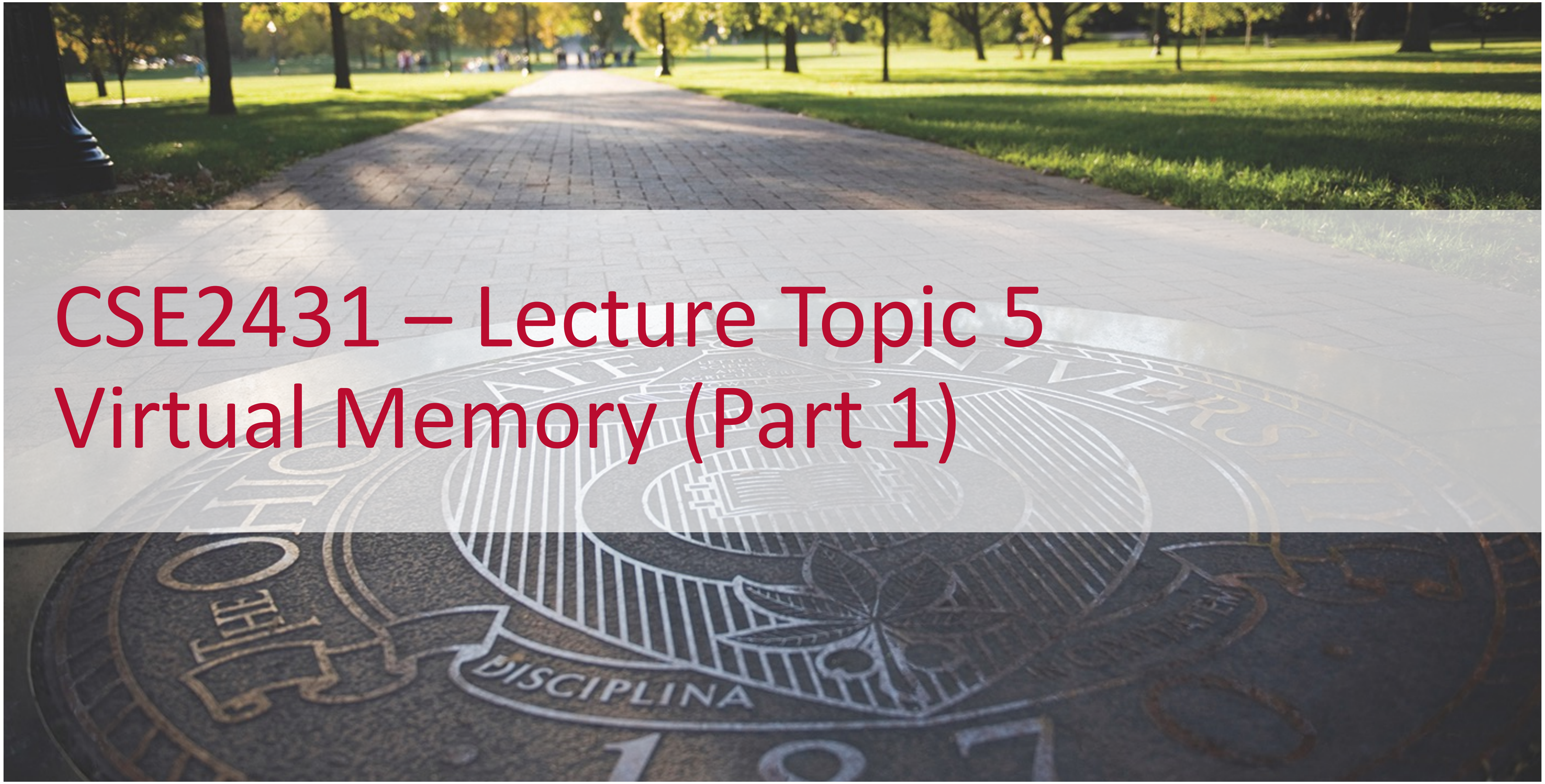


CSE2431 – Lecture Topic 5
Virtual Memory (Part 1)

THE OHIO STATE UNIVERSITY
COLLEGE OF ENGINEERING

Virtual Memory (part 1)

Instructor: Luan Duong, Ph.D.

CSE 2431: Introduction to Operating Systems

Reading: Chapter 13-17 in Required Textbook

Previous Lecture

- **CPU Scheduling Algorithms:**
  - Non-preemptive: FIFO/FCFS, SJF
  - Preemptive: Shortest-Time-to-Completion-First, Round Robin
  - Priority: Multi-level Feedback Queue (MLFQ)
  - Multi-processor scheduling / Real-time scheduling
- **Threads:**
  - What are threads? How to create and run threads
  - Why are threads useful?
  - Thread implementation and scheduling (user-level, kernel-level and hybrid)

Outline: Virtual Memory

- **Memory overview (part 1)**
  - Overview of memory
  - Memory and bit operations
  - Virtual memory in Operating System
  - Virtual memory usage
- Virtual memory procedures (part 2)
  - Dynamic relocation
  - Paging
  - Swaping

Overview of Memory

- **Hardware abstraction**: it contains **a number of bytes** and the CPU can access any byte (byte is the **minimal unit** for **memory access**)
  - CPU tells the memory chip which byte to access (i.e., the address of the byte)

- **Software:** a process' data and code must be loaded into memory before CPU can access them
  - A process can access data at a specific location by using **load/store** instruction
  - A process can control its flow by **using jump/conditional jump instruction**
  - Usually developers don't worry about these; compilers perform the translation

What are bytes? What are bits?

- Bytes/bits are units of measure in this topic
  - b: bit = a single ‘1’ or ‘0’
  - B: byte = 8 bits
  - K = 2^{10}, M=2^{20}, G=2^{30}, T=2^{40}, P=2^{50}, E=2^{60}.

- Example: 1KB means 2^{10} bytes; 1Gb means 2^{30} bits.

- When purchasing something, be careful whether it is **B or b.**
  - In my experience, memory and storage devices usually use B but network devices usually use b.

Bit operations

- How many bits do we need?
- Suppose you have a basket of apples, and you need to give each apple a unique ID, how many bits do you need for this ID?

- How many different entities can N bits represent?
  - Answer: 2^N. E.g. two bits can represent 00, 01, 10, and 11.

- To represent **M different** entities, how many bits do we need?
  - Answer: $[\boldsymbol{log_2 M}]$
  - E.g., if you need to assign a unique ID to 16 apples, you need at least 4 bits for the ID.

Bit operations – Bits in computer memory

- For memory, usually the minimal entity is a **byte**
  - That is why for 32-bit machines, the max memory size is 2^{32} bytes = 4GB
- For disk, usually the minimal entity is a **sector**, which is **512 bytes**
  - That is why for 32-bit machines, the max file size is 2^{32} sectors = 2TB
- **That is why we have moved to 64-bit machines today.**

Bit operations

- We frequently use “/” (divide), “%” (modular), and “*” (multiple) operations in memory management.
  - 13/5=2; 13%5=3; 13*5 = 65

- Let’s do some exercises:
  - 1234567 / 98 = ?; 1234567 % 98 = ?; 1234567 * 98 = ?
  - 1234567 / 100 = ?; 1234567 % 100 = ?; 1234567 * 100 = ?

- As you may notice, for a human being, doing the second row is significantly easier than doing the first row.

Bit operations

- For decimal numbers
  - M % 10^n = last n digits of M;
  - M / 10^n = first (m-n) digits of M, assuming M has m digits.
  - M * 10^n = adding n ”0”s to the end of M

- Similarly, for binary numbers, calculating /, %, and * by factors of 2 is easier.

Bit operations

- M % 2^n = last n bits of M
- M / 2^n = first (m-n) bits of M, assuming M has m bits.
- M * 2^n = M << n

- Examples: 14 / 4 = 3; 14 % 4 = 2; 14 * 4 = 56
  - 14 = 1110b; 4=2^2; 1110b / 2^2 = 11b = 3; 1110b % 2^2 = 10b = 2; 1110b * 2^2 = 111000b = 56

- For computers, calculating "% 2^n", "/ 2^n" and "* 2^n" is easier and faster (same as human beings)
  - That is why in computers, many entities' sizes are 2^n bytes

Bit operations – Summary

- N bits can represent 2^N entities.
- To represent M different entities, we need ceiling($\log_2 M$) bits.

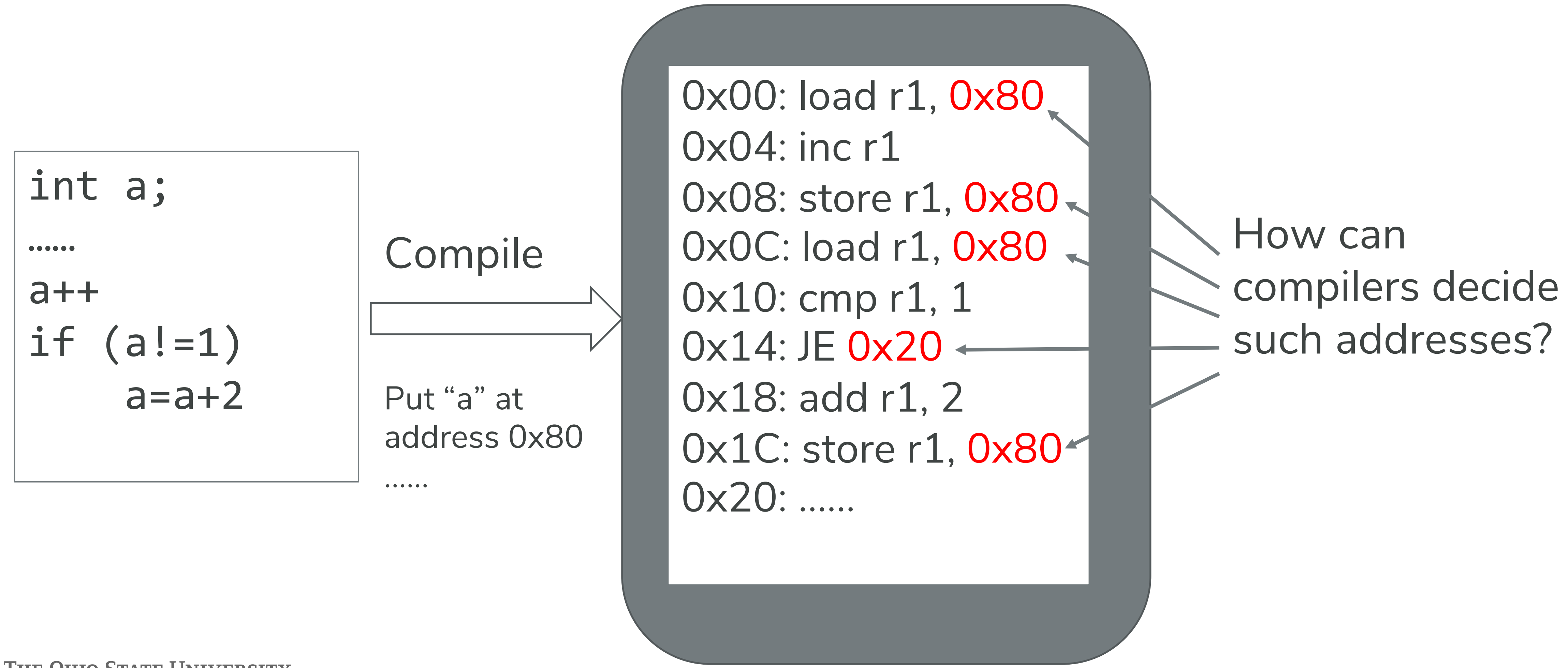
- $M \% 2^n$ = last n bits of M
- $M / 2^n$ = first (m-n) bits of M, assuming M has m bits.
- $M * 2^n = M << n$

- We will need these equations in memory management.

Outline: Virtual Memory

- **Memory overview (part 1)**
  - Overview of memory
  - Memory and bit operations
  - Virtual memory in Operating System
  - Virtual memory usage
- Virtual memory procedures (part 2)
  - Dynamic relocation
  - Paging
  - Swaping

Compiling a program



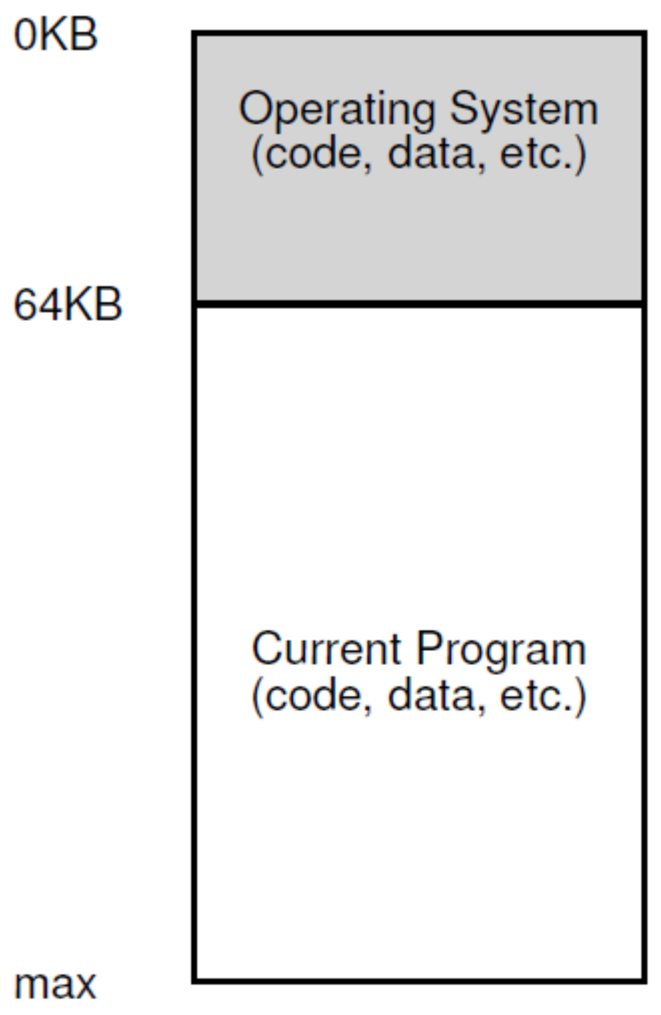
```
int a;
……
a++
if (a!=1)
    a=a+2
```

```
0x00: load r1, 0x80
0x04: inc r1
0x08: store r1, 0x80
0x0C: load r1, 0x80
0x10: cmp r1, 1
0x14: JE 0x20
0x18: add r1, 2
0x1C: store r1, 0x80
0x20: ……
```

Good old days

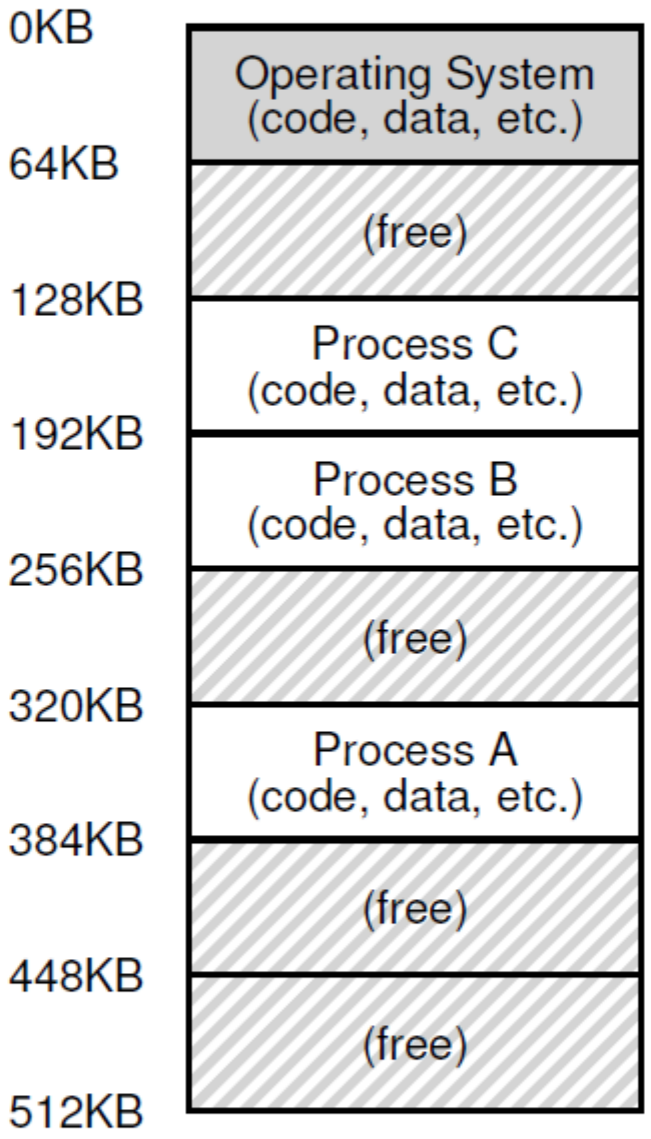
- A computer runs only one process
  - The process is always loaded at a fixed memory location
  - The process has full control of all physical memory (except memory used by OS)

- These facts mean that the memory addresses (used by load/store/jump) of a program can be decided at compile time


0KB
Operating System (code, data, etc.)
64KB
Current Program (code, data, etc.)
max

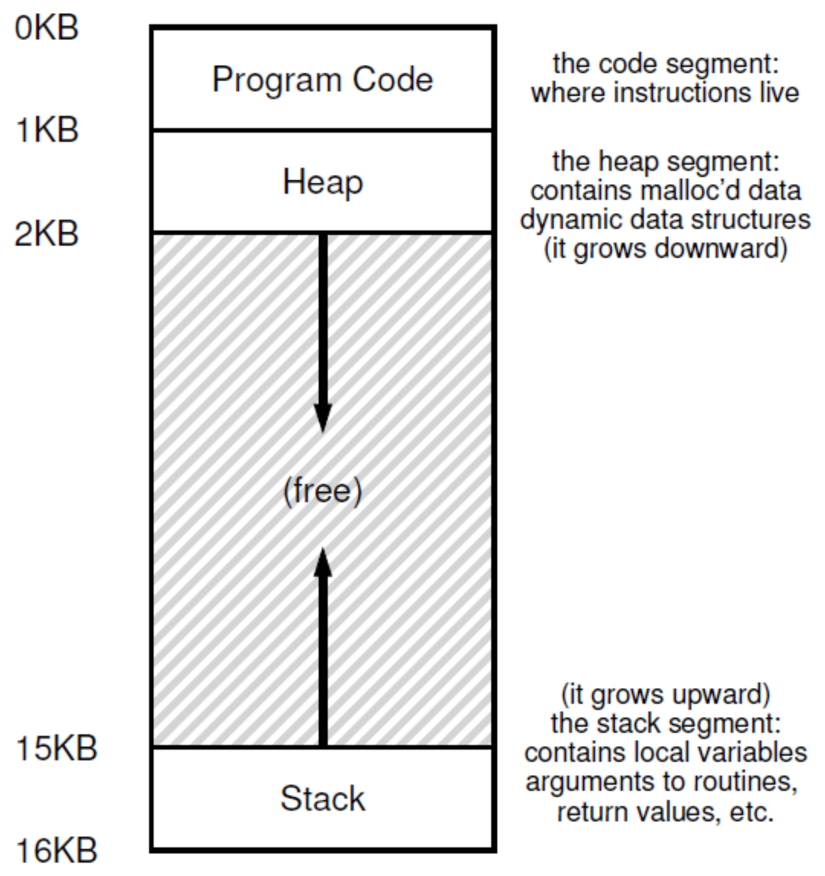
Today's computer

- A computer runs multiple processes in parallel
  - They can be loaded at different locations at different times
  - None have full control of physical memory
  - It's better if we can prevent a process from accessing memory of another process
- Impossible to decide physical memory addresses at compile time.
  - A compiler cannot simply say "let's put variable a at physical address 0x80".


0KB
64KB
128KB
192KB
256KB
320KB
384KB
448KB
512KB
Operating System (code, data, etc.)
(free)
Process C (code, data, etc.)
Process B (code, data, etc.)
(free)
Process A (code, data, etc.)
(free)
(free)

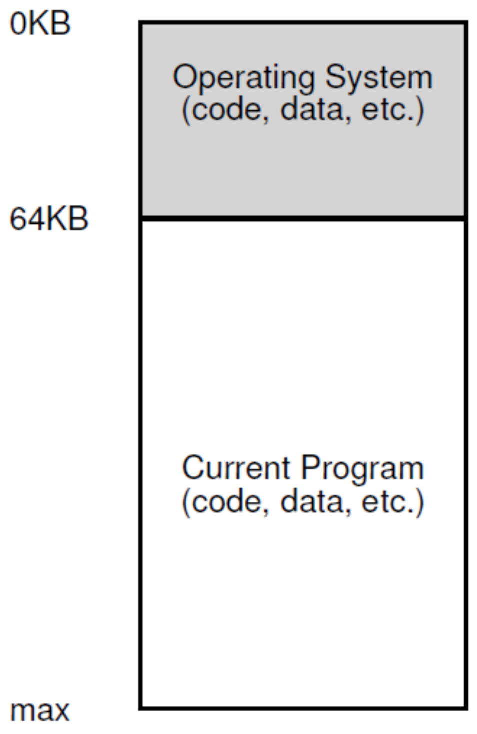
To make things even more complex

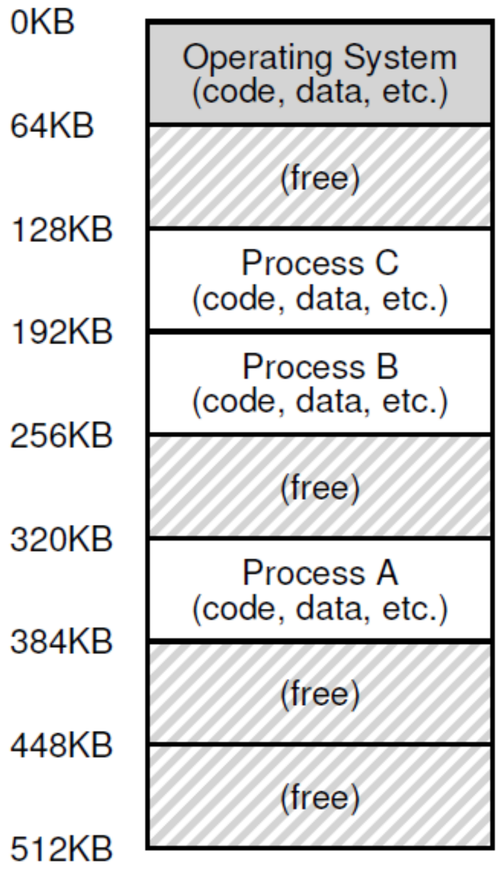
- Even inside a process, memory is not fixed and is not used contiguously
- Address space of a process usually contains multiple segments:
  - Static segment (fixed size): program code and global variables
  - Heap (variable size): contains dynamically allocated memory by malloc
  - Stack (variable size): contains local variables, arguments, and return values of function calls


0KB
1KB
2KB
15KB
16KB
Program Code
Heap
(free)
Stack
the code segment: where instructions live
the heap segment: contains malloc'd data dynamic data structures (it grows downward)
(it grows upward) the stack segment: contains local variables arguments to routines, return values, etc.

Virtual memory in Modern OS

- OS provides an illusion to each process that memory is like this:





But in fact, memory is divided like this!

Virtual memory in Modern OS

- Remember virtualization?
- OS provides an illusion that:
  - Each process has full control of memory
  - Each process has a contiguous virtual address space from 0 (or a fixed offset) to MAX (MAX can be even larger than physical memory size)
- Actually
  - All processes share physical memory
  - A process' memory can be non-contiguous physically
  - A process' data or code can be partially on disk
- OS and CPU collaborate to translate virtual address into physical address.
  - Compilers can say "let's put variable a at virtual address physical address 0x80=

Virtual Memory – Design goals

- **Transparency:** the compilers and programmers should not be aware of the translation between virtual and physical address.
- **Efficiency:** as fast as possible
- **Protection:** one process should not be able to access memory addresses of another process.
  - Otherwise, memory accesses should be restricted operations and we will need a system call for every memory access. That is too expensive.

Memory usage

Before we learn the details of address translation, let’s first review how a programmer uses memory.

Memory Management Interfaces

- Global/Static variables: stored in data segment; compilers handle them
- Local variables: stored in stack; compilers generate code to handle them
- Dynamic allocation: programmers need to use `malloc`/`free` to handle them

`malloc`

- `void* malloc(size_t size)`

- This function is straightforward, you pass it a size asking for some room on the heap, and it either succeeds and gives you back a pointer to the newly-allocated space, or fails and returns NULL

- NULL in C is not anything special, usually just a macro for the value zero.

`malloc`

- `void* malloc(size_t size)`
- This function is straightforward, but deciding size is sometimes harder than you expect.
- Some hints
  - Use `sizeof` instead of using constant integers.
  - Some data structures take additional space. E.g. string takes one additional byte to store "\0" at the end; "\n" may contain two characters.
  - Think about what you actually want to allocate

malloc

- `xPtr = (int *) malloc(100*sizeof(int));`

pointer to desired type

cast the pointer returned by malloc() to the desired pointer type

number of elements to be allocated

number of bytes per element

malloc

- `int *xPtr = (int *) malloc(100*sizeof(int));`
- `xPtr[1] = 10;`
- What is going on in the computer memory?

	Address	Value
`xPtr` ↔	6000	8000
	…..	…..
`*xPtr, xPtr[0]` ↔	8000	unknown
`*(xPtr+1), xPtr[1]` ↔	8004	10
	…..	…..
`*(xPtr+98), xPtr[98]` ↔	8392	unknown
`*(xPtr+99), xPtr[99]` ↔	8396	unknown

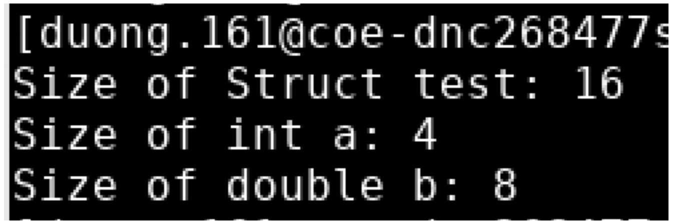
`&(xPtr[98]) is memory address of xPtr[98] and is equal to 8392`

`sizeof`

- `sizeof` operator returns the size of the datatype or the parameter we pass to it.
- Rule of thumb: if the size of a data structure can be determined at compile time, then `sizeof` can help
- Example:
  - `sizeof(int) = 4`
  - `sizeof(float) = 4`
  - `sizeof(char) = 1`
  - `sizeof(double) = 8`

`sizeof`

- Example:
  - `sizeof(int) = 4`
  - `struct test{int a; double b;};`
    - `sizeof(struct test)` (note: `sizeof` struct **may not be equal** to the sum of `sizeof` each element)



```
[duong.161@coe-dnc268477s
Size of Struct test: 16
Size of int a: 4
Size of double b: 8
```

  - `int a[20]; sizeof(a) = sizeof(int)*20`

`sizeof`

- Rule of thumb: if the size of a data structure can be determined at compile time, then `sizeof` can help

- Counter-Example:
  - `int *a = malloc(10*sizeof(int)); sizeof(a)?`
  - **int*** points to an address location as it is a pointer to a variable. So, the sizeof(int *) simply implies the value of the memory location on the machine, and memory locations themselves are 4-byte to 8-byte integer
  - 32-bit Machine: sizeof(int*) will return a value of 4 because add. value of memory location on a 32-bit machine is 4-byte integers.
  - 64-bit Machine: sizeof(int*) will return 8 because address value of memory location on a 64-bit machine is 8-byte integers.

`sizeof`

- Rule of thumb: if the size of a data structure can be determined at compile time, then `sizeof` can help

- Counter-Example:
  - `int *a = malloc(size); sizeof(a)?`
  - `int a[20]; int *b = a;`
    - `sizeof(a) = 20*sizeof(int)`
    - But how about `sizeof(b)? sizeof(b) = sizeof(int *)`
  - `sizeof()` does not take runtime information into consideration.

Alternatives to `sizeof`

- If I malloc() something and want to know its size later, what to do?
  - I need to store the size explicitly as a variable; it’s our own job.

- String has its own function to determine size:
  - E.g. `char str[20]; strcpy(str, “aaa”);`
  - `sizeof(str)=20; strlen(str) = 3;`
  - **So be careful which one you really need!**

Exercise for `malloc`

- Allocate an integer
  - `(int *)val = (int*)malloc(sizeof(int));`
  - `*val = 5;`
- Allocate an array of ten integers
  - `int* val = (int*)malloc(sizeof(int)*10);`
  - `val[0]=5; val[1]=2;`
- Allocate an array of ten pointers to integers
  - `int** val= (int**)malloc(sizeof(int*)*10);`
  - `*val[0]=5; *val[1]=2;`

Exercise for `malloc`

- Allocate an integer
  - `(int *)val = (int*)malloc(sizeof(int));`
  - `*val = 5;`
- Allocate an array of ten integers
  - `int* val = (int*)malloc(sizeof(int)*10);`
  - `val[0]=5; val[1]=2;`
- Allocate an array of ten pointers to integers
  - `int** val= (int**)malloc(sizeof(int*)*10);`
  - ~~`*val[0]=5; *val[1]=2;`~~
  - `val[0]=(int*)malloc(sizeof(int)); *val[0]=5;`

`malloc`: Facts

- `malloc` itself is not a system call. It makes system calls internally.
  - You should not use those system calls directly.

- Memory returned by `malloc` contains random bytes.
  - You cannot assume they are all zeroes. Java's "new" operation, on the other hand, returns all zeroes.

`free`

- `free(void *)`
- It deallocates a piece of memory allocated by malloc.
- Although it looks straightforward, it creates a nightmare for programmers, and it is one of the most important motivations that people invented new programming languages supporting automatic garbage collection.

malloc and free

- **Each malloc call** should be accompanied **with a free call** finally. **No more no less!**

- You should **always** pass the return value of malloc to the free call.

Common mistakes

- Forget to `malloc`
    - `int *val; *val=5;`
- `malloc` with inappropriate size; inappropriate use of sizeof
    - See previous slide
- Forget to initialize
    - Remember malloc returns random bytes

Common mistakes

- Forget to `free`: memory leak
  - It will not have immediate effect if your machine has large memory, but if your program runs for long, it will deplete your memory.
- Use a variable after `free`
- Free a variable more than once
- Free something that is not returned by malloc
  - `int a=0; int *b=&a; free(b)`

Common mistakes

- Be careful of the lifetime of a variable:
  - Global variable: will not be deallocated until the program is terminated
  - Local variable: will be deallocated after a function call returns
  - Malloced variable: will be deallocated after free call

- What is the problem of the following code? How to fix it?

```
int* add(int a, int b){
    int ret = a+b;
    return &ret;
}
```

ret is a local variable; it will be deallocated after "add" function returns, so the return value points to an invalid address
Fix: use "int *ret = malloc(sizeof(int));" instead.

Memory bugs: hard to debug

- Delayed effect: a bug’s effect may not occur when the wrong code is executed. Instead, it may happen later.
  - E.g. when you update an invalid address, your code may succeed (it may also cause a segment fault), but it may change an existing variable, so you will not notice this until you access the other variable.
  - When you find some weird bugs, you should think about memory bugs

- There are tools to help you debug memory problems: ***valgrind***, etc

Short tutoral of Valgrind

- Valgrind: a tool to detect memory errors
  - It is already installed on CSE machines.

- Usage:
  - Compile your program with –g option.
  - Run your program with “valgrind <your program>”
  - It will generate a report. You can use specific options to get more information.

Short tutoral of Valgrind

test1.c

```
1: #include <stdio.h>
2:
3: int main(){
4:     int *a;
5:     printf("a=%d\n", *a);
6: }
```

```
% valgrind ./test1
……
==15718== Use of uninitialised value of size 8
==15718==    at 0x4004D0: main (test1.c:5)
==15718==
==15718== Invalid read of size 4
==15718==    at 0x4004D0: main (test1.c:5)
==15718==  Address 0x0 is not stack'd, malloc'd or
(recently) free'd
……
```

Short tutoral of Valgrind

```
test2.c
1: #include <stdio.h>
2: #include <stdlib.h>
3:
4: int main(){
5:     int *a=malloc(sizeof(int)*3);
6:     int i;
7:     for(i=0; i<4; i++){
8:         a[i]=i;
9:     }
10:   free(a);
11:}
```

```
% valgrind ./test2
......
==16983== Invalid write of size 4
==16983==    at 0x400533: main (test2.c:8)
==16983==  Address 0x4c3504c is 0 bytes after a block of size 12 alloc'd
==16983==    at 0x4A06A2E: malloc (vg_replace_malloc.c:270)
==16983==    by 0x400515: main (test2.c:5)
......
```

Short tutoral of Valgrind

test3.c

```
1: #include <stdio.h>
2: #include <stdlib.h>
3:
4: int main(){
5:     int *a=malloc(sizeof(int)*4);
6:     int i;
7:     for(i=0; i<4; i++){
8:         a[i]=i;
9:     }
10:}
```

```
% valgrind ./test3
.......
==17388==     possibly lost: 12 bytes in 1 blocks
......
==17388== Rerun with --leak-check=full to see
details of leaked memory
```

```
% valgrind --leak-check=full ./test3
......
==17442== 12 bytes in 1 blocks are possibly lost
in loss record 1 of 1
==17442==    at 0x4A06A2E: malloc
(vg_replace_malloc.c:270)
==17442==    by 0x4004D5: main (test3.c:5)
......
```

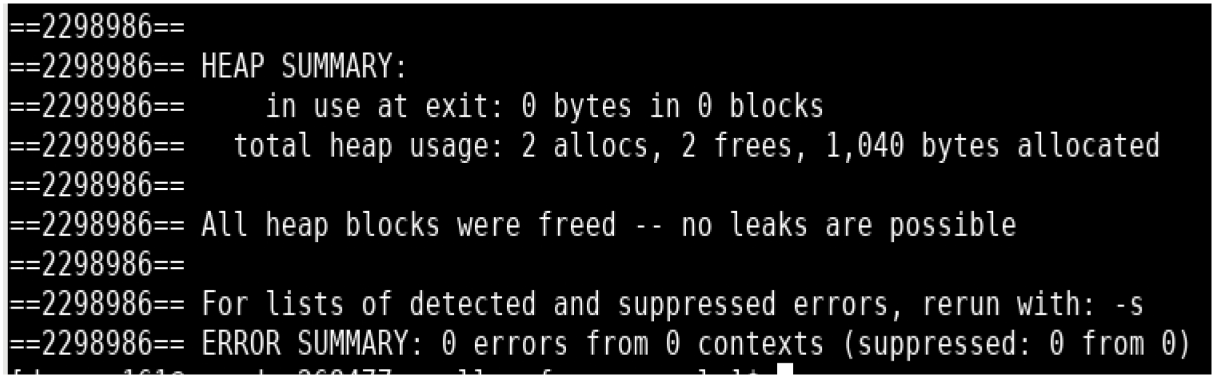
Short tutoral of Valgrind

Test_fix.c

```c
#include <stdio.h>
#include <stdlib.h>

int main(){
    int *a=malloc(sizeof(int)*4);
    int i;
    for(i=0; i<4; i++){
        a[i]=i;
    }
   free(a);
}
```

% valgrind ./test_fix



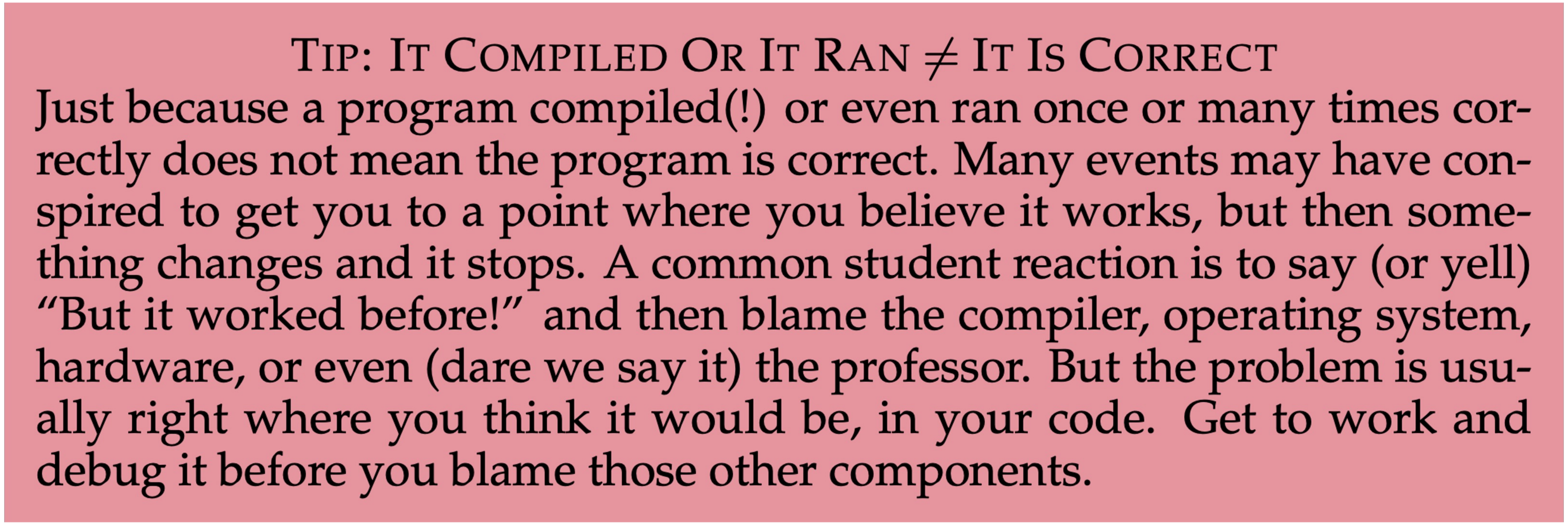
```
==2298986==
==2298986== HEAP SUMMARY:
==2298986==     in use at exit: 0 bytes in 0 blocks
==2298986==   total heap usage: 2 allocs, 2 frees, 1,040 bytes allocated
==2298986==
==2298986== All heap blocks were freed -- no leaks are possible
==2298986==
==2298986== For lists of detected and suppressed errors, rerun with: -s
==2298986== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Valgrind is NOT a panacea

- There are problems that it cannot detect
  - E.g. Your code accesses an array with an out-of-bound index, but the address may point to another valid variable.

- Whenever you are using a pointer or an array, ask yourself:
  - Where is it allocated?
  - What is its size?
  - Where is it deallocated?
  - **If you don’t know the answer, then there is probably a problem**

A must-know (from [OSTEP] book)



TIP: IT COMPILED OR IT RAN $\neq$ IT IS CORRECT

Just because a program compiled(!) or even ran once or many times cor-
rectly does not mean the program is correct. Many events may have con-
spired to get you to a point where you believe it works, but then some-
thing changes and it stops. A common student reaction is to say (or yell)
"But it worked before!" and then blame the compiler, operating system,
hardware, or even (dare we say it) the professor. But the problem is usu-
ally right where you think it would be, in your code. Get to work and
debug it before you blame those other components.

Java

- Managing memory for a complex software is a nightmare.
- People invented new programming languages, represented by Java:
  - Automatic detection of accesses to unallocated memory
  - Automatic initialization
  - Automatic garbage collection: no need to free.
    - It alleviates the memory leak problem, but does not eliminate it: the programmers still need to make sure the unused variables are not referenced any more.
- They make memory management significantly easier, at the cost of performance.

Summary: Virtual memory overview

- **Memory overview (part 1)**
  - Overview of memory
  - Memory and bit operations
  - Virtual memory in Operating System
  - Virtual memory usage (`malloc, sizeof, free`)
- Virtual memory procedures (part 2)
  - Dynamic relocation
  - Paging
  - Swaping