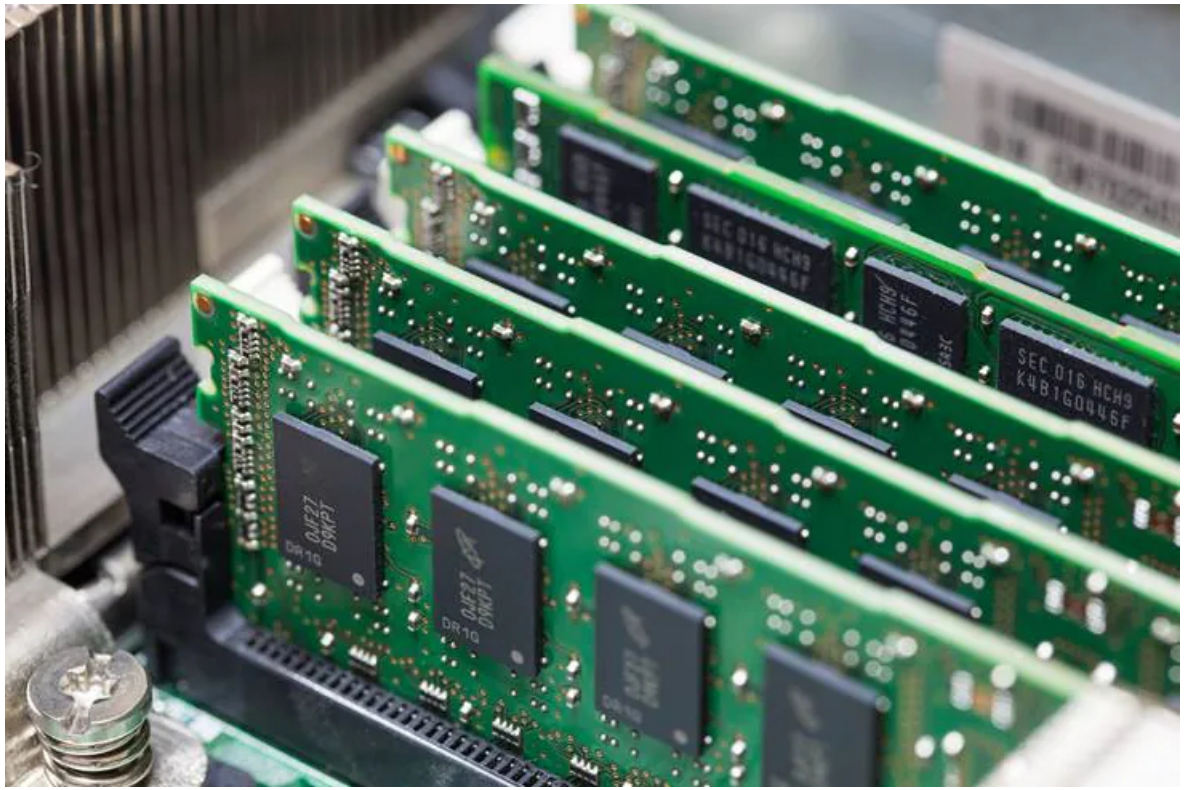
Lecture 6 — Memory

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

March 11, 2023

Main Memory



Fetch instructions from memory, fetch operands, write result.

… All to or from memory!

A single simple instruction like an addition could result in 4 memory accesses.

Therefore, we are spending a lot of time interacting with memory.

Memory as a Resource

Application developers behave as if:

(1) main memory is unlimited; and

(2) all of main memory is at the program's disposal.

Why do program developers pretend that memory is infinite and unshared?

Size Matters Not

Compared to the early days of computing, memory available is huge.

The Commodore 64, introduced in 1982, had a whopping 64 KB of memory.

A 4-byte (32-bit) integer was a significant fraction of memory.

Size Matters Not

Application developers had to scrimp and save to avoid wasting even a single integer's worth of memory.

This is why languages like C and Java support types like `short`.

Mr. Krabs when he loses a penny:

Size Matters Not

Now: if 1000 integers were wasted unnecessarily, would anyone notice or care?

“Kicking the can down the road”

Even though memory might be big enough for every process to have its own area, that would not work if every developer assumes memory is unshared.

The OS as Manager

Modern operating systems manage the shared resource of memory for them.

This was not always the case; applications used to be responsible for it.

There were third party programs to let the user do some, too.

The Empire is on the Verge of Success

https://www.youtube.com/watch?v=KMFycV3SS9o

No Memory Management

The simplest way to manage memory is, well, not to manage memory at all.

Early mainframe computers and even personal computers into the 1980s had no memory management strategy.

Programs would just operate directly on memory addresses.

No Memory Management

The section of memory that is program-accessible depended a lot on the operating system, if any, and other things needed (e.g., the BASIC compiler).

So to write a program, we need to know the “start” & “end” address.

These would differ from machine to machine, making it that much harder to write a program that ran on different computers.

No Memory Management - No Rules

A program executed an instruction directly on a memory address, such as writing 385 into memory location 1024.

Suppose you wanted to have two programs running at the same time.

If the first program writes to address 1024, and the second program writes to address 1024, the second program overwrote the first program's changes.

No Memory Management - No Rules

Alternatively, the first program can use memory locations less than, say, 2048 and the second uses memory locations above 2048.

This level of co-ordination gets more and more difficult as more and more programs are introduced to the system.

It is next to impossible if we do not control (have the source code to) all the programs that are to execute concurrently.

No Memory Management - A Solution

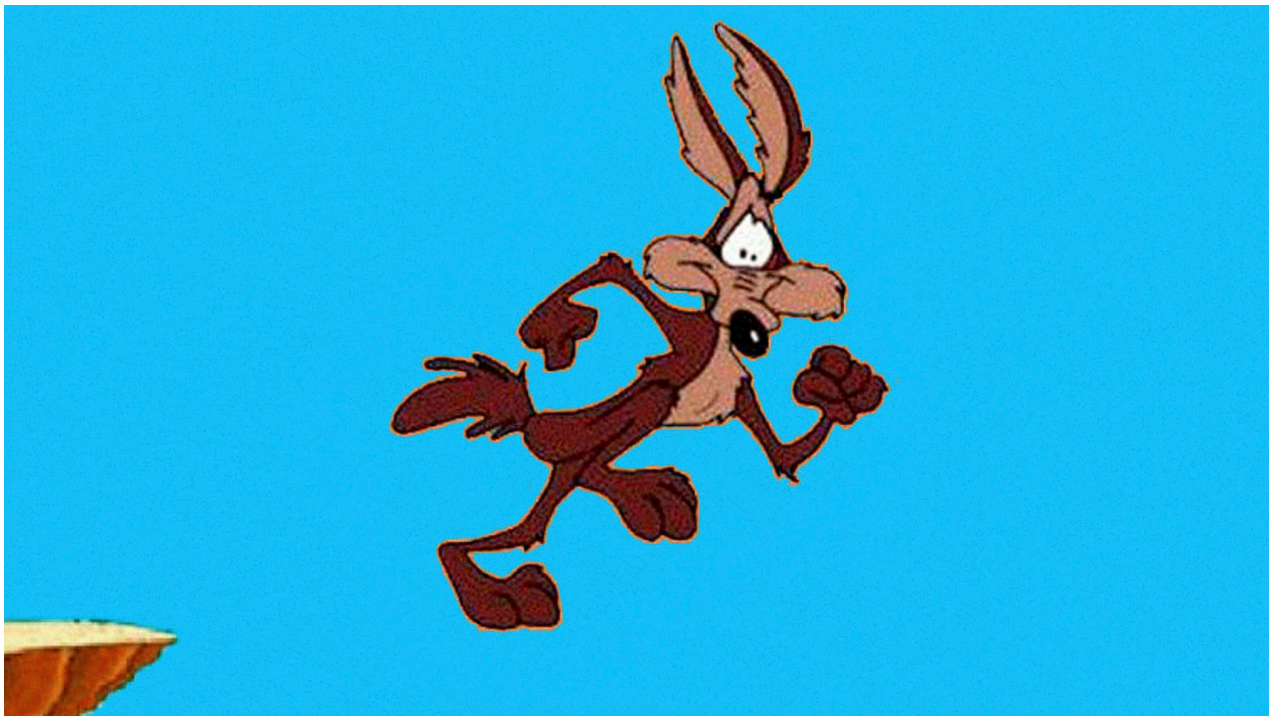
On every process switch, save the entire contents of memory to disk.
Then restore the memory contents of the next process to run.

This kind of swapping is, to say the least, incredibly expensive.

But the problem is avoided; only 1 process is ever in memory at a time.

Jumping without a Parachute

Another problem: there is no protection for the operating system.



Jumping without a Parachute

The operating system is typically placed in either low memory or high memory, or in some cases, a bit of both.

An errant memory access might overwrite a part of the OS in memory.

This can not only lead to crashes, but also corrupting important files on disk.

IBM 360 Solution

The IBM 360 solved this problem by dividing memory into 2 KB blocks and each was assigned a 4-bit protection key, held in special registers in the CPU.

The Program Status Word (PSW) also contained a 4 bit key.

The 360 hardware would then identify as an error an attempt to access memory with a protection code different from the PSW key.

The OS was the only software allowed to change the protection keys.

Thus, no program could interfere with another or with the operating system

Base and Limit

We can generalize this solution by having two values maintained:

The base and limit addresses.

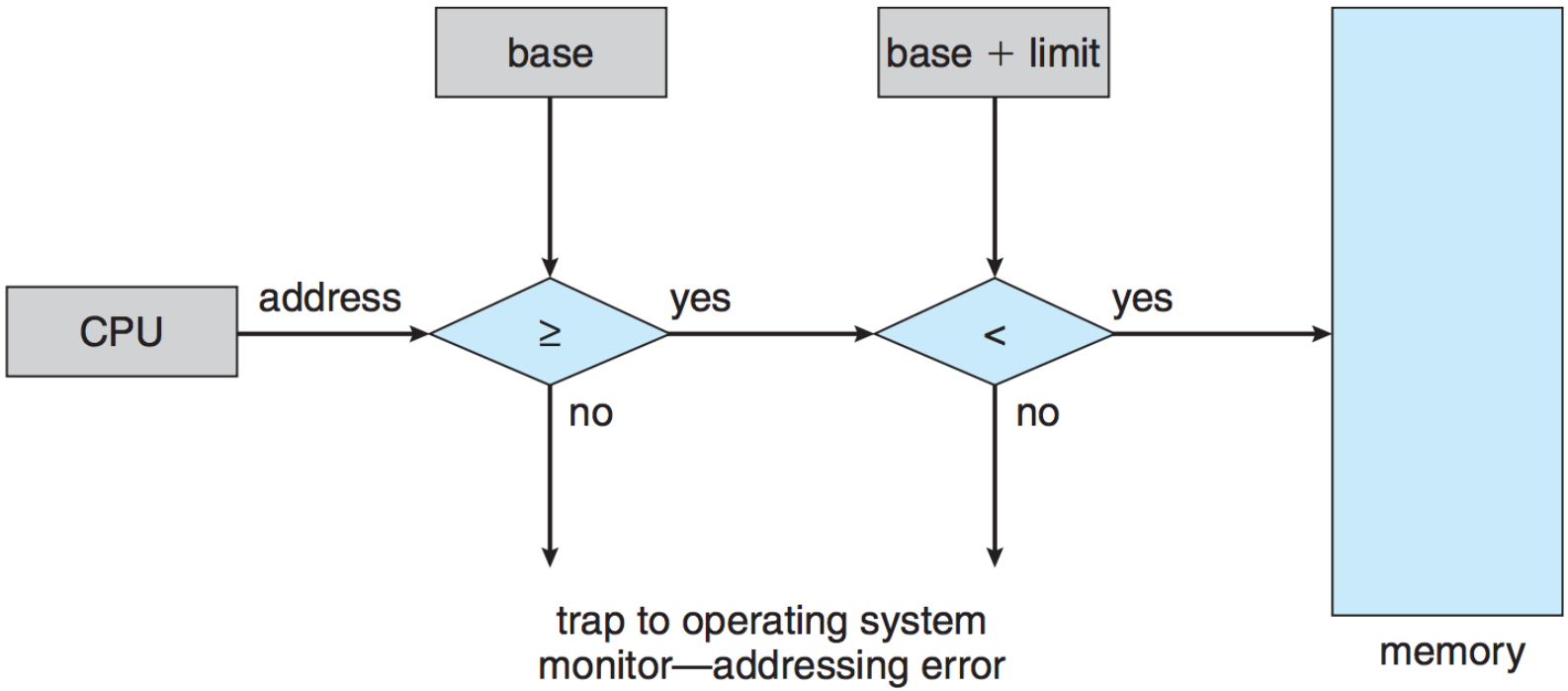
These define the start and end addresses of the program's memory.

Every memory access is compared to the base and the [base + limit] address.

If an attempted memory access falls outside that acceptable range: error.

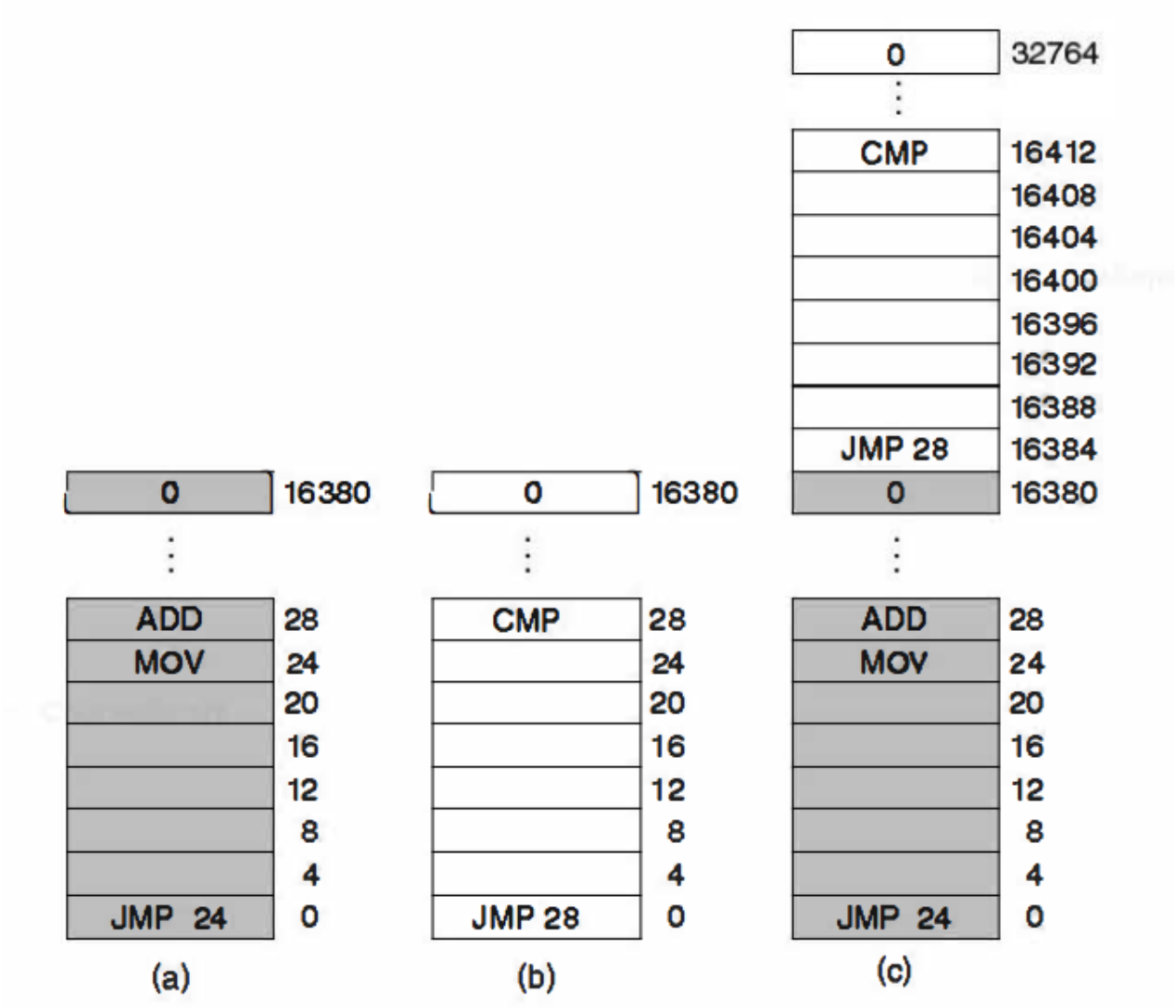
Base and Limit

The base & limit variables are often registers; comparison is done via hardware.



Base and Limit

Imagine we have two programs numbers simply 1 and 2, each 16 KB in size.
We will load them into memory in different consecutive areas:



Base and Limit Problem

Program 1 will execute as expected.

The problem is immediately obvious when Program 2 runs.

The instruction at address 16384 is executed: `JMP  28` takes execution to memory address 28 and not the expected address of 16412.

The problem is that both programs reference absolute physical locations.

Static Relocation

The IBM 360's stopgap solution to this was to do static relocation.

If a program was being loaded to a base address 16384 the constant 16384 was added to every program address during the load process.

While slow, if every address is updated correctly, the program works.

Static Relocation is Hard

A command like `JMP 28` must be relocated, but the 28 in a command like `ADD R1,28` is a constant and should not be changed.

How do we know which is an address and which is a constant?

It gets worse: in C a pointer contains an address, but addresses are just numbers.

So in theory we could just dereference any integer variable and it would take us to a memory address (whether it's valid or not is not the point here).

That makes it even harder to know if a number is just a number or an address.

Variable Names

When we are writing a program, unless it's in assembly, we do not usually refer to variables by their memory locations.

The command we write looks something like `x = 5;`

We know that variable x is stored in memory, the question arises: when is the variable assigned a location in memory?

Assigning a Variable a Location

There are three opportunities to assign a variable a place in memory:

1. **Compile time**
2. **Load time**
3. **Execution time**

Address Space

It is clear that having no memory management leaves us with a number of problems.

We need an abstraction; a layer of indirection.

We do this with a concept called address space.

Address Space

An address space is a set of addresses that a process can use.

Each process has its own address space, independent of other processes'.
    Except when we create shared memory.

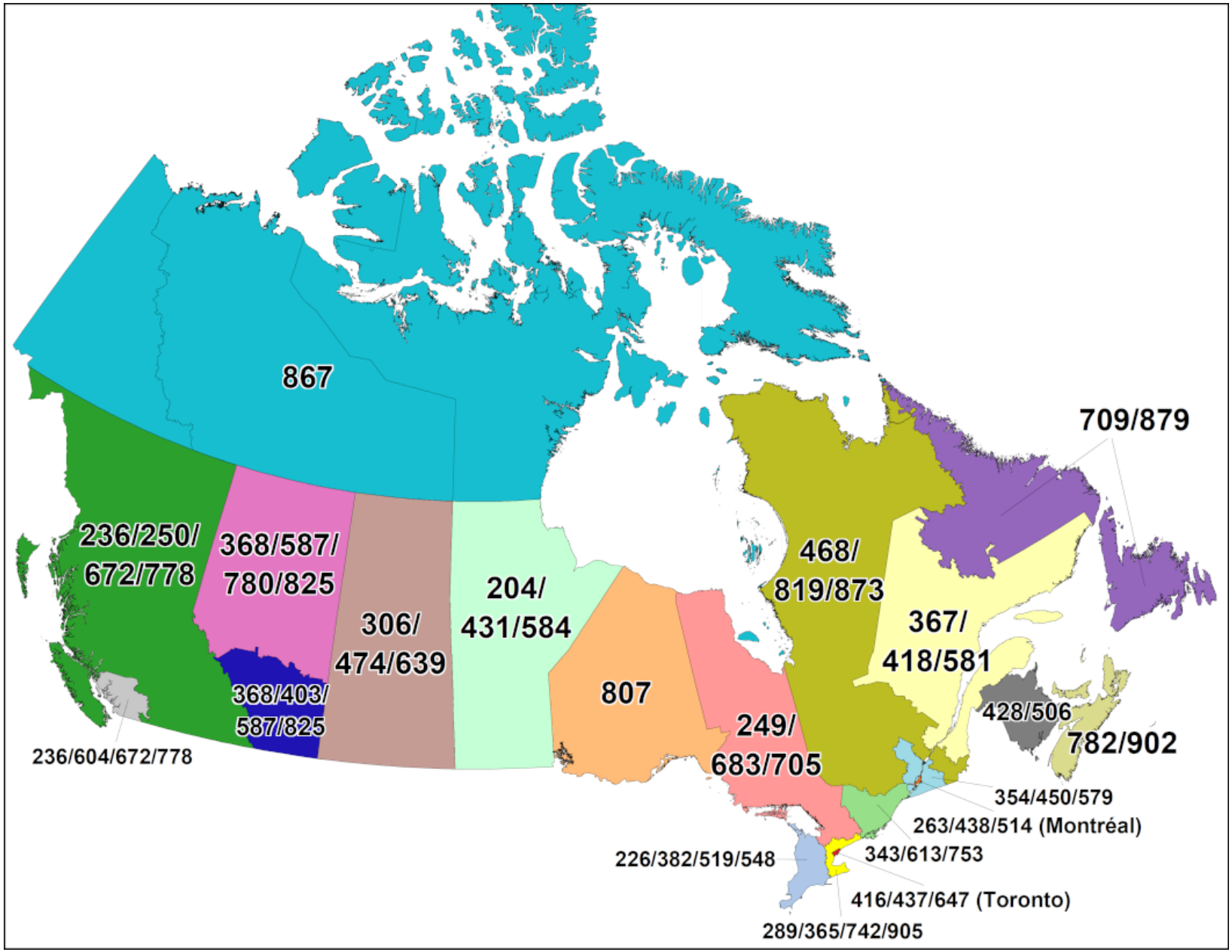
Address Space: Telephone Numbers

Telephone numbers in Canada and the USA take the form of NNN-NNNN.

In theory, any number in the range 000-0000 to 999-9999 could be issued, but in practice certain numbers are reserved (like the 000 or 555 prefixes).

Given the number of phones in the countries, 7 digits can't possibly be enough.

Address Space: Telephone Numbers



Area Codes for Memory

This is the idea we want to apply: let each process have its own area code.

So process can write to location 1024 and process 2 can write to location 1024 and these are two distinct locations, perhaps 21024 and 91024 respectively.

Now, instead of altering the addresses in memory, we will effectively prefix every memory access with an area code.

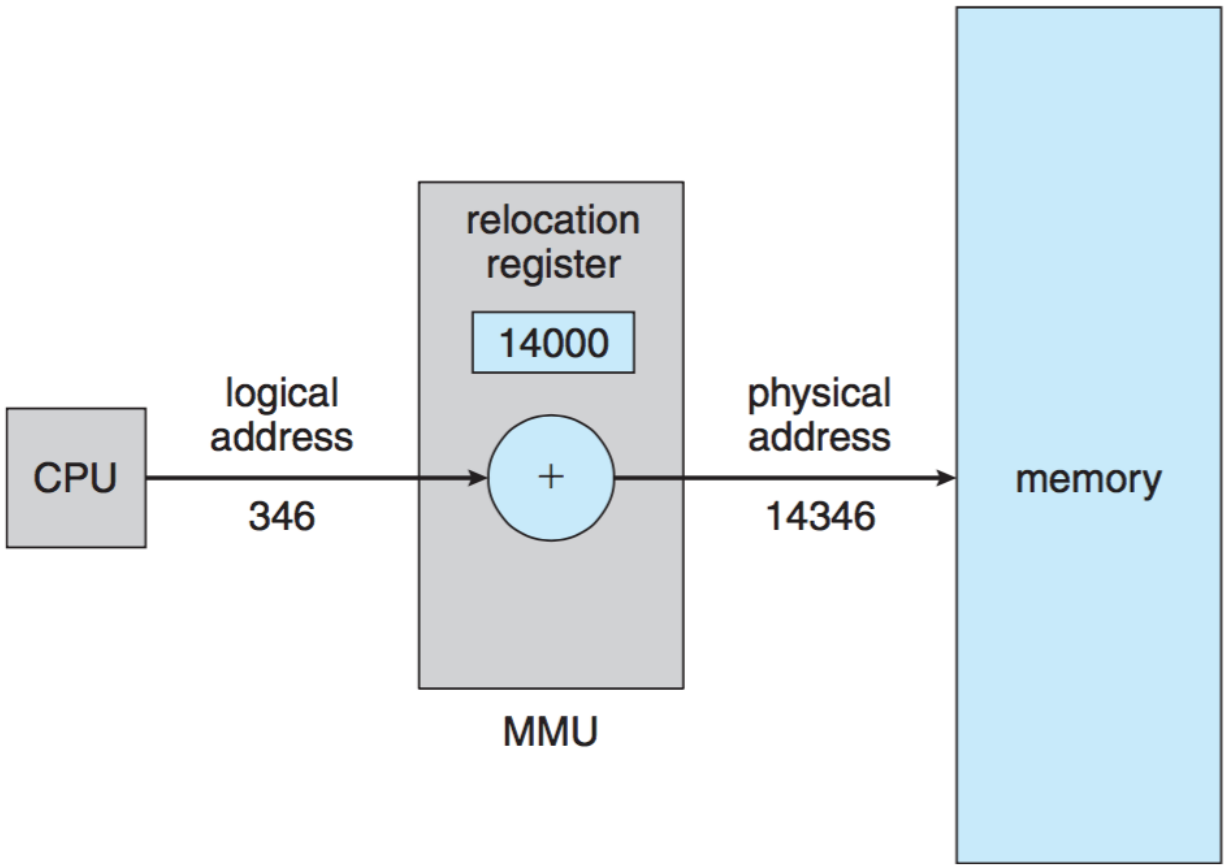
Area Codes for Memory

The address that is generated by the CPU, e.g., the 28 in `JMP 28`, is the logical address.

We then add the area code to it to produce the physical address (the actual location in memory and the address that it sent over the bus).

Relocation Register

In practice, to speed this up, it is done via some hardware, and the “area code” is a register called the relocation register:



Relocation Register

The process itself does not know the physical address (14346 in the example).

It knows only the logical address (346).

This is a run-time mapping of variables to memory.

We get some protection between processes.

We would get more protection if we brought back the limit register and compared the physical address to the base and [base + limit] values again.

The Other Kind of Relocation

This scheme also gives us something new: we can relocate a process in memory if we change the relocation register's value accordingly.

A process that is currently loaded into memory with a relocation register value of 14000 can easily be moved to another location.

Copy all the memory from relocation register to the limit to a new location and then update the relocation register to the new starting location (90000).

After that, the old location of the process's memory can be marked as available or used by another process.

The Price We Pay

Every memory access now includes an addition (or 2 if the limit register is used).

Comparisons are pretty quick for the CPU, but addition can be quite a bit slower, because of carry propagation time.

So every memory access has a penalty associated with it:

The addition of the relocation register value to the issued CPU address.

Swapping

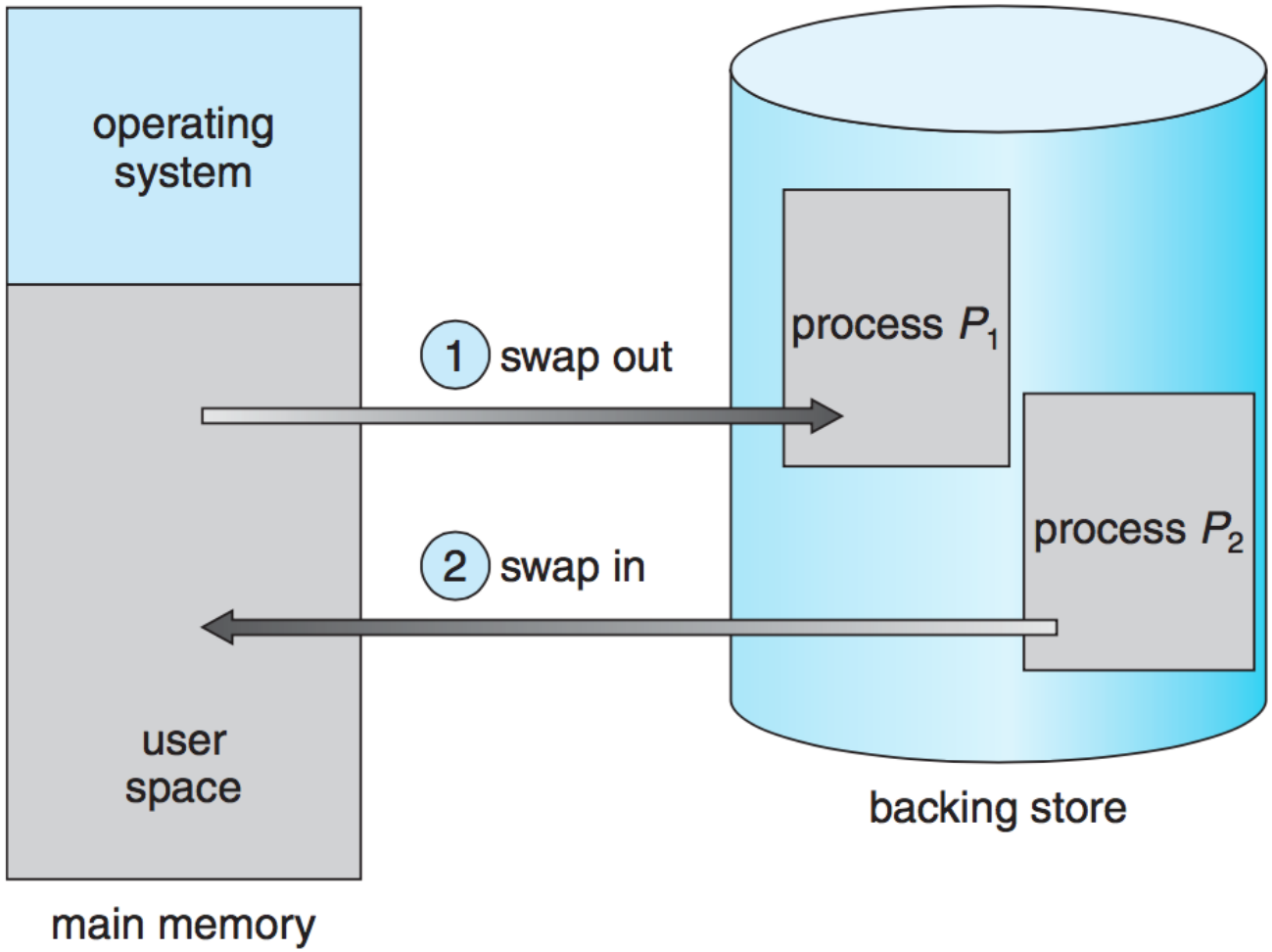
To run, a process must be in main memory.

Given enough processes, sufficiently demanding processes, it will not be possible to keep all of them in memory at the same time.

Processes that are blocked may be taking up space in memory. It might be logical to make room for processes that are ready to run.

Moving a process from memory to disk (or vice-versa) is called swapping.

Swapping


operating system
process P_1
1 swap out
process P_2
2 swap in
user space
backing store
main memory

Swapping is Painful

Unfortunately, swapping a process to disk is very painful.

If the process is using 1 GB of memory, to swap a process out to disk, we need to write 1 GB of memory to disk.

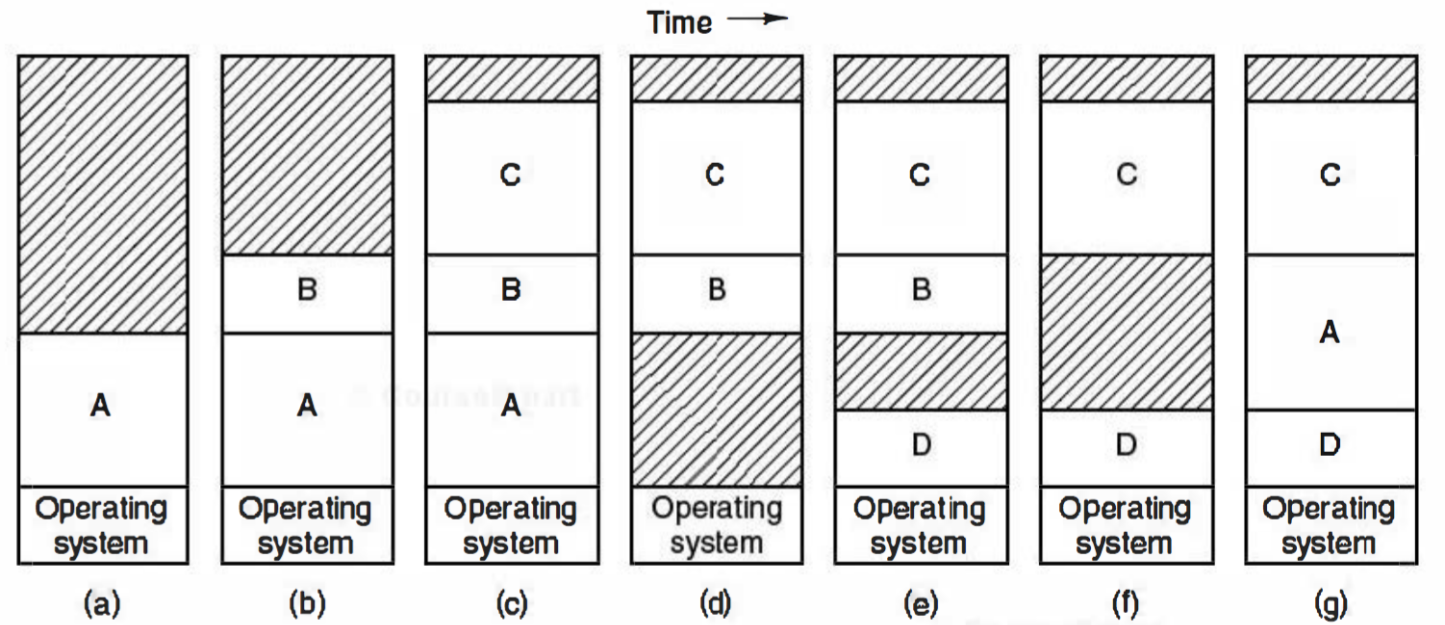
To load that process back later, it means reading another 1 GB from disk and putting it into main memory.

So, swapping is something we would like to do as little as possible.

Swapping and Relocation

When swapping a process back in from disk it is not necessary to put it back in exactly the same place as it originally was.

Just update the relocation register.



Memory Allocation

Thus far we have considered process memory as a large, fixed-sized block.

A process gets a big section of memory and operates in that area.

As you know from previous programming experience, use of the `new` keyword in some languages, or `malloc()` in C, will result in dynamic memory allocation.

We will have to deal with this next…