C++ Misc, Buffering & Virtual Memory

Computer Systems Programming, Spring 2025

Instructor: Travis McGaha

Teaching Assistants:

Andrew Lukashchuk
Angie Cao
Aniket Ghorpade
Ashwin Alaparthi
Austin Lin
Hassan Rizwan
Lobi Zhao
Pearl Liu
Perrie Quek

pollev.com/tqm

- Do you usually eat breakfast?

Administrivia

- HW06 – Hash Table
  - Posted☺
  - Due Friday 3/21 at midnight, leaving open till Sunday night tho
  - AG posted

- Mid-semester Survey Posted!
  - Due Sunday 3/23 & Anonymous
  - Please give feedback, it is useful for me to make the course better!
    And a lot has changed this semester!

Lecture Outline

- **C++ Misc**
- Locality & Buffering again
- Virtual Memory

Lecture Outline

- **C++ Misc**
  - **Refresh cont.**
  - Assignment operator
  - Initializer List
  - Casting
- Locality & Buffering again
- Virtual Memory

Poll Everywhere

pollev.com/tqm

- Final output of this code?

```
int& stuff(int& x, int y) {
  int& z = y;
  z = 12;
  x += 3;
  return x;
}

int main() {
  int a = 1;
  int b = 2;

  int& c = stuff(a, b);
  c++;

  cout << a << endl;
  cout << b << endl;
  cout << c << endl;
}
```

Poll Everywhere

pollev.com/tqm

- How many times does a **`string`** constructor get invoked here?

```
int main() {
  string a("hello");
  string b("like");
  string* c = new string("antennas");
}
```

Poll Everywhere

pollev.com/tqm

- How many times does the **`string`** destructor get invoked here?

```
int main() {
  string a("hello");
  string b("like");
  string* c = new string("antennas");
}
```

Lecture Outline

- **C++ Misc**
  - Refresh cont.
  - **Assignment operator**
  - Initializer List
  - Casting
- Locality & Buffering again
- Virtual Memory

Assignment != Construction

- "=" is the assignment operator
  - Assigns values to an *existing, already constructed* object

```
Point w;          // default ctor
Point x(1, 2);    // two-ints-argument ctor
Point y(x);       // copy ctor
Point z = w;      // copy ctor
y = x;            // assignment operator
```

Method `operator=()`

equivalent code:
`y.operator=(x);`

Overloading the "=" Operator

- You can choose to define the "=" operator
  - But there are some rules you should follow:

```
Point& Point::operator=(const Point& rhs) {
  if (this != &rhs) {  // (1) always check against this
    x_ = rhs.x_;
    y_ = rhs.y_;
  }
  return *this;        // (2) always return *this from op=
}

Point a;          // default constructor
a = b = c;        // works because = return *this
a = (b = c);      // equiv. to above (= is right-associative)
(a = b) = c;      // "works" because = returns a non-const
                  // reference to *this
```

More important when data members are Dynamic memory

Should be a reference to *this to allow chaining

Explicit equivalent:

```
a.operator=(b.operator=(c));
```

Synthesized Assignment Operator

- If you don't define the assignment operator, C++ will synthesize one for you
  - It will do a *shallow* copy of all of the fields (*i.e.* member variables) of your class
  - Sometimes the right thing; sometimes the wrong thing

Usually wrong whenever a class has dynamically allocated data

```
#include "SimplePoint.h"

... // definitions for Distance() and SetLocation()

int main(int argc, char** argv) {
  SimplePoint x;
  SimplePoint y(x);
  y = x;            // invokes synthesized assignment operator
  return EXIT_SUCCESS;
}
```

Lecture Outline

- **C++ Misc**
  - Refresh cont.
  - Assignment operator
  - **Initializer List**
  - Casting
- Locality & Buffering again
- Virtual Memory

Initialization Lists

- C++ lets you *optionally* declare an initialization list as part of a constructor definition
  - Initializes fields according to parameters in the list
  - The following two are (nearly) identical:

```
Point::Point(int x, int y) {
  x_ = x;
  y_ = y;
}
```

```
// constructor with an initialization list
Point::Point(int x, int y) : x_(x), y_(y) {

}
```

data member name

Expression

Body can be empty

Initialization vs. Construction

```
class Point3D {
 public:
  // constructor with 3 int arguments
  Point3D(int x, int y, int z) : y_(y), x_(x) {
    z_ = z;
  }
 private:
  int x_, y_, z_;  // data members
};  // class Point3D
```

First, initialization list is applied.

2) set y_

1) set x_

3) set z_ (garbage)

4) set z_

Next, constructor body is executed.

- Data members in initializer list are initialized in the order they are defined in the class, not by the initialization list ordering (!)
  - Data members that don't appear in the initialization list are *default initialized/constructed* before body is executed

- Initialization preferred to assignment to avoid extra steps
  - Real code should never mix the two styles

Example WITHOUT Initializer list

- Not using an initializer list in a constructor is like separately declaring a variable and initializing it

```
class Point {
 public:
  // constructor with 3 int arguments
  Point3D(int x, int y, int z) {
    x_ = x;
    y_ = y;
    z_ = z;
  }

 private:
  int x_, y_, z_;  // data members
};  // class Point
```

Sort of translates to…

```
Point::Point(int x, int y) {
  int x_;
  int y_;
  int z_;
  x_ = x;
  y_ = y;
  z_ = z;
}
```

Example WITH Initializer list

- Not using an initializer list in a constructor is like separately declaring a variable and initializing it

```
class Point {
 public:
  // constructor with 3 int arguments
  Point3D(int x, int y, int z) : x_(x), y_(y), z_(z) {
  }

 private:
  int x_, y_, z_;  // data members
};  // class Point
```

Sort of translates to…

```
Point::Point(int x, int y) {
  int x_ = x;
  int y_ = y;
  int z_ = z;
}
```

Example WITHOUT Initializer list

- If we don't use an initializer list for more complex types…
  Then this results in unnecessary default construction of fields.
  - Here, the string name_ has two dynamic allocations

```
class Song {
 public:
  // constructor with 3 int arguments
  Song(string name, int rating) {
    name_ = name;
    rating_ = rating;
  }

 private:
  string name_;
  int rating_;  // data members
};  // class Song
```

Sort of translates to…

```
Song::Song(string name, int rating) {
  string name_;   ← Default constructed
  int rating_;
  name_ = name;   ← assigned
  rating_ = rating;
}
```

Poll Everywhere

pollev.com/tqm

- What does this code do?
  Does it work as intended?

```
int main() {
  int x = 5;
  int& y = x;
  int& z;

  y += 2;
  z = x;

  z += 1;

  cout << x << endl;
}
```

Example WITHOUT Initializer list

- If we don't use an initializer list for reference data members, it just doesn't work

```
class Song {
 public:
  // constructor with 3 int arguments
  Song(string name, int rating) {
    name_ = name;
    rating_ = rating;
  }

 private:
  string name_;
  int& rating_;  // data members
};  // class Song
```

Sort of translates to…

```
Song::Song(string name, int rating) {
  string name_;
  int& rating_;
  name_ = name;
  rating_ = rating;
}
```

References need to be initialized!

Example WITH Initializer list

- Use an initializer list ☺

```
class Song {
 public:
  // constructor with 3 int arguments
  Song(string name, int rating) :
    name_(name), rating_(rating) {}

 private:
  string name_;
  int& rating_;  // data members
};  // class Song
```

Sort of translates to…

```
Song::Song(string name, int rating) {
  string name_ = name;
  int& rating_ = rating;
}
```

Lecture Outline

- **C++ Misc**
  - Refresh cont.
  - Assignment operator
  - Initializer List
  - **Casting**
- Locality & Buffering again
- Virtual Memory

Explicit Casting in C

- Simple syntax: `lhs = (new_type) rhs;`
- Used to:
  - Convert between pointers of arbitrary type
    - Doesn't change the data, but treats it differently
  - Forcibly convert a primitive type to another
    - Actually changes the representation

(void*) my_ptr

(double) my_int

- You *can* still use C-style casting in C++, but sometimes the intent is not clear

Casting in C++

- C++ provides an alternative casting style that is more informative:
  - `static_cast<to_type>(expression)`
  - `dynamic_cast<to_type>(expression)`
  - `const_cast<to_type>(expression)`
  - `reinterpret_cast<to_type>(expression)`

- Always use these in C++ code
  - Intent is clearer
  - Easier to find in code via searching

static_cast

Any well-defined conversion

- `static_cast` can convert:
  - casting `void*` to `T*`
  - Non-pointer conversion
    - *e.g.* `float` to `int`
- If you are doing a cast not related to object inheritance, it will most likely be this one.

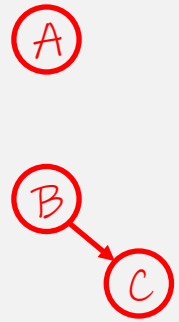
```
void foo() {
  int b = 3;
  float c;

  c = static_cast<float>(b);
}
```

static_cast

Any well-defined conversion

- `static_cast` can convert:
  - Pointers to classes **of related type**
    - Compiler error if classes are not related
    - Dangerous to cast *down* a class hierarchy

```
class A {
 public:
  int x;
};

class B {
 public:
  float y;
};

class C : public B {
 public:
  char z;
};
```




```
void foo() {
  B b; C c;

  // compiler error Unrelated types
  A* aptr = static_cast<A*>(&b);
  // OK  Would have worked without cast
  B* bptr = static_cast<B*>(&c);
  // compiles, but dangerous
  C* cptr = static_cast<C*>(&b);
}   What happens when you do cptr->z?
```

dynamic_cast

- `dynamic_cast` can convert:
  - Pointers to classes **of related type**
  - References to classes **of related type**
- `dynamic_cast` is checked at both compile time and run time
  - Casts between unrelated classes fail at compile time
  - Casts from base to derived fail at run time if the pointed-to object is not the derived type
- Can be used like `instanceof` from java

dynamiccast.cc

```
class Base {
 public:
  virtual void foo() { }
  float x;
};

class Der1 : public Base {
 public:
  char x;
};
```

```
void bar() {
  Base b; Der1 d;

  // OK (run-time check passes)
  Base* bptr = dynamic_cast<Base*>(&d);
  assert(bptr != nullptr);

  // OK (run-time check passes)
  Der1* dptr = dynamic_cast<Der1*>(bptr);
  assert(dptr != nullptr);

  // Run-time check fails, returns nullptr
  bptr = &b;
  dptr = dynamic_cast<Der1*>(bptr);
  assert(dptr != nullptr);
}
```

const_cast

- const_cast adds or strips const-ness
  - Dangerous (!)

```
void foo(int* x) {
  *x++;
}

void bar(const int* x) {
  foo(x);                          // compiler error
  foo(const_cast<int*>(x));        // succeeds
}

int main(int argc, char** argv) {
  int x = 7;
  bar(&x);
  return EXIT_SUCCESS;
}
```

reinterpret_cast

- `reinterpret_cast` casts between *incompatible* types
  - Low-level reinterpretation of the bit pattern
  - *e.g.* storing a pointer in an `int`, or vice-versa
    - Works as long as the integral type is "wide" enough
  - Converting between incompatible pointers
    - Dangerous (!)
  - Use any other C++ cast if you can.

  - You may find it useful in HW3 (which is posted today)

Lecture Outline

- C++ Misc
- **Locality & Buffering again**
- Virtual Memory

Everything is Bytes

- In our computers, everything is stored as bits and bytes. We can read/write things other than characters. We just need to tell how many bytes to read

- Read an integer:

```
int fd = open(...);
int x;
read(fd, &x, sizeof(x));
```

- Write a struct:

```
struct Point {
  float x, y;
};

Point p{3.0F, 2.0F};
write(fd, &p, sizeof(p));
```

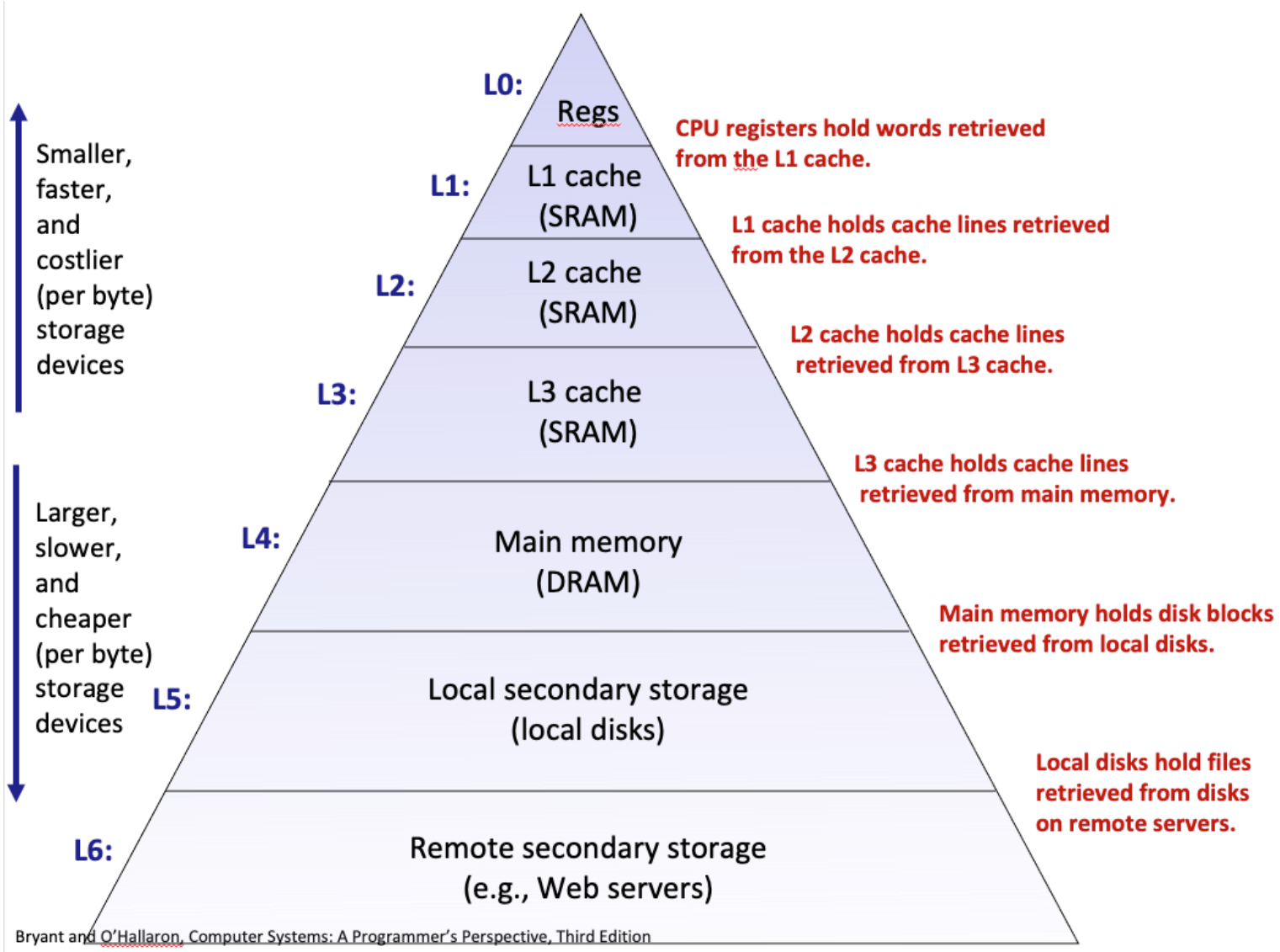
- Read a string? Why doesn't this work

```
string x;
read(fd, &x, sizeof(x));
```

Lecture Outline

- C++ Misc
- **Locality & Buffering again**
- Virtual Memory

Memory Hierarchy



C++ isotream vs POSIX

- C++ iostream: user level portable library for **i**nput/**o**utput streams. Should work on any environment that has the C++ standard library
  - E.g. cout, operator<<, endl, cin, operator>>, getline, etc.

- POSIX C API: **P**ortable **O**perating **S**ystem **I**nterface. Functions that are supported by many operating systems to support many OS-level concepts (Input/Output, networking, processes, pipes, threads…)

Buffered writing

- By default, C++ `iostream` usually uses buffering on top of POSIX:
  - When one writes with **`cout`**, the data being written is copied into a buffer allocated by `C++ iostream` inside your process' address space
  - As some point, once enough data has been written, the buffer will be "flushed" to the operating system.
    - When the buffer fills (often 1024 or 4096 bytes)
  - This prevents invoking the write system call and going to the filesystem too often

Buffered Writing Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  string msg {"hi"};
  std::ofstream fout("hi.txt");

  // read "hi" one char at a time
  fout.put(msg.at(0));

  fout.put(msg.at(1));


  return EXIT_SUCCESS;
}
```

buf

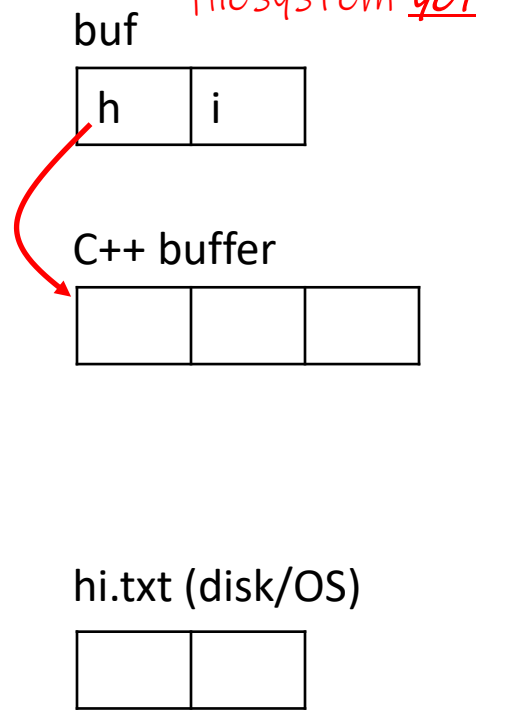
hi.txt (disk/OS)

Buffered Writing Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  string msg {"hi"};
  std::ofstream fout("hi.txt");

  // read "hi" one char at a time
  fout.put(msg.at(0));

  fout.put(msg.at(1));


  return EXIT_SUCCESS;
}
```

Store 'h' into buffer, so that we do not go to filesystem *yet*

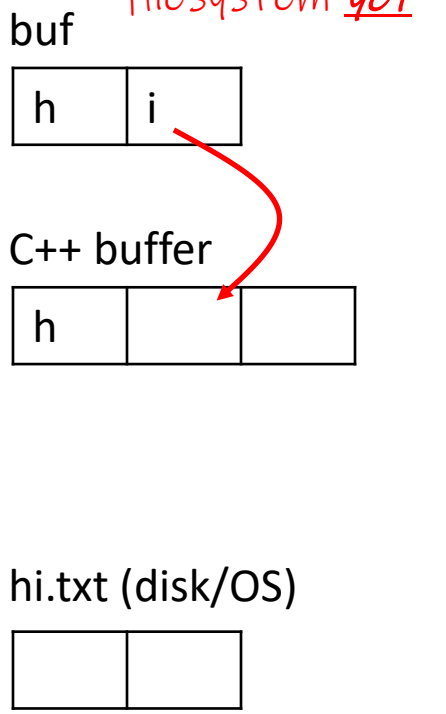


Buffered Writing Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  string msg {"hi"};
  std::ofstream fout("hi.txt");

  // read "hi" one char at a time
  fout.put(msg.at(0));

  fout.put(msg.at(1));


  return EXIT_SUCCESS;
}
```

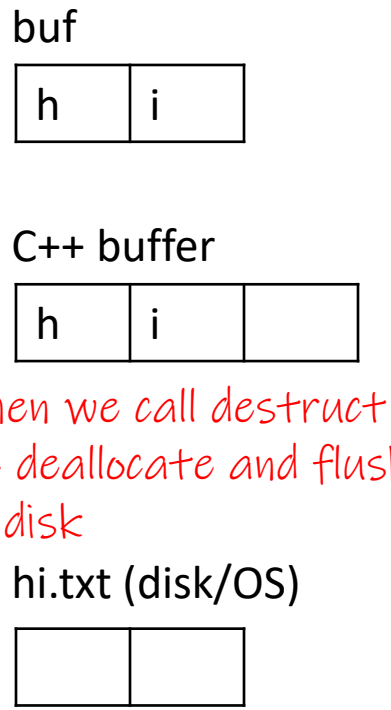
Store 'i' into buffer, so that we do not go to filesystem *yet*



Buffered Writing Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  string msg {"hi"};
  std::ofstream fout("hi.txt");

  // read "hi" one char at a time
  fout.put(msg.at(0));

  fout.put(msg.at(1));


  return EXIT_SUCCESS;
}
```



Buffered Writing Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  string msg {"hi"};
  std::ofstream fout("hi.txt");

  // read "hi" one char at a time
  fout.put(msg.at(0));

  fout.put(msg.at(1));


  return EXIT_SUCCESS;
}
```

buf

h	i

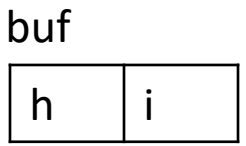
hi.txt (disk/OS)

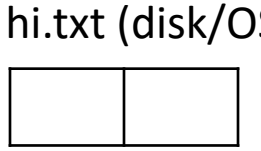
h	i

Unbuffered Writing Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  string msg {"hi"};
  int fd = open("hi.txt", O_WRONLY | O_CREAT);

  // read "hi" one char at a time
  write(fd, &(msg.at(0)), sizeof(char));

  write(fd, &(msg.at(1)), sizeof(char));

  close(fd);
  return EXIT_SUCCESS;
}
```

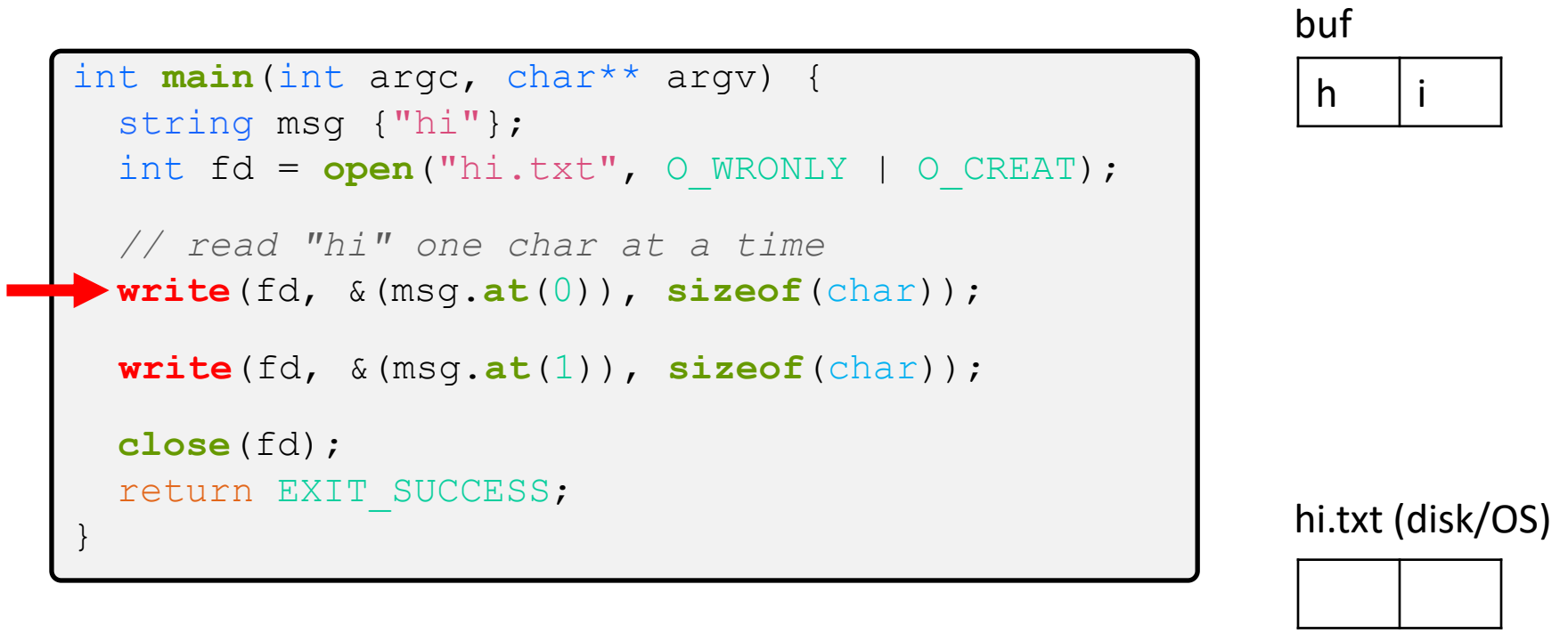




Unbuffered Writing Example

Arrow signifies what will be executed next

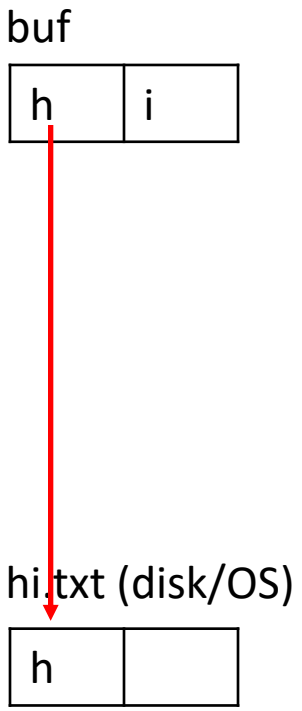
```
int main(int argc, char** argv) {
  string msg {"hi"};
  int fd = open("hi.txt", O_WRONLY | O_CREAT);

  // read "hi" one char at a time
  write(fd, &(msg.at(0)), sizeof(char));

  write(fd, &(msg.at(1)), sizeof(char));

  close(fd);
  return EXIT_SUCCESS;
}
```



Unbuffered Writing Example

Arrow signifies what will be executed next

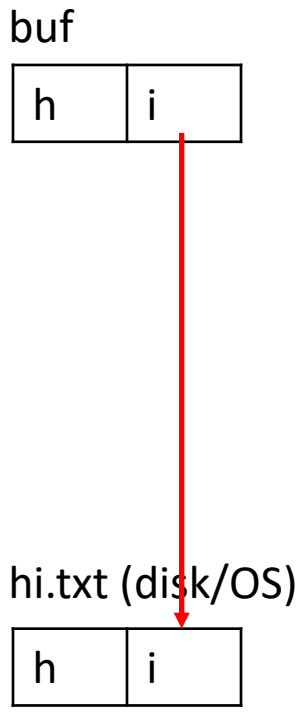
```
int main(int argc, char** argv) {
  string msg {"hi"};
  int fd = open("hi.txt", O_WRONLY | O_CREAT);

  // read "hi" one char at a time
  write(fd, &(msg.at(0)), sizeof(char));

  write(fd, &(msg.at(1)), sizeof(char));

  close(fd);
  return EXIT_SUCCESS;
}
```



Unbuffered Writing Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  string msg {"hi"};
  int fd = open("hi.txt", O_WRONLY | O_CREAT);

  // read "hi" one char at a time
  write(fd, &(msg.at(0)), sizeof(char));

  write(fd, &(msg.at(1)), sizeof(char));

  close(fd);
  return EXIT_SUCCESS;
}
```



Unbuffered Writing Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  string msg {"hi"};
  int fd = open("hi.txt", O_WRONLY | O_CREAT);

  // read "hi" one char at a time
  write(fd, &(msg.at(0)), sizeof(char));

  write(fd, &(msg.at(1)), sizeof(char));

  close(fd);
  return EXIT_SUCCESS;
}
```

buf

h	i

Two OS/File system accesses instead of one ☹

hi.txt (disk/OS)

h	i

Buffered Reading

- By default, C `stdio` uses buffering on top of POSIX:
  - When one reads with **`fread`**`()`, a lot of data is copied into a buffer allocated by `stdio` inside your process' address space
  - Next time you read data, it is retrieved from the buffer
    - This avoids having to invoke a system call again
  - As some point, the buffer will be "refreshed":
    - When you process everything in the buffer (often 1024 or 4096 bytes)
  - Similar thing happens when you write to a file

Buffered Reading Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  std::array<char, s> buf {};
  std::ifstream fin("hi.txt");

  // read "hi" one char at a time
  fout.get(arr.at(0));

  fout.get(arr.at(1));


  return EXIT_SUCCESS;
}
```

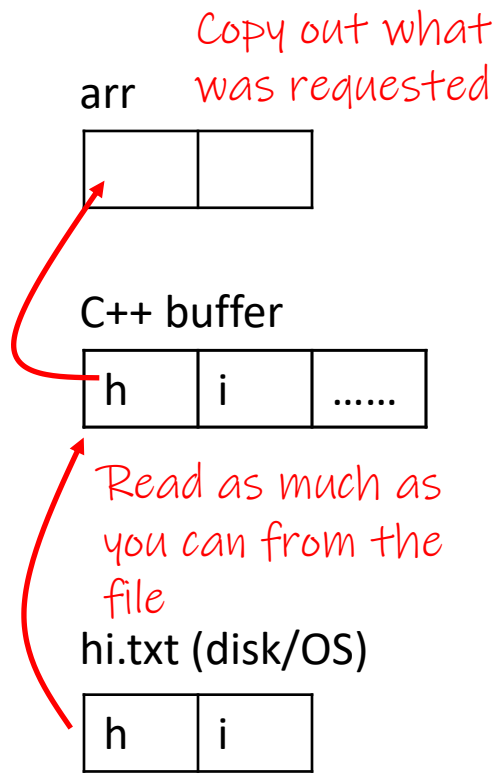
arr

hi.txt (disk/OS)

Buffered Reading Example

Arrow signifies what will be executed next

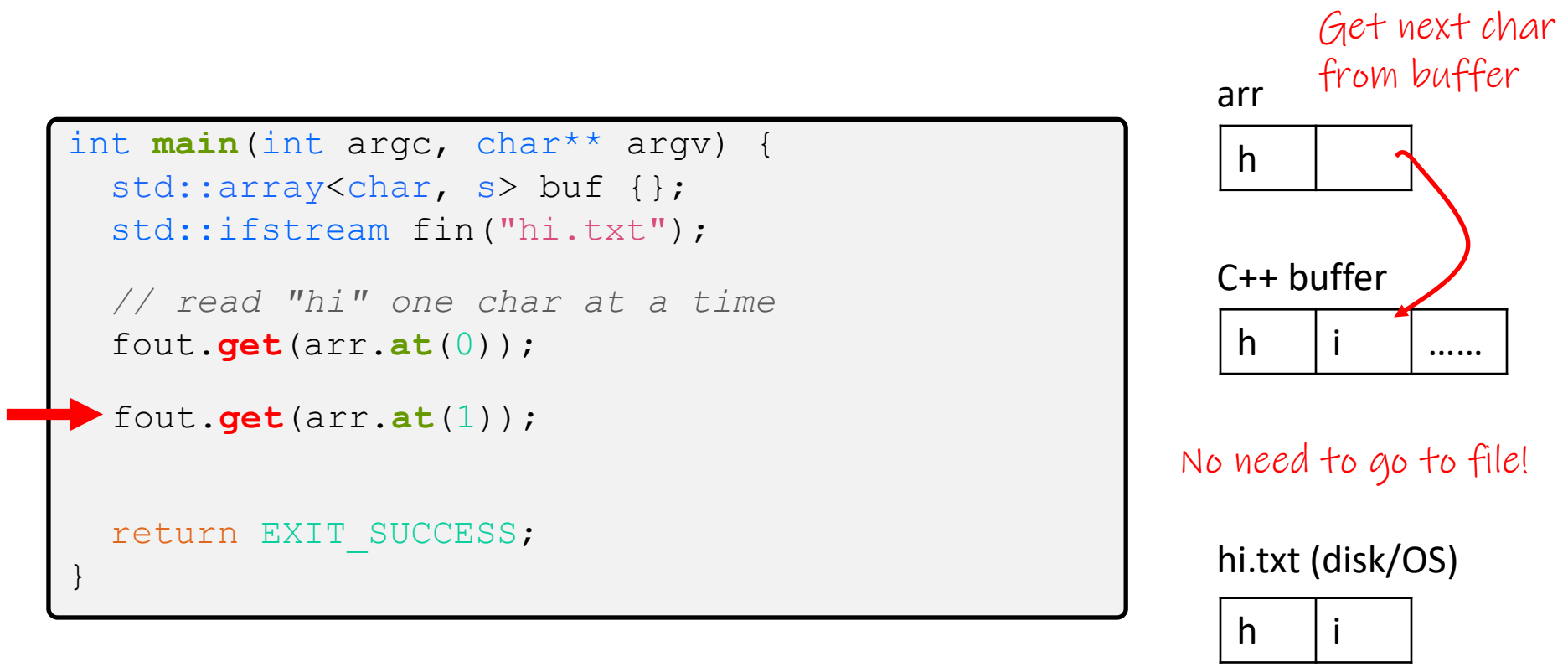
```
int main(int argc, char** argv) {
  std::array<char, s> buf {};
  std::ifstream fin("hi.txt");

  // read "hi" one char at a time
  fout.get(arr.at(0));

  fout.get(arr.at(1));


  return EXIT_SUCCESS;
}
```



Buffered Reading Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  std::array<char, s> buf {};
  std::ifstream fin("hi.txt");

  // read "hi" one char at a time
  fout.get(arr.at(0));

  fout.get(arr.at(1));


  return EXIT_SUCCESS;
}
```



Buffered Reading Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  std::array<char, s> buf {};
  std::ifstream fin("hi.txt");

  // read "hi" one char at a time
  fout.get(arr.at(0));

  fout.get(arr.at(1));


  return EXIT_SUCCESS;
}
```

arr

h	i

C++ buffer

h	i	.....

hi.txt (disk/OS)

h	i

Buffered Reading Example

Arrow signifies what will be executed next

```
int main(int argc, char** argv) {
  std::array<char, s> buf {};
  std::ifstream fin("hi.txt");

  // read "hi" one char at a time
  fout.get(arr.at(0));

  fout.get(arr.at(1));


  return EXIT_SUCCESS;
}
```

arr

h	i

hi.txt (disk/OS)

h	i

Buffering Details

- Buffering doesn't just mean we read the whole file in one go
  - We just read a large amount at a time to minimize trips to disk
  - We only read the full data in a file if the file happens to be smaller than the buffer
- **The key point is trying minimizing trips to the OS/File System**

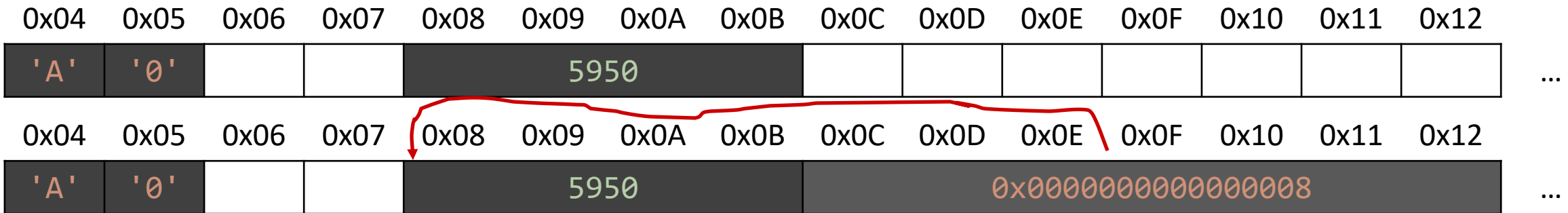
Lecture Outline

- C++ Misc
- Locality & Buffering again
- **Virtual Memory**
  - **High Level**
  - Page Replacement

Memory as an array of bytes

- Everything in memory is made of bits and bytes
  - Bits: a single 1 or 0
  - Byte: 8 bits
- Memory is a giant array of bytes where everything* is stored
  - Each byte has its own address (“index”)
- Some types take up one byte, others more

```
int main() {
  char c = 'A';
  char other = '0';
  int x = 5950;
  int* ptr = &x;
}
```



Poll Everywhere

pollev.com/tqm

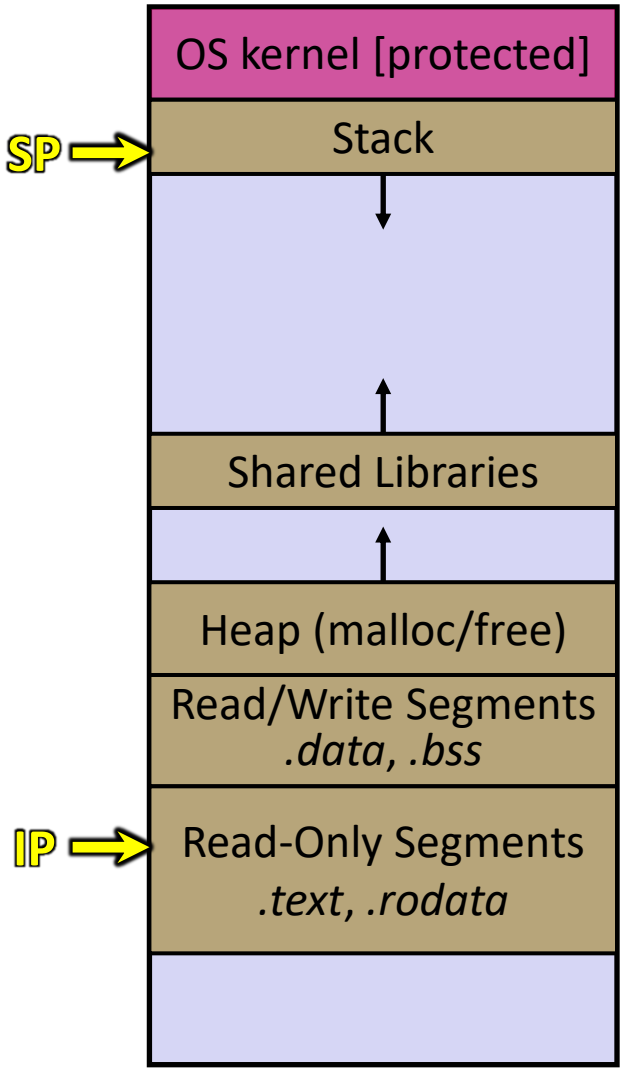
- What does this print for **`x`** and the **`ptr`**?

```
int main() {
  int x = 5;
  int* ptr = &x;
  pid_t pid = fork();

  if (pid == 0) {
    *ptr += 1;
    cout << x << endl;
    cout << ptr << endl;
    exit(EXIT_SUCCESS);
  }

  waitpid(pid, NULL, 0);
  *ptr += 1;
  cout << x << endl;
  cout << ptr << endl;
}
```

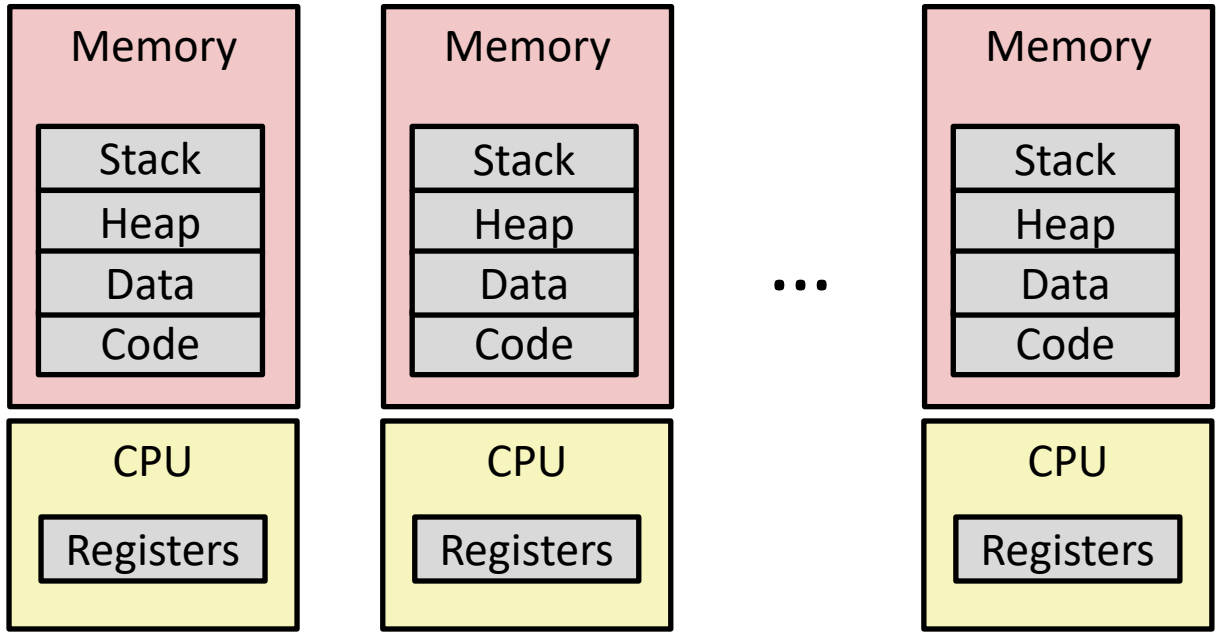
Review: Processes

- Definition: An instance of a program that is being executed (or is ready for execution)

- Consists of:
  - Memory (code, heap, stack, etc)
  - Registers used to manage execution (stack pointer, program counter, ...)
  - Other resources

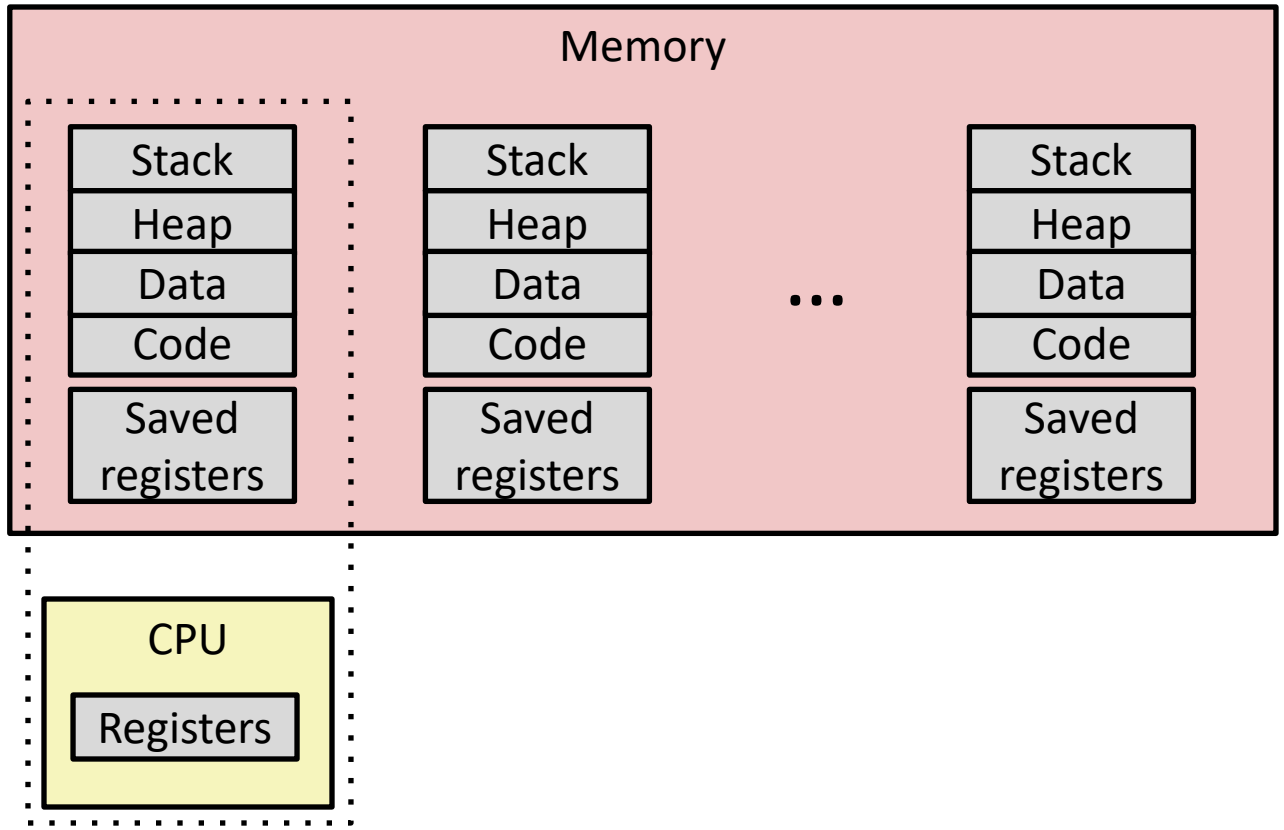


Multiprocessing: The Illusion



- Computer runs many processes simultaneously
  - Applications for one or more users
    - Web browsers, email clients, editors, …
  - Background tasks
    - Monitoring network & I/O devices

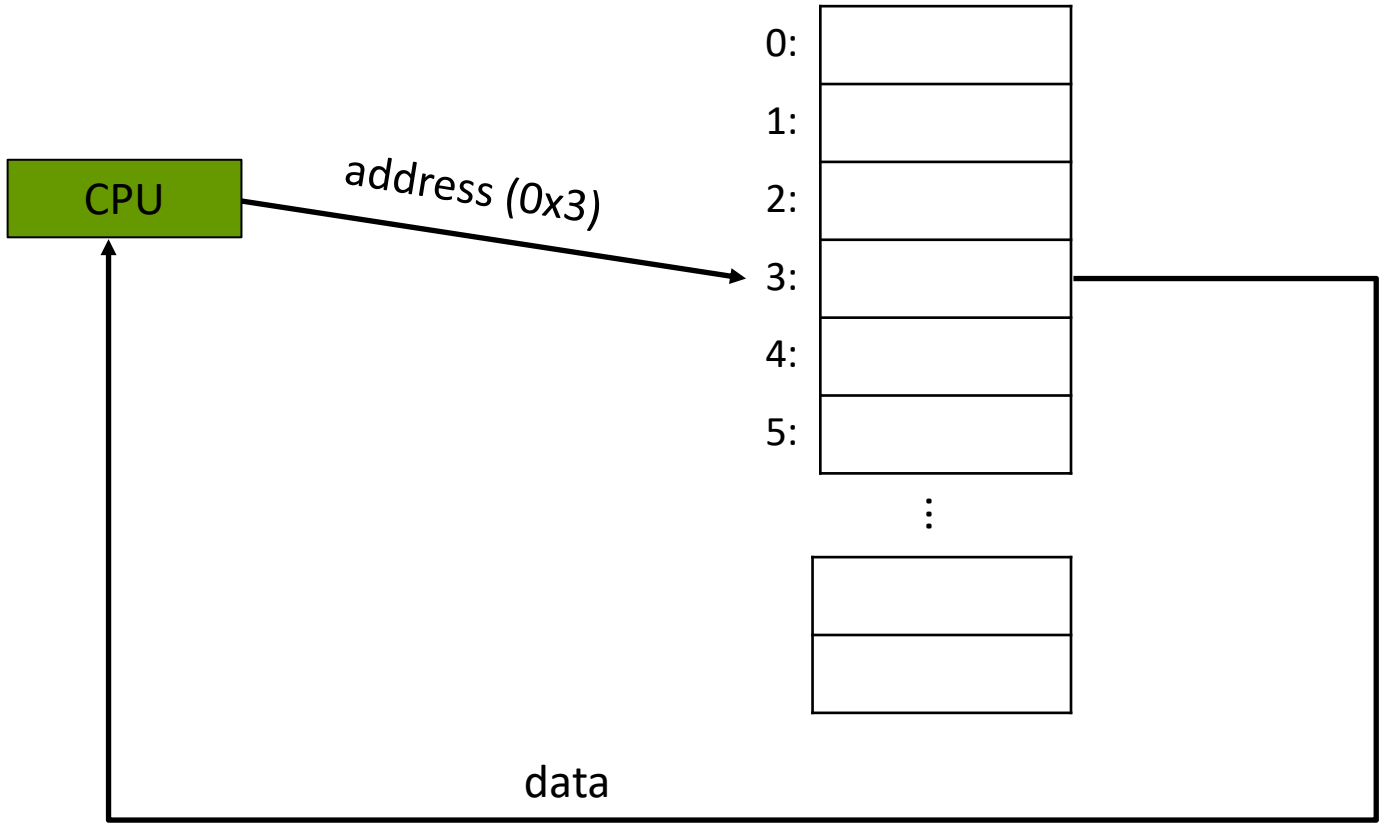
Multiprocessing: The (Traditional) Reality



- Single processor executes multiple processes concurrently
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system (later in course)
  - Register values for nonexecuting processes saved in memory

Memory (as we know it now)

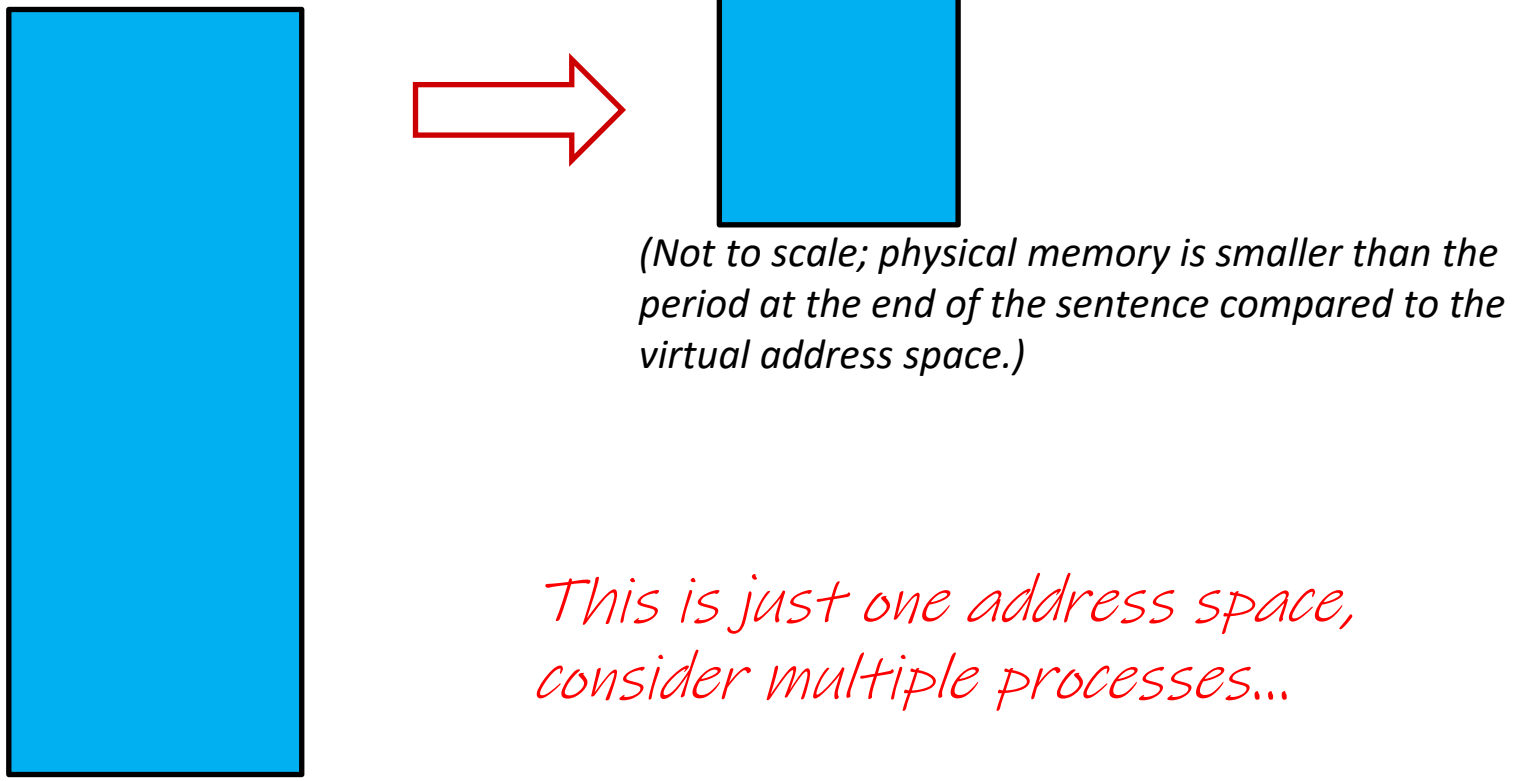
- The CPU directly uses an address to access a location in memory



Problem 1: How does everything fit?

On a 64-bit machine, there are 2^{64} bytes, which is:
18,446,744,073,709,551,616 Bytes
(1.844×10^{19})

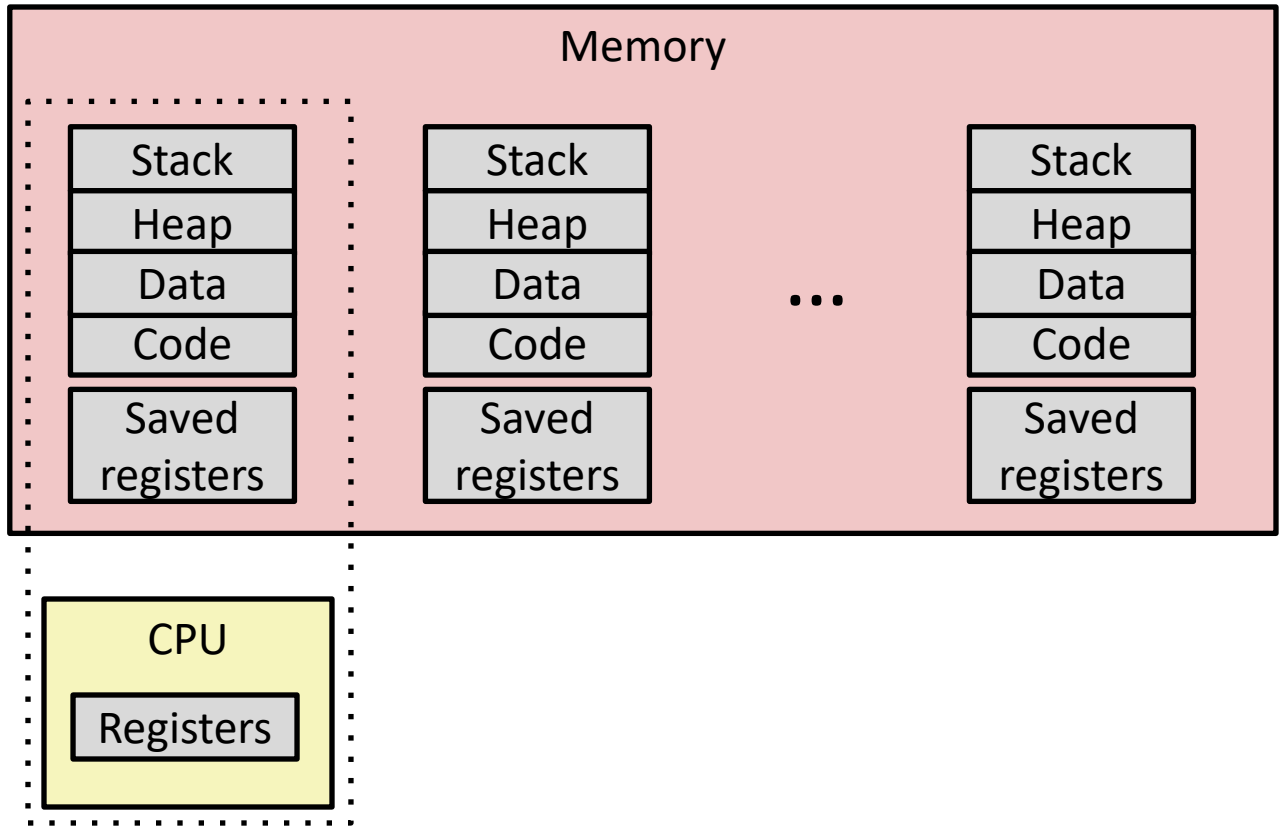
Laptops usually have around 8GB which is 8,589,934,592 Bytes (8.589×10^{9})



(Not to scale; physical memory is smaller than the period at the end of the sentence compared to the virtual address space.)

This is just one address space, consider multiple processes...

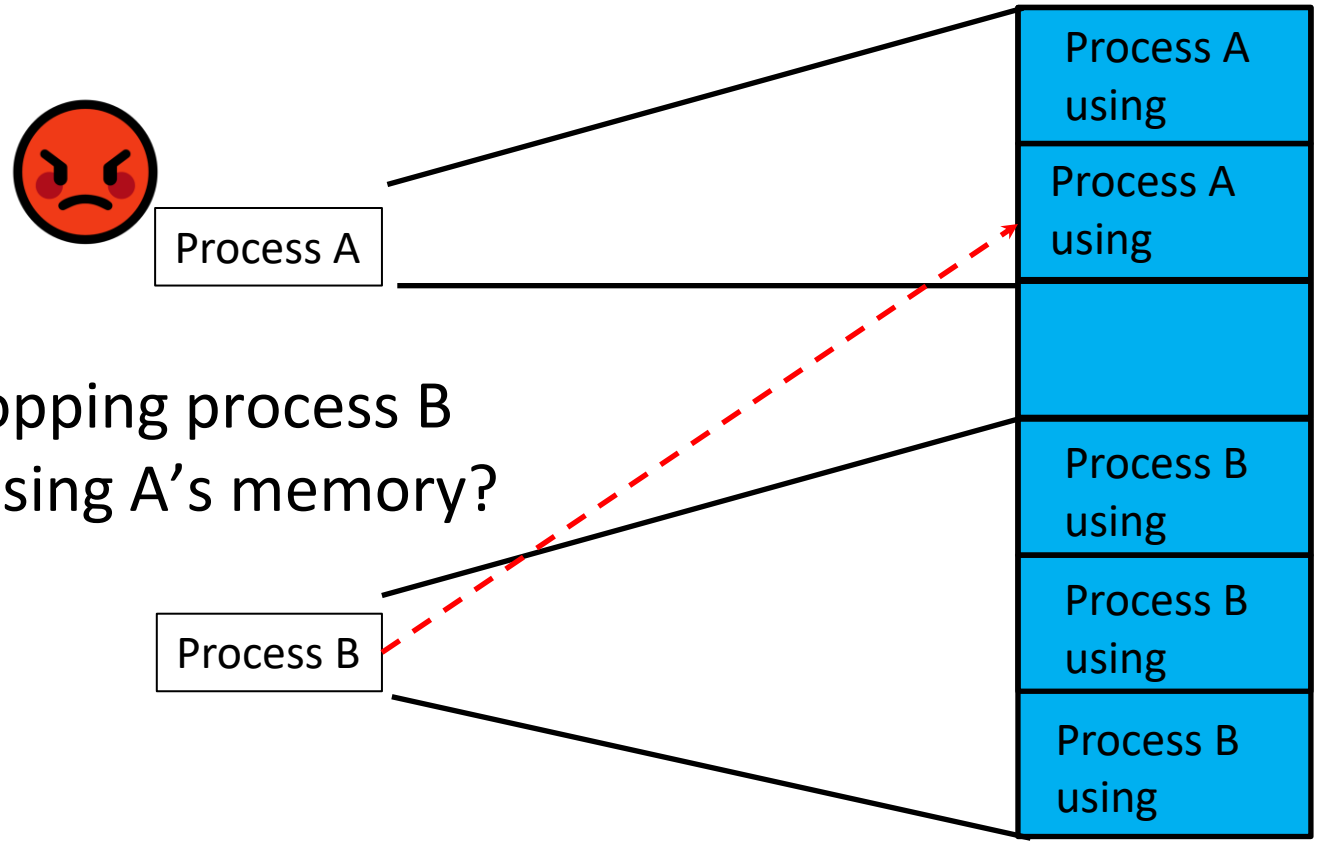
Problem 2: Sharing Memory



- How do we enforce process isolation?
  - Could one process just calculate an address into another process?

Problem 2: Sharing Memory

- How do we enforce process isolation?
  - Could one process just calculate an address into another process?

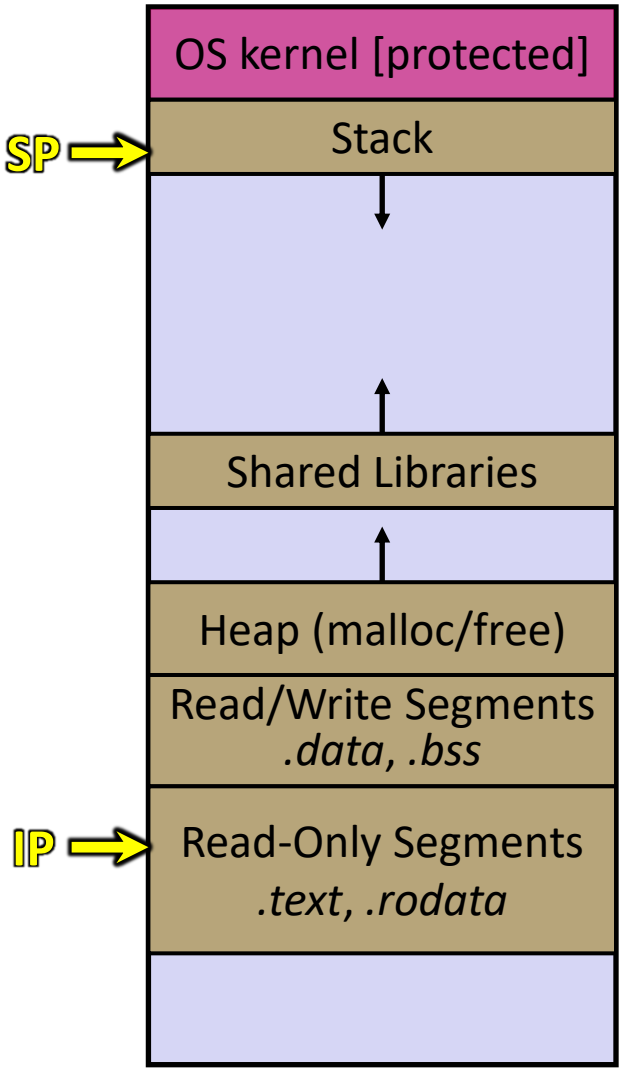


- What is stopping process B from accessing A's memory?

Problem 3: How do we segment things

- A process' address space contains many different "segments"

- How do we keep track of which segment is which and the permissions each segment may have?
  - (e.g., that Read-Only data can't be written)



Idea:

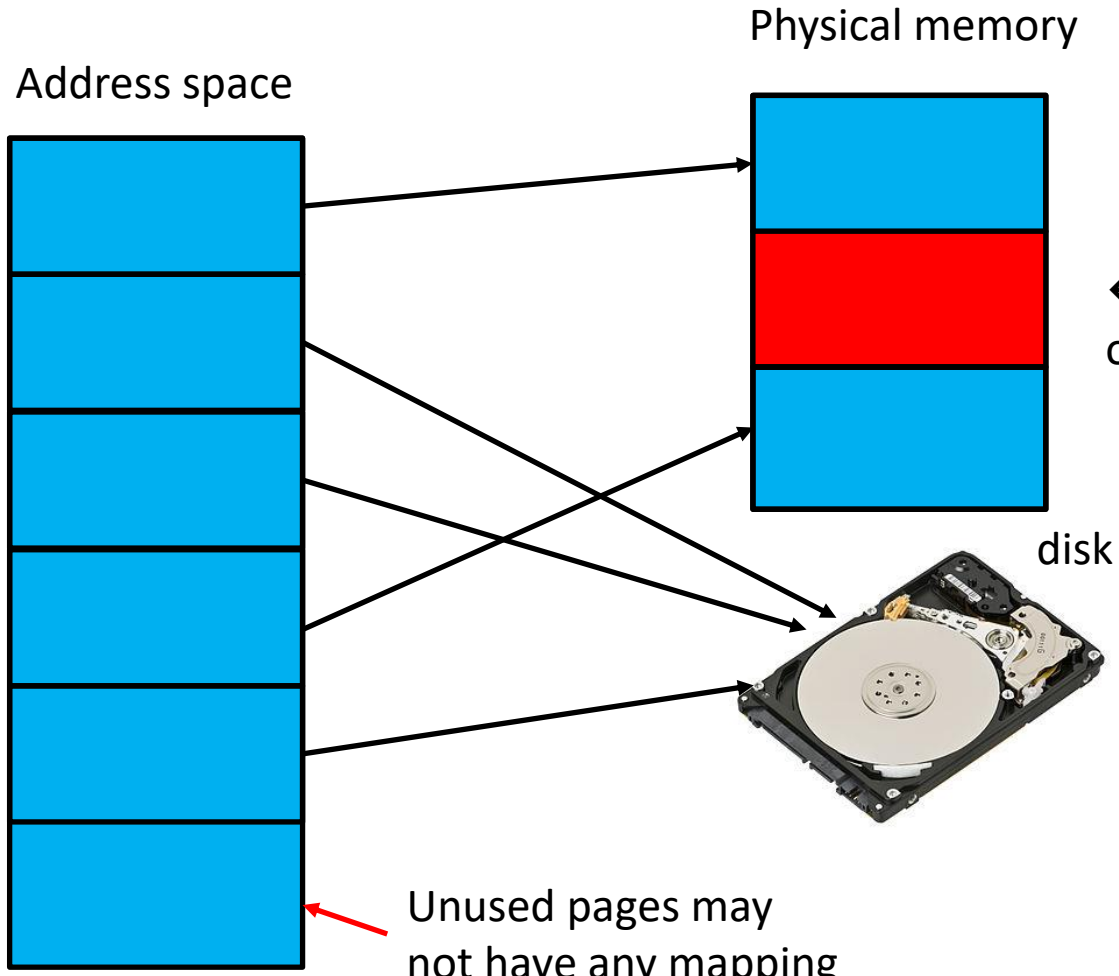
- We don’t need all processes to have their data in physical memory, **just the ones that are currently running**

- For the process’ that are currently running: we don’t need all of their data to be in physical memory, **just the parts that are currently being used**

- Data that isn’t currently stored in physical memory, can be stored elsewhere (disk).
  - Disk is "permanent storage" usually used for the file system
  - Disk has a longer access time than physical memory (RAM)

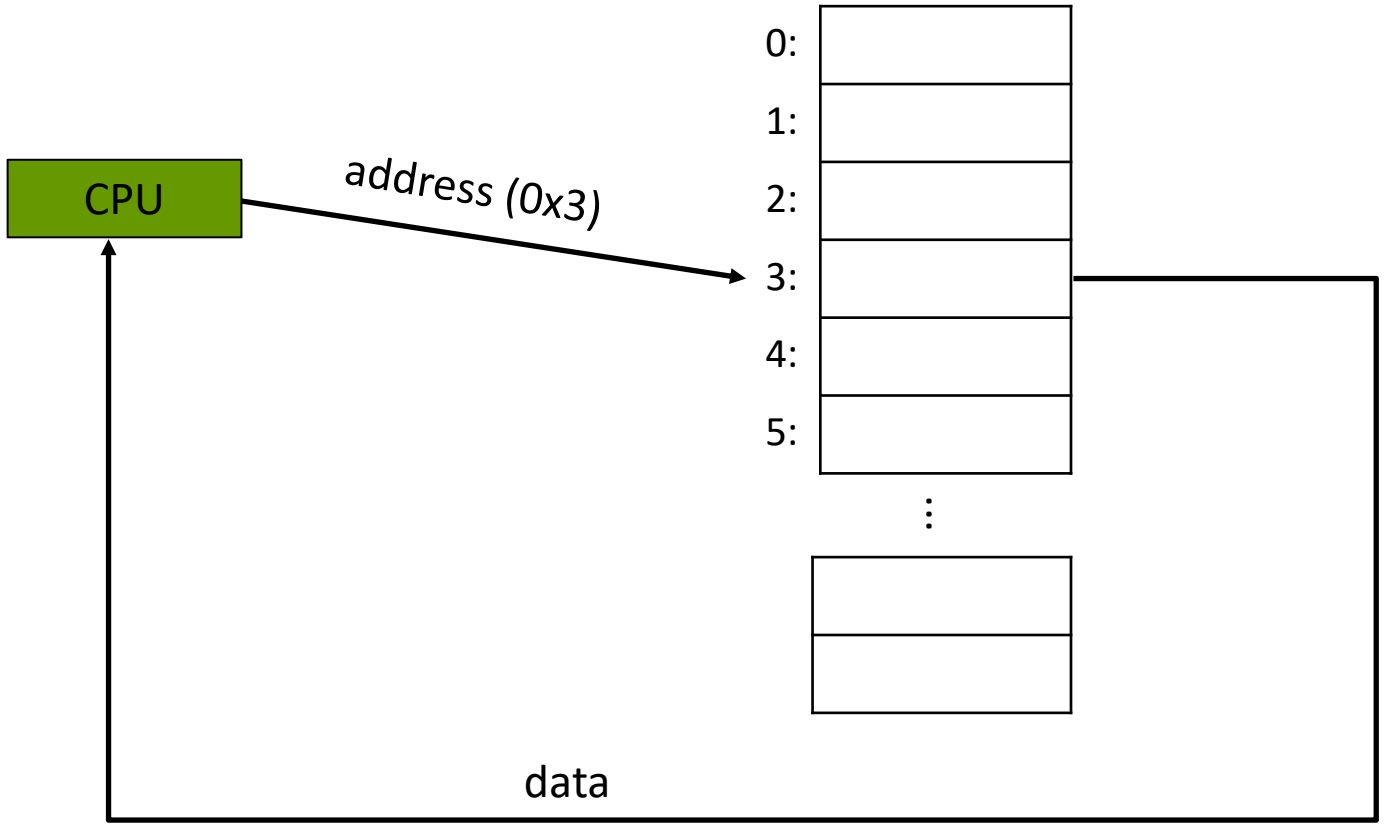
Pages

- Memory can be split up into units called “pages”



This doesn't work anymore

- The CPU directly uses an address to access a location in memory



Indirection

- "Any problem in computer science can be solved by adding another level of indirection."
  - David wheeler, inventor of the subroutine (e.g. functions)

- The ability to indirectly reference something using a name, reference or container instead of the value itself. A flexible mapping between a name and a thing allows chagcing the thing without notifying holders of the name.
  - May add some work to use indirection
  - Example: Phone numbers can be transferred to new phones

- Idea: instead of directly referring to physical memory, add a level of indirection

Definitions

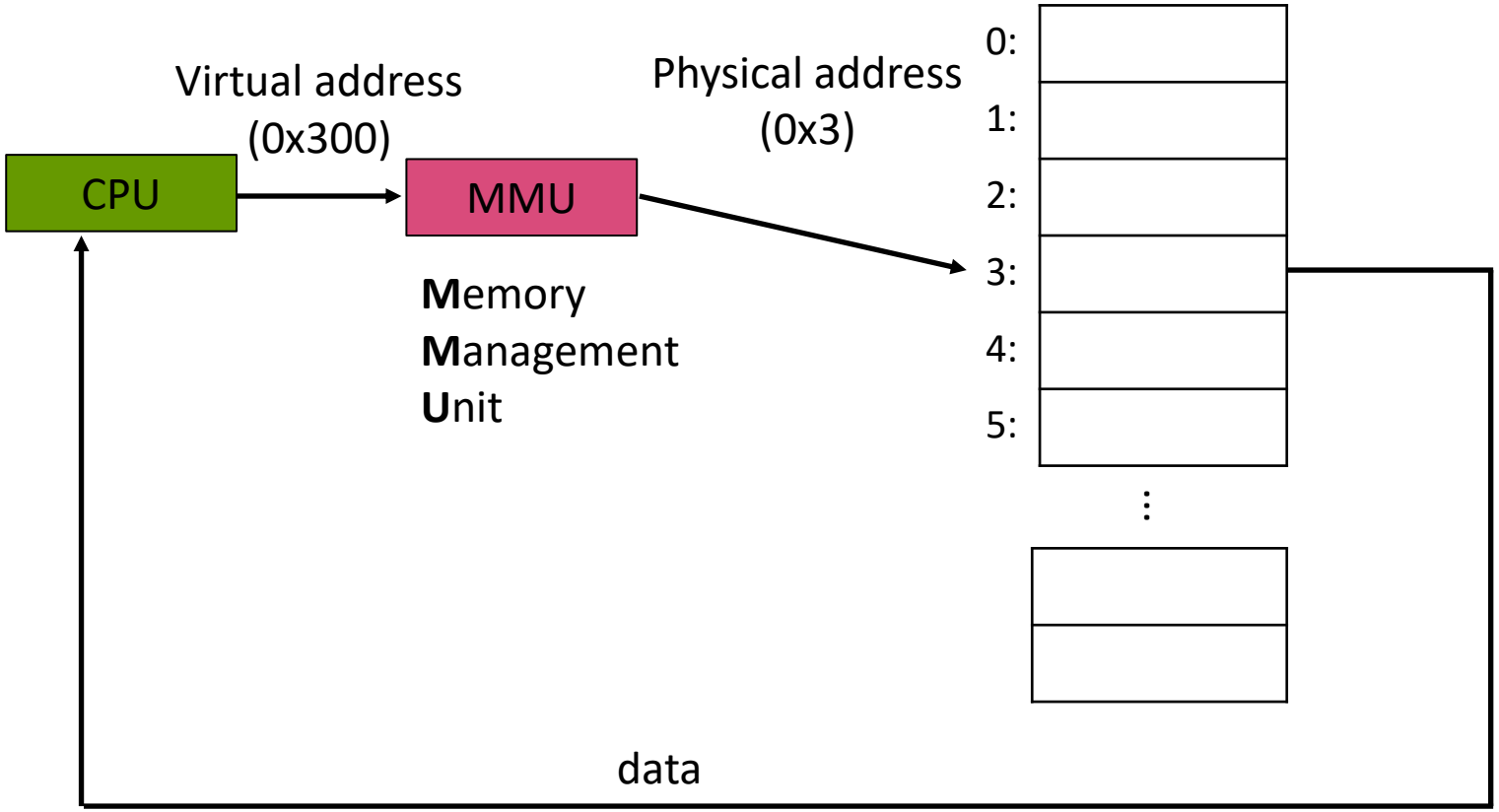
- Addressable Memory: the total amount of memory that can be theoretically be accessed based on:
  - number of addresses (“address space”)
  - bytes per address (“addressability”)

- Physical Memory: the total amount of memory that is physically available on the computer

- Virtual Memory: An abstraction technique for making memory look larger than it is and hides many details from the programs.

Virtual Address Translation

- Programs don't know about physical addresses; virtual addresses are translated into them by the MMU



Page Tables

More details about translation on Wednesday

- Virtual addresses can be converted into physical addresses via a page table.

- There is one page table per processes, managed by the MMU

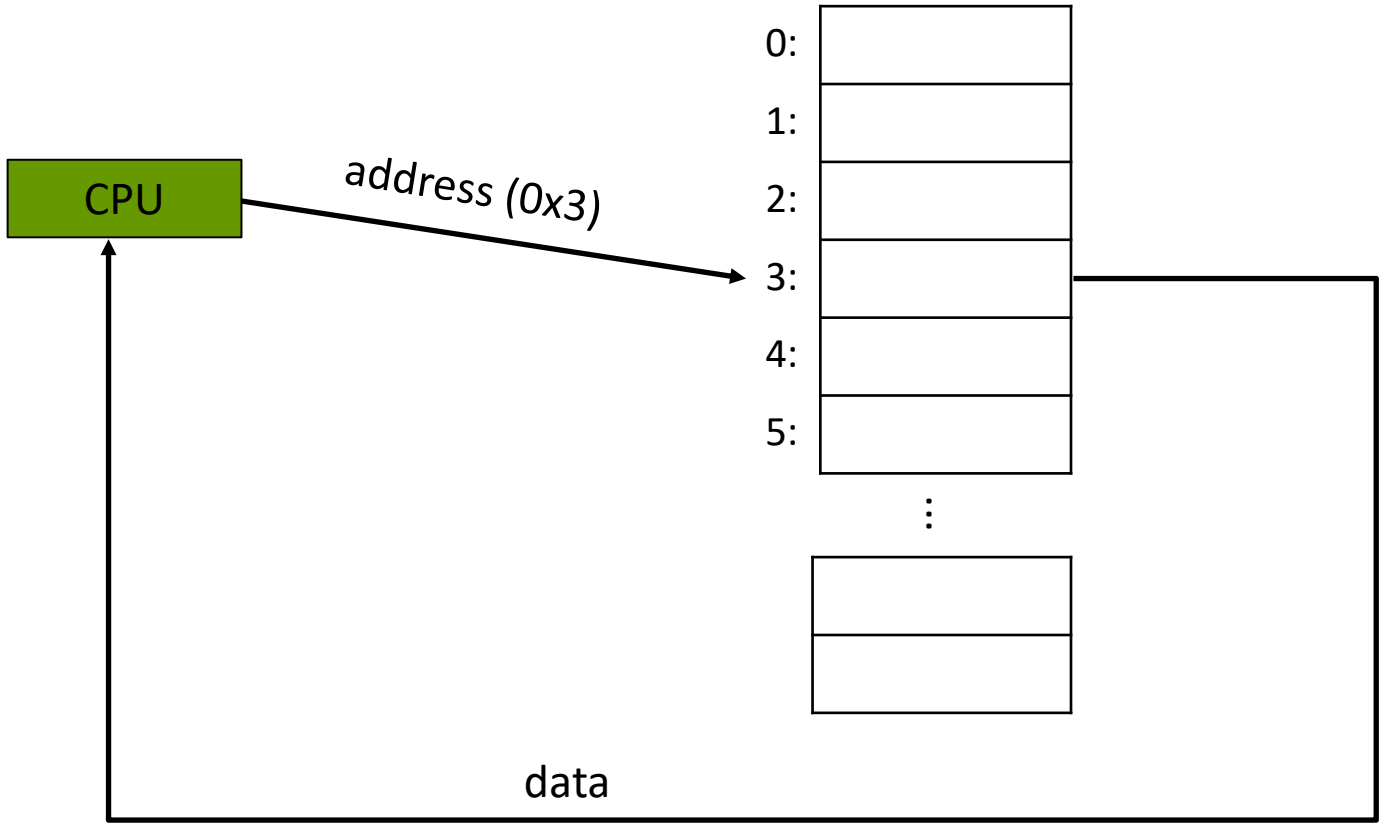
Virtual page #	Valid	Physical Page Number
0	0	null
1	1	0
2	1	1
3	0	disk

Valid determines if the page is in physical memory

If a page is on disk, MMU will fetch it

This doesn't work anymore

- The CPU directly uses an address to access a location in memory



Indirection

- "Any problem in computer science can be solved by adding another level of indirection."
  - David wheeler, inventor of the subroutine (e.g. functions)

- The ability to indirectly reference something using a name, reference or container instead of the value itself. A flexible mapping between a name and a thing allows chagcing the thing without notifying holders of the name.
  - May add some work to use indirection
  - Example: Phone numbers can be transferred to new phones

- Idea: instead of directly referring to physical memory, add a level of indirection

Idea:

- We don’t need all processes to have their data in physical memory, **just the ones that are currently running**

- For the process’ that are currently running: we don’t need all their data to be in physical memory, **just the parts that are currently being used**

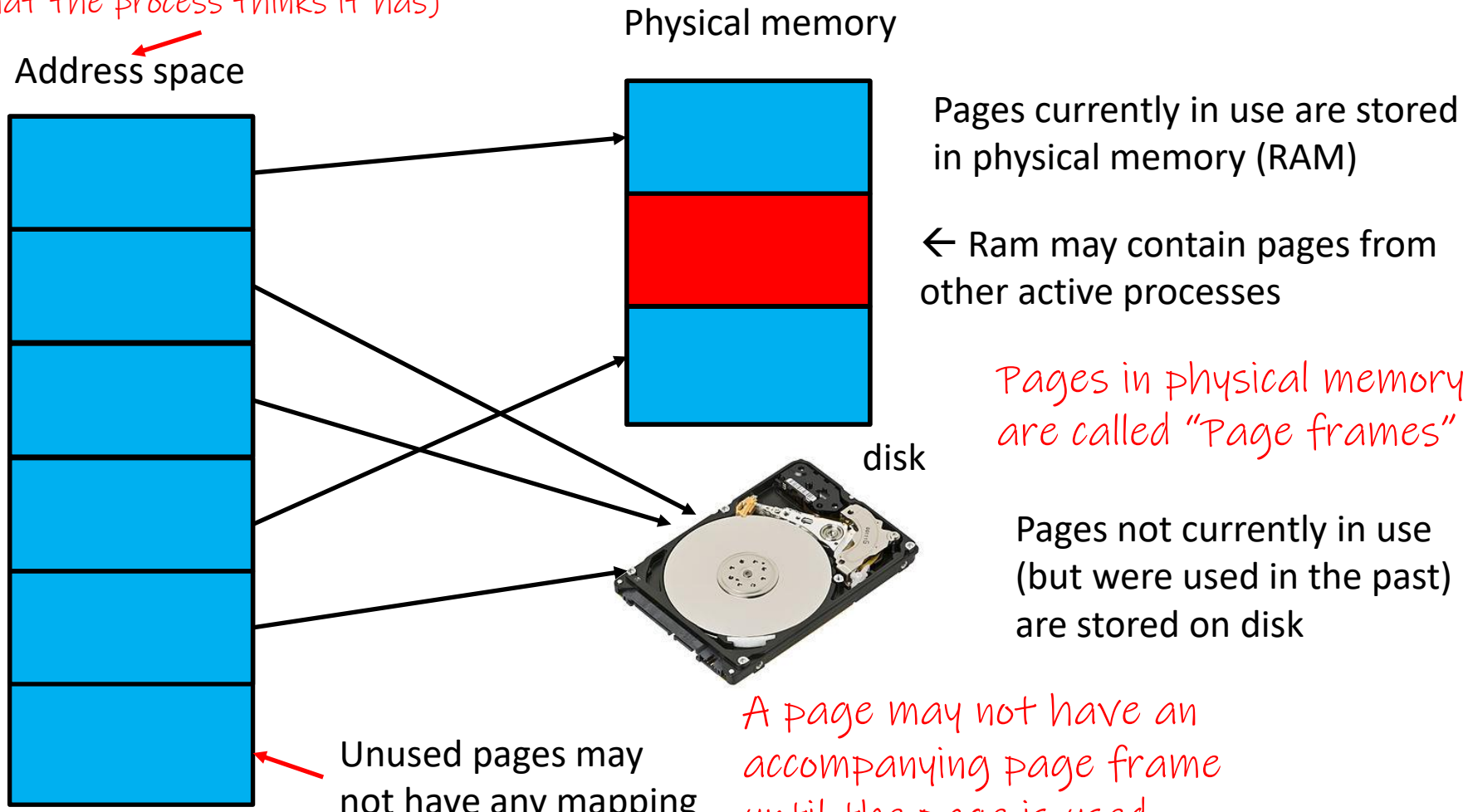
- Data that isn’t currently stored in physical memory, can be stored elsewhere (disk).
  - Disk is "permanent storage" usually used for the file system
  - Disk has a longer access time than physical memory (RAM)

Pages

Pages are of fixed size ~4KB
4KB -> (4 * 1024 = 4096 bytes.)

- Memory can be split up into units called "pages"

(what the process thinks it has)



Pages currently in use are stored in physical memory (RAM)

← Ram may contain pages from other active processes

Pages in physical memory are called "Page frames"

Pages not currently in use (but were used in the past) are stored on disk

A page may not have an accompanying page frame until the page is used

Definitions

Sometimes called "virtual memory" or the "virtual address space"

- Addressable Memory: the total amount of memory that can be theoretically be accessed based on:
  - number of addresses ("address space")
  - bytes per address ("addressability")

IT MAY OR MAY NOT EXIST ONHARDWARE (like if that memory is never used)

- Physical Memory: the total amount of memory that is physically available on the computer

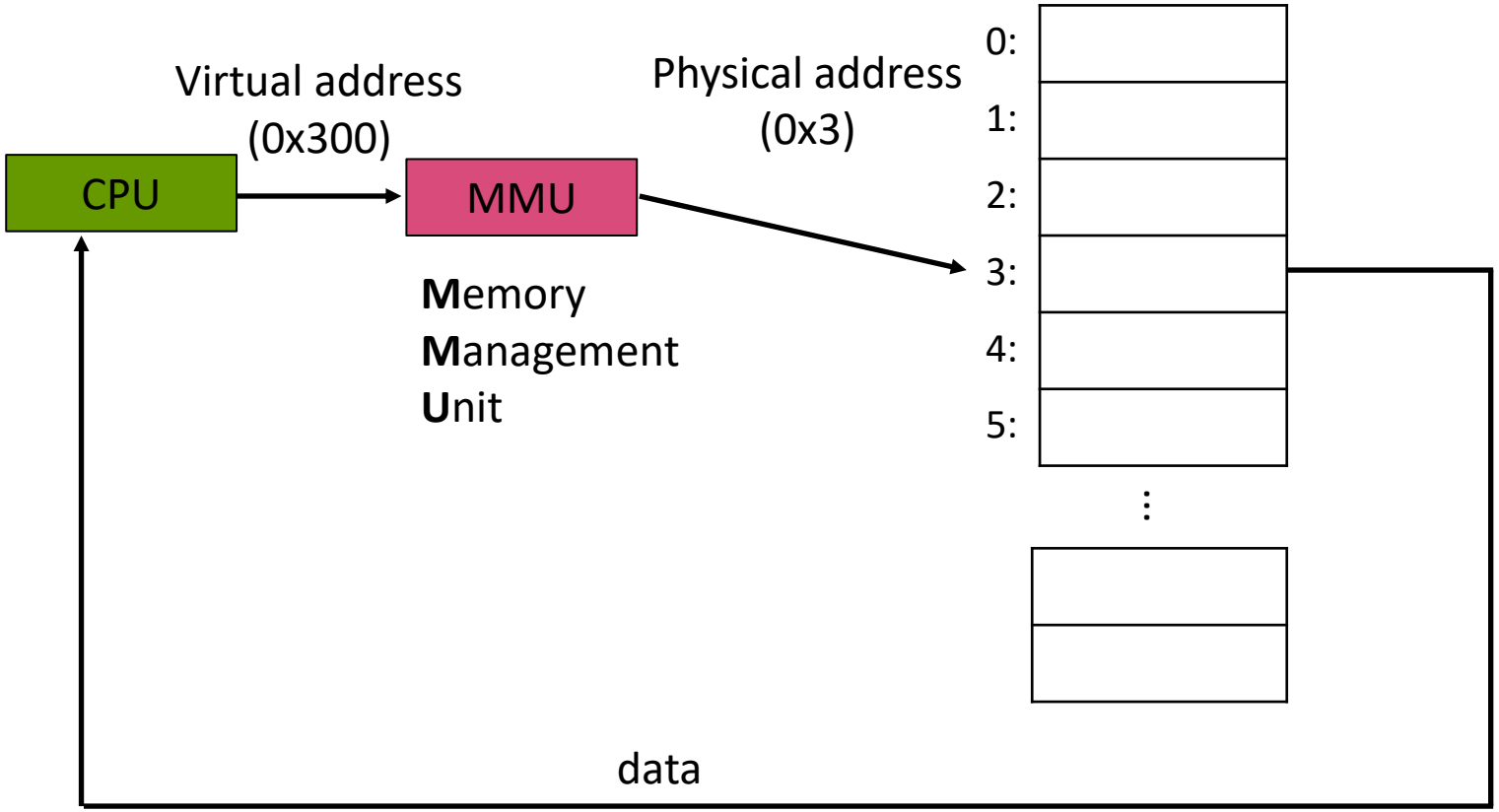
Physical memory holds a subset of the addressable memory being used

- Virtual Memory: An abstraction technique for making memory look larger than it is and hides many details from the programs.

Virtual Address Translation

THIS SLIDE IS KEY TO THE WHOLE IDEA

- Programs don't know about physical addresses; virtual addresses are translated into them by the MMU



Page Tables

More details about translation later

- Virtual addresses can be converted into physical addresses via a page table.

- There is one page table per processes, managed by the MMU

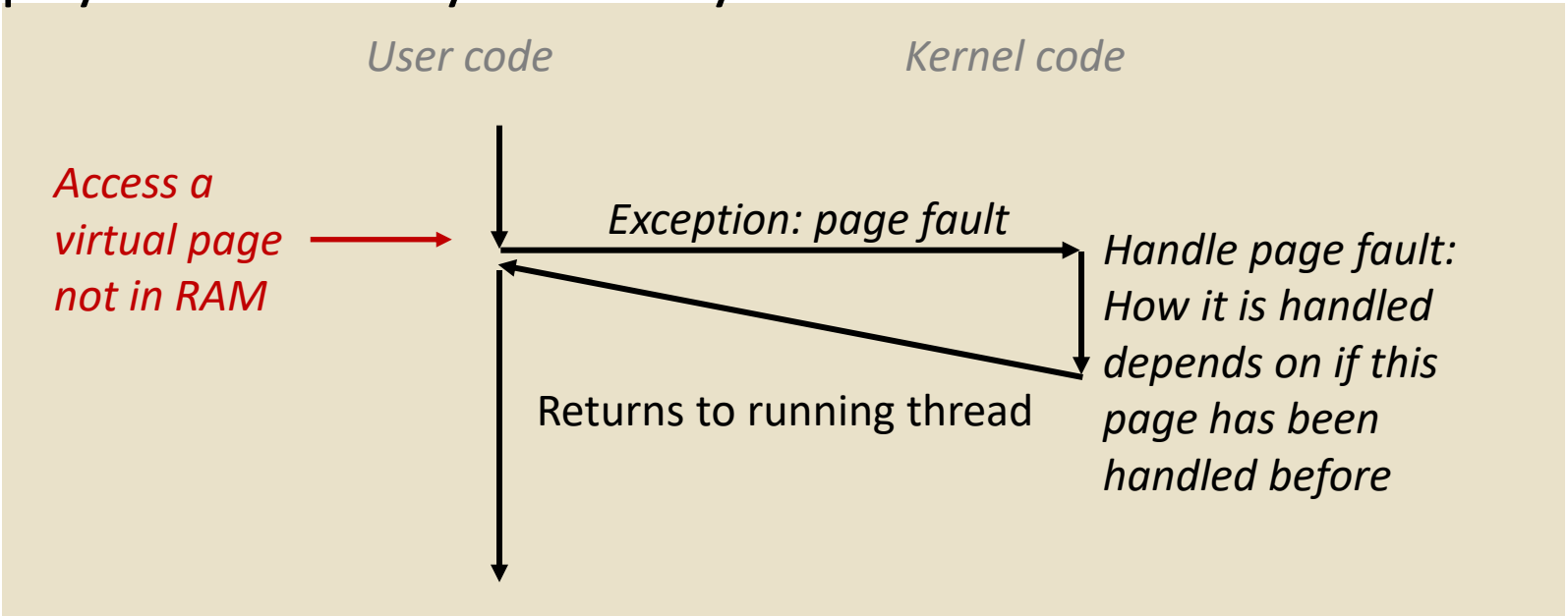
Virtual page #	Valid	Physical Page Number
0	0	null //page hasn't been used yet
1	1	0
2	1	1
3	0	disk

Valid determines if the page is in physical memory

If a page is on disk, MMU will fetch it

Page Fault Exception

- An *Exception* is a transfer of control to the OS *kernel* in response to some ***synchronous event*** *(directly caused by what was just executed)*
- In this case, writing to a memory location that is not in physical memory currently



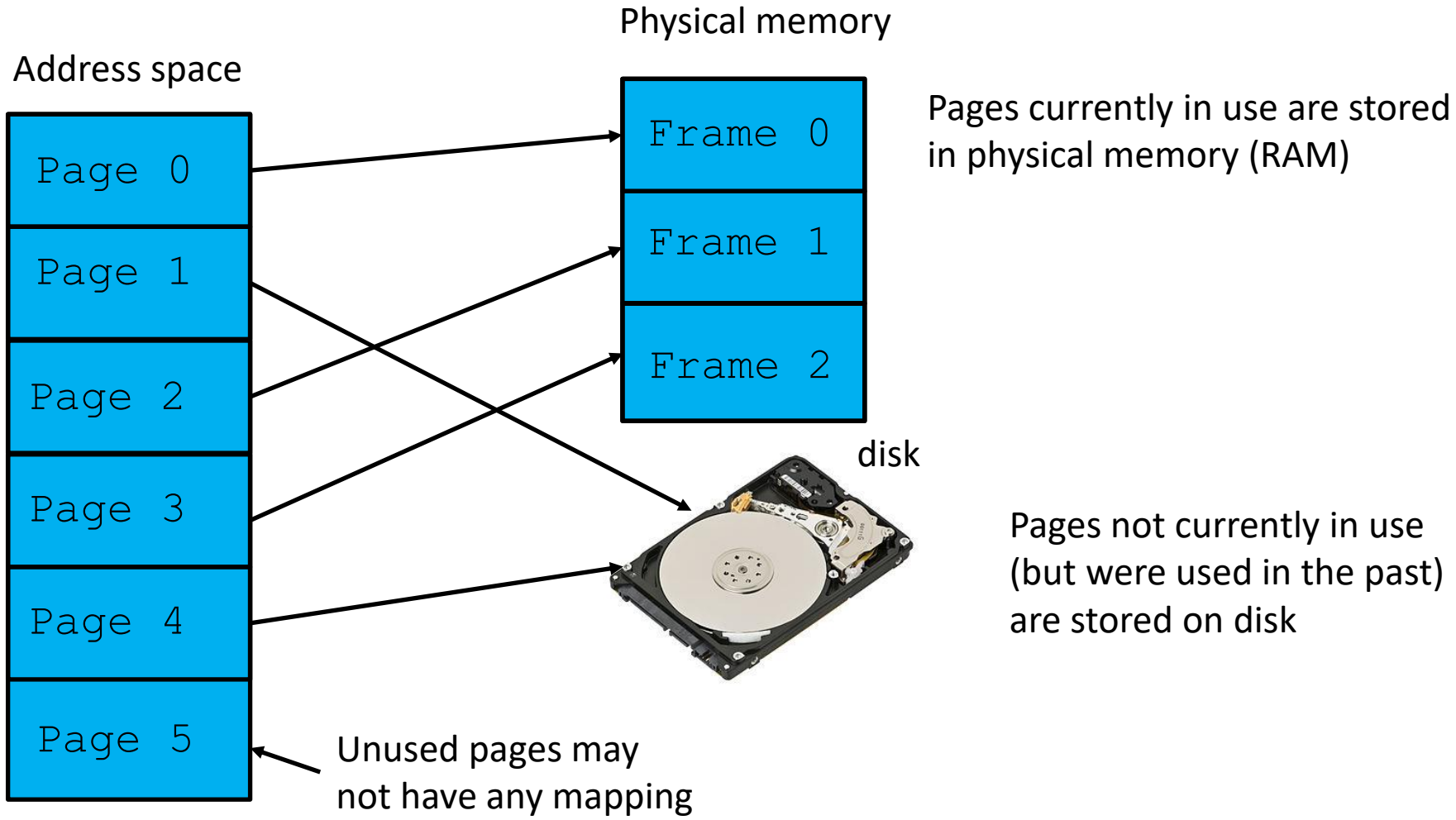
Problem: Paging Replacement

- We don't have space to store all active pages in physical memory.
- If physical memory is full and we need to load in a page, then we choose a page in physical memory to store on disk in the **swap file**
- If we need to load in a page from disk, how do we decide which page in physical memory to "evict"
- Goal: Minimize the number of times we have to go to disk. It takes a while to go to disk.

Poll Everywhere

pollev.com/tqm

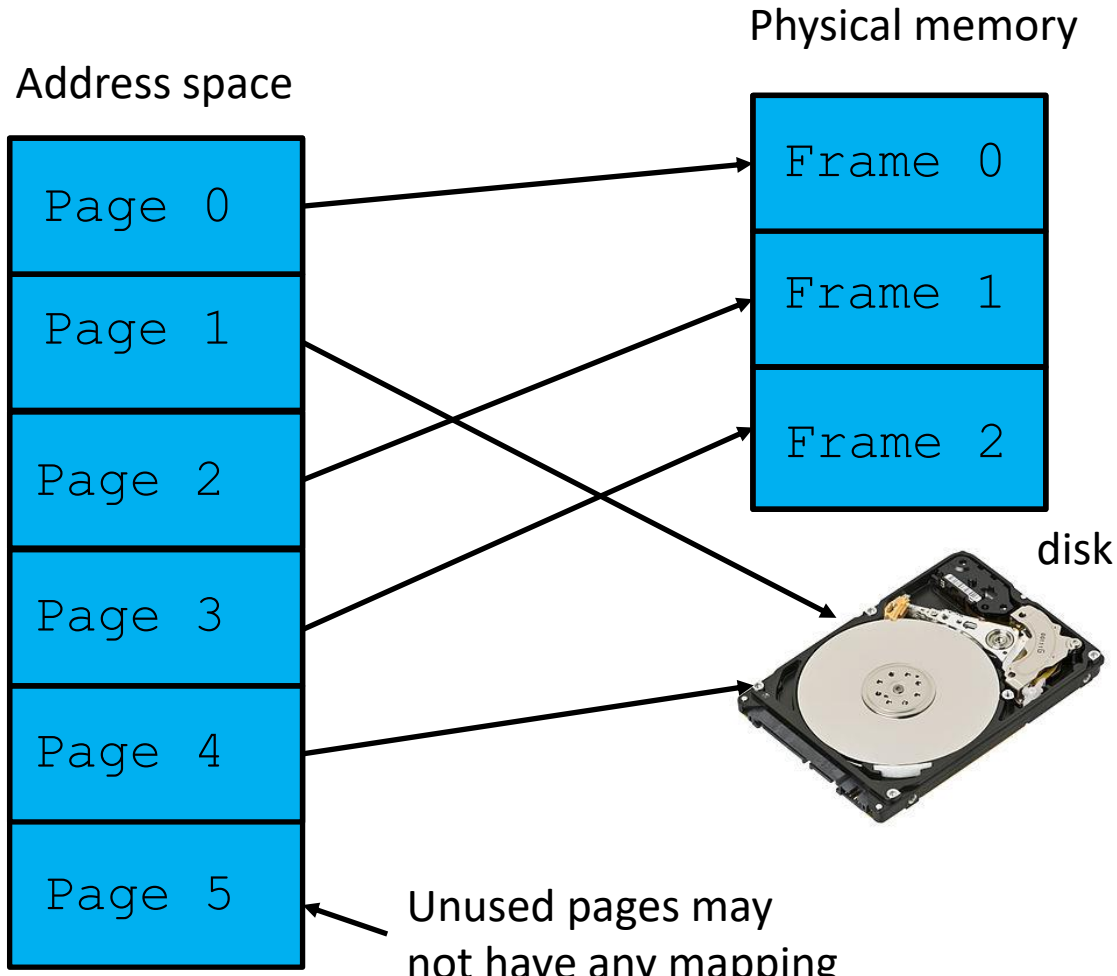
- What happens if this process tries to access an address in page 3?



Poll Everywhere

pollev.com/tqm

- What happens if this process tries to access an address in page 3?



Pages currently in use are stored in physical memory (RAM)

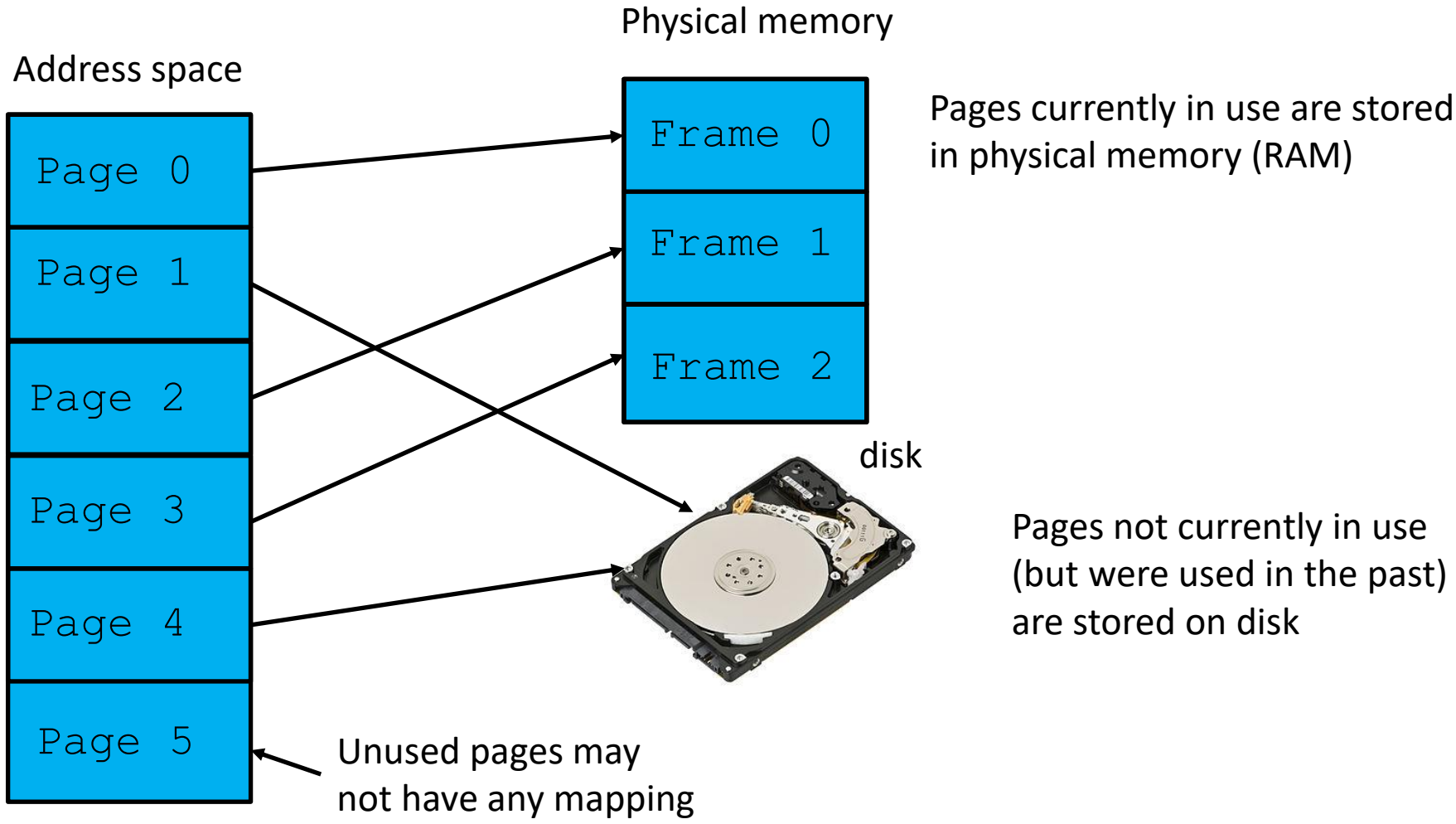
The MMU access the corresponding frame (frame 2)

Pages not currently in use (but were used in the past) are stored on disk

Poll Everywhere

pollev.com/tqm

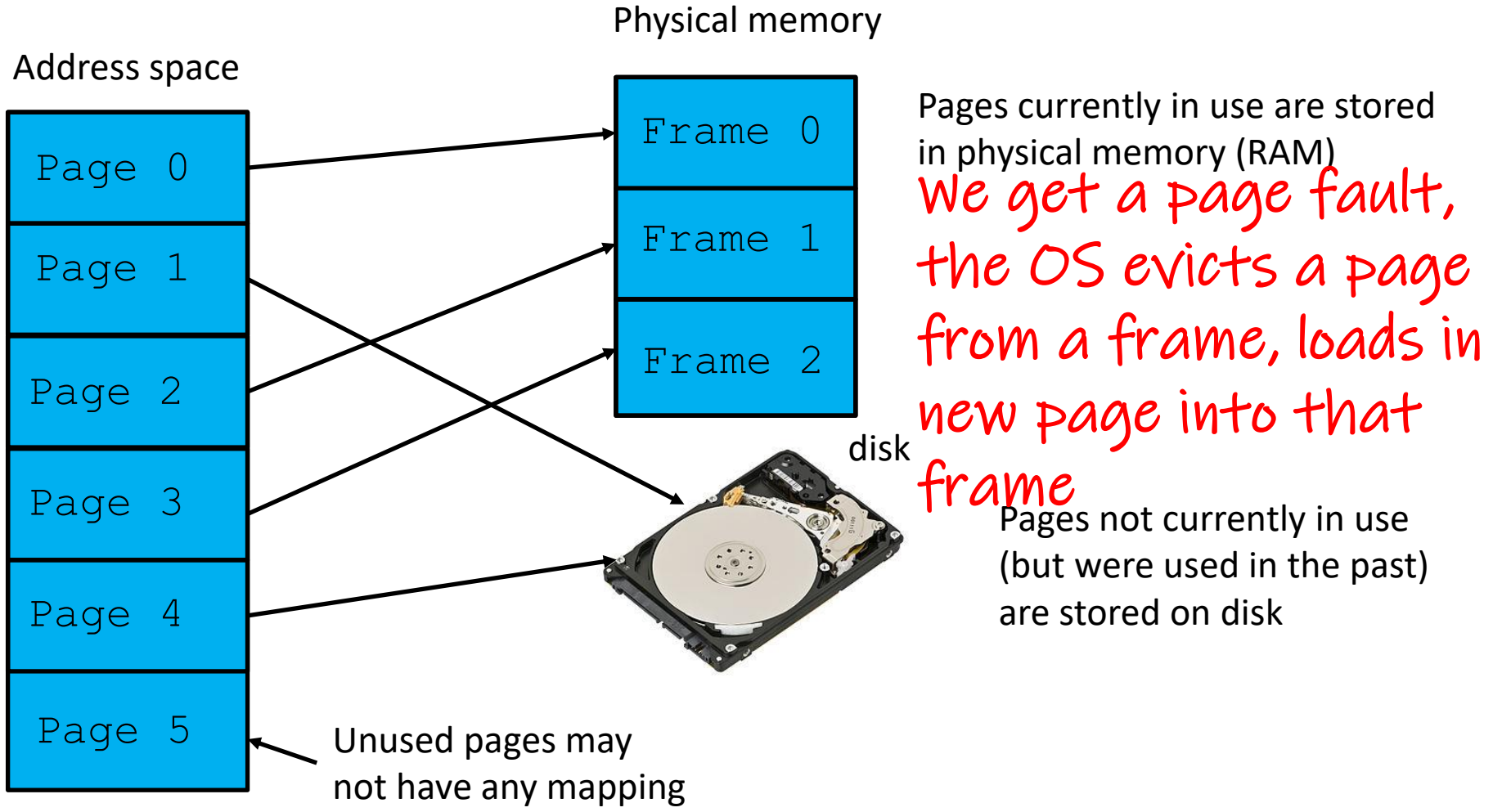
- What happens if we need to load in page 1 and physical memory is full?



Poll Everywhere

pollev.com/tqm

- What happens if we need to load in page 1 and physical memory is full?



Lecture Outline

- C++ Misc
- Locality & Buffering again
- **Virtual Memory**
  - High Level
  - **Page Replacement**

Problem: Paging Replacement

- We don't have space to store all active pages in physical memory.
- If we need to load in a page from disk, how do we decide which page in physical memory to "evict"
- Goal: Minimize the number of times we have to go to disk. It takes a while to go to disk.

Paging Replacement Algorithms

- Simple Algorithms:
  - Random choice
    - "dumbest" method, easy to implement
  - FIFO
    - Replace the page that has been in physical memory the longest

- Both could evict a page that is used frequently and would require going to disk to retrieve it again.

(Theoretically) Optimal Algorithm

- If we knew the precise sequence of requests for pages in advance, we could optimize for smallest overall number of faults
  - Always replace the page to be used at the farthest point in future
  - Optimal (but unrealizable since it requires us to know the future)

- Off-line simulations can estimate the performance of a page replacement algorithm and can be used to measure how well the chosen scheme is doing

- Optimal algorithm can be approximated by using the past to predict the future

Least Recently Used (LRU)

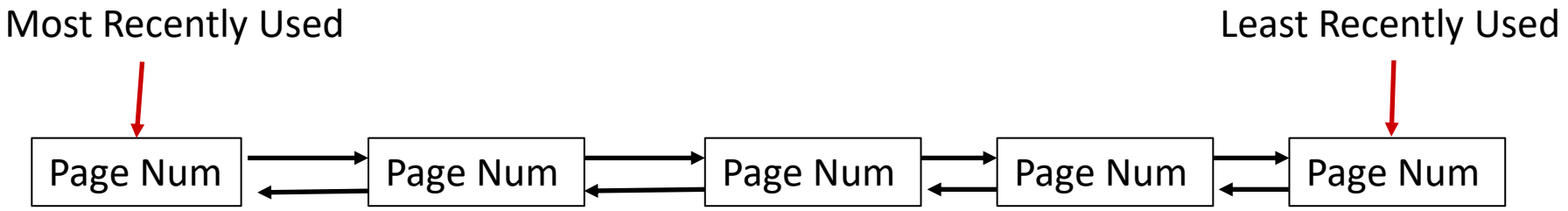
- Assume pages used recently will be used again soon
  - Throw out page that has been unused for longest time
- Past is usually a good indicator for the future
- LRU has significant overhead:
  - A timestamp for *each* memory access that is updated in the page table
  - Sorted list of pages by timestamp

How to Implement LRU?

- Counter-based solution:
  - Maintain a counter that gets incremented with each memory access
  - When we need to evict a page, pick the page with lowest counter
- List based solution
  - Maintain a linked list of pages in memory
  - On every memory access, move the accessed page to end
  - Pick the front page to evict
- HashMap and LinkedList
  - Maintain a hash map and a linked list
  - The list acts the same as the list-based solution
  - The HashMap has keys that are the page number, values that are pointers to the nodes in the linked list to support O(1) lookup

LRU Data Structure

- We can use a linked list to implement LRU

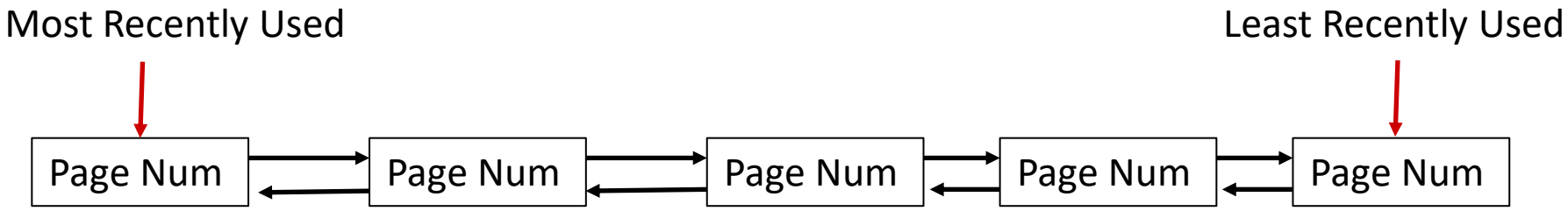


- What is the algorithmic runtime analysis to:
  - lookup a specific block?
  - Removal time?
  - Time to move a block to the front or back?

Discuss

LRU Data Structure

- We can use a linked list to implement LRU



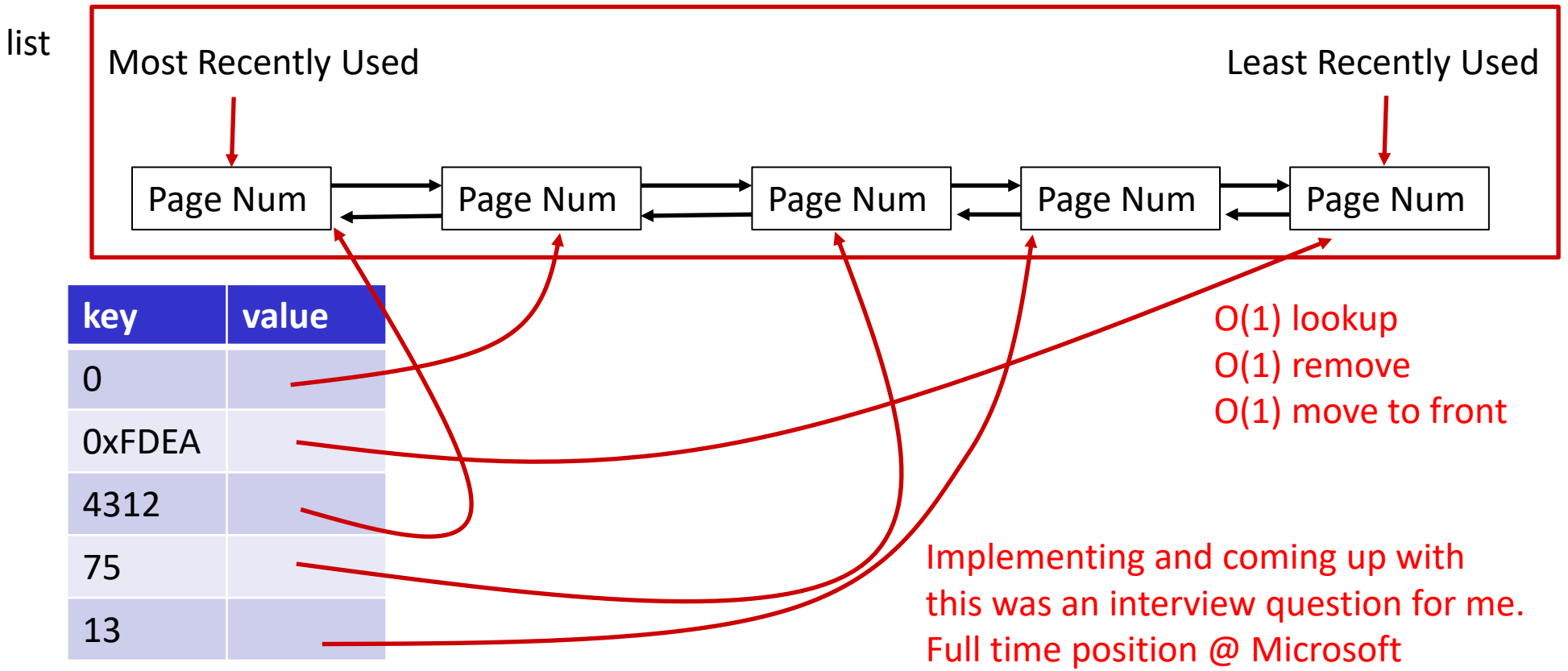
- What is the algorithmic runtime analysis to:
  - lookup a specific block? O(n)
  - Removal time? O(1)
  - Time to move a block to the front or back? O(1)

Discuss

Is there a structure we know of that has O(1) lookup time?

Chaining Hash Cache

- We can use a combination of two data structures:
  - `linked_list<page_info>`
  - `hash_map<page_num, node*>`



That's it for now

- More next time!
  - Threads!!!!!!!!!!!!!
  - My favourite topic ☺
  - And one of the most useful