6.7.4.

(To be really precise: In ENCM 369, we will **model** TLB misses as exceptions. TLB misses really do cause exceptions in MIPS, but some TLB misses for other architectures do not require exception handling.)
Half-clock-cycle organization of the 5-stage MIPS R2000 pipeline (1985)

This works for LW, SW, and R-type instructions—details for branches and jumps are not shown.

Note how the work is distributed over 9 half-clock-cycles.

The pipelined processor designs of textbook Chapter 7 are simplified versions of this (now very old) “real-world” design.
Organization of the 5-stage MIPS R2000 pipeline, continued

“Real-world” processors in 2014 tend to have deeper pipelines—10 stages or more. There might be 2–3 stages for each of I-cache and D-cache access, so, for example, up to 2–3 D-cache accesses “in flight” at the same time, in different stages of handling LW or SW.

Exam questions on pipelining will be about the systems like the ones in Chapter 7, with simple memory, NOT the MIPS R2000 or any other “real-world” processor!

However, it’s important to know that TLB and cache steps have to be built into “real-world” pipeline designs.
TLB miss handling (Warning: complicated topic!)

I-cache and D-cache misses are handled by hardware.

But TLB misses can be handled entirely by software, or by a mix of software or hardware, depending on the processor design.

Use of software allows flexibility—different OS kernels can do TLB miss handling different ways on the same hardware.

Page 2 of the “TLB organization” lecture document has an overview of TLB miss handling. **Read that carefully.**
TLB miss handling: Quick summary

Viewing this slide is not a complete substitute for reading the “TLB organization” document!

The document describes three cases:

- The page is in physical memory but the translation is not in the TLB. This is handled *quickly* by copying the translation to the TLB and restarting the instruction that caused the miss.

- The page is on disk, **not** in physical memory. This is called a page fault. There is a delay of millions of core clock cycles while the page is copied from disk to memory.

- The VPN indicates an invalid memory access. Usually this would cause the kernel to terminate the offending process.
Water spilling

Adding water to the small cup will eventually cause a spill to the big bucket.
Data spilling

Small, fast storage is like the small cup, and big, slower storage is like the big bucket. Data has to spill when a storage system is full.

In virtual memory systems, main memory spills to disk when main memory is full.

The analogy has a weakness. Sequences of 1’s and 0’s differ from each other, and some are needed more urgently than others. You can’t say that about groups of water molecules in cups and buckets!
Disk access is slow!

Spilling pages to disk is better than killing processes due to insufficient main memory for the combined needs of all processes.

But because disk access is so slow relative to DRAM access, there is a **major performance penalty** if pages are frequently copied back and forth between main memory and disk.

Hard drives based on **flash** memory storage so-called solid-state drives, or SSDs—are **significantly faster** than hard drives built with magnetic disks. But even SSDs are much, much slower than DRAM.
Effect of hard drive speed on human behaviour

Source: http://xkcd.com/1328/
I have a bunch of things open right now.

Really, how bad could a fire be, compared to the pain of rebooting the OS and restarting applications?
Review: Processes and Address Spaces

Processes are running user programs.

Each process has its own virtual address space—a collection of virtual pages.

For all processes, the operating system kernel manages the association of virtual pages with physical pages.

TLB circuits allow fast translation from virtual page numbers (VPNs) to physical page numbers (PPNs).
Process switching

Suppose instructions from Process 1 are running. A switch to another process is caused by an exception, which could be one of many kinds of events . . .

*What are some of these kinds of events?*

In handling the exception, the kernel might decide to give process 2 some time to run. (Suppose process 2 was running in the recent past.)

*If that happens, what must the kernel do to make the switch from process 1 to process 2 work correctly?*
Context switch from Process 1 to Process 2

- Kernel copies register values—PC, GPRs, FPRs, etc.—of process 1 to kernel data memory.
- Kernel copies previously-saved register values for process 2 from kernel data memory back to registers.
- Kernel ensures that VPN-to-PPN translation will work correctly for process 2.
- The kernel starts a countdown timer for a timeslice for process 2.
- Kernel lets process 2 resume execution.

(FPR stands for floating-point register. FPRs are coming up a bit later in the course.)
Ensuring correct VPN-to-PPN translation after a context switch

A simple way: The kernel uses a special TLB instruction to invalidate all TLB entries.

Suppose process 1 is suspended and process 2 resumes execution. There will be a lot of TLB misses, which will bring valid translations for process 2 into the I-TLB and D-TLB.

This method would work for a TLB like the one in the middle of page 1 of the “TLB Organization” lecture document.
This is the diagram from the middle of page 1 of the “TLB organization” document. In this TLB, a TLB entry has **no information** about which process the translation belongs to.

<table>
<thead>
<tr>
<th>valid</th>
<th>dirty</th>
<th>reference</th>
<th>write access</th>
<th>tag (VPN)</th>
<th>physical page number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0x7ffffff</td>
<td>0x8032b</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0x10001</td>
<td>0x81997</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0x10003</td>
<td>0x80653</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0x7ffffff</td>
<td>0x80471</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0x7ffe</td>
<td>0x80100</td>
</tr>
</tbody>
</table>
Ensuring correct VPN-to-PPN translation after a context switch, continued

A more complex method that could offer better performance: See the TLB in shown at the bottom of page 1 of the “TLB Organization” lecture document.

Each process gets its own address-space identification (ASID) number, so valid translations for two or more processes can be in the TLB at the same time.
This is the diagram from the bottom of page 1 of the “TLB organization” document. In this TLB, each TLB entry has an **ASID field** that says which process the translation belongs to.

<table>
<thead>
<tr>
<th>valid</th>
<th>dirty</th>
<th>reference</th>
<th>write access</th>
<th>address space ID</th>
<th>tag (VPN)</th>
<th>physical page number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0x0042</td>
<td>0x7fffffff</td>
<td>0x8032b</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0x0042</td>
<td>0x10001</td>
<td>0x81997</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0x0042</td>
<td>0x10003</td>
<td>0x80653</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0x005b</td>
<td>0x7fffffff</td>
<td>0x80471</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x005b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x0042</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0x0042</td>
<td>0x7fffffff</td>
<td>0x80100</td>
</tr>
</tbody>
</table>

**ASID register** 0x0042
Page faults and page replacement policies

To handle a page fault, the kernel must copy a page from disk to a physical memory page.

Suppose physical memory is not close to full—many physical pages are not currently in use. How should the kernel choose the physical page that will receive the page from disk?

If physical memory is close to full—very few physical pages are not currently in use. What does the kernel have to do?
Computer with virtual memory and two levels of cache: Possible outcomes when a process tries to read memory . . .

Start

**TLB hit?**

- **yes**
  - **L1 cache hit?**
    - **yes** outcome A
    - **no** outcome D
  - **no** outcome B

- **no**
  - **page table hit?**
    - **yes** outcome E
    - **no** outcome F

**L2 cache hit?**

- **yes** outcome C
- **no**
ENCM 369 Memory System Summary

To applications programmers, memory may seem to be a simple large array of words.

To operating system programmers and hardware designers, memory is a complicated, multi-level system.

Important: Applications programmers should be aware of caches and virtual memory, because caches and VM can be deciding factors in whether performance of applications is acceptable.
Outline of Slide Set 10 for Lecture Section 01

Introduction to Virtual Memory

Pages and Address Translation

TLBs and TLB miss handling

Memory System Performance Example
“Memory Performance Example” lecture document

Version 1

```c
int max_element(int a[][16000],
    int nrow)
{
    result = a[0][0];
    for (i = 0; i < nrow; i++)
        for (j = 0; j < 16000; j++)
            if (a[i][j] > result)
                result = a[i][j];
    return result;
}
```

Version 2

```c
int max_element(int a[][16000],
    int nrow)
{
    result = a[0][0];
    for (j = 0; j < 16000; j++)
        for (i = 0; i < nrow; i++)
            if (a[i][j] > result)
                result = a[i][j];
    return result;
}
```

Both versions were tested with \texttt{nrow = 16000} — total number of memory accesses within the \texttt{for} loops was $16000^2 = 256$ million for each function. \textit{What were the run times for each version?}
Memory system performance example, continued

Version 2 took more than 45 times as long as Version 1 to do the same work. What can explain that?

Some relevant numbers . . .

- size of 2-D array:
  4 bytes/int × 256 × 10^6 ints = 1024 × 10^6 bytes;

- cache capacity: 6 MB = 6 × 2^{20} bytes — much less than total memory use;

- cache block size:
  64 bytes = 64 bytes / 4 bytes per int = 16 ints

- number of virtual memory pages used for array:
  1024 × 10^6 bytes / 4096 bytes per page = 250,000 pages — a much larger number than the number of translations that can fit in a TLB.
Let’s scale the problem down to a 3-row, 4-column array.

Numbers within the words show **order of access** to words by the two versions of the `max_element` function.

Now imagine the order of access for each of the two versions when we scale back up to a 16000 × 16000 array.

```c
int x[3][4];
x is an array of 3 arrays of 4 ints.
```

**Version 1 access is sequential.**

**Version 2 access is NOT sequential.**
The diagram shows allocation of words for
\[
\text{int } x[M][N];
\]
where \(M\) and \(N\) are constants.

\(x[i][j]\) and \(x[i][j+1]\) are adjacent in memory but \(x[i][j]\) and \(x[i+1][j]\) may be far apart from one another.

The rules are similar for 2-D arrays of other types, such as double.