Today’s Plan

- P0 Review, Q&A — review the concepts of memory and pointers
- EGOS demo — a demo of our operating system starting from P1
- Context & Threads — introduce two new concepts for P1 (just a start)

Review

```
1: int main() {
2:     char* loc = (char*) 0x1234abcd;
3:     loc[0] = 0x89;  // crashes here
4:     loc[1] = 0x12;
5:     loc[2] = 0xaa;

6:     return 0;
7: }
```

As a user-application, why this code crashes at line3 (not 2)?

Memory address space

```
1: int main() {
2:     char* loc = (char*) 0x1234abcd;
3:     loc[0] = 0x89;
4:     loc[1] = 0x12;
5:     loc[2] = 0xaa;

6:     return 0;
7: }
```

Address	Content
#ffffffff	8bits
...	
...	
#00000002	8bits
#00000001	8bits
#00000000	8bits

To run the code, we first need a memory address space, which is an abstraction of a 2-column table.

Code & Stack

```
1: int main() {
2:     char* loc = (char*) 0x1234abcd;
3:     loc[0] = 0x89;
4:     loc[1] = 0x12;
5:     loc[2] = 0xaa;

6:     return 0;
7: }
```

Specifically, we need two memory regions — code segment and stack segment.

Address	Content
...	...
application stack end	...
...	...
application stack start	...
...	...
application code end	...
...	...
application code start	...
...	...

Code segment

```
1: int main() {
2:     char* loc = (char*) 0x1234abcd;
3:     loc[0] = 0x89;
4:     loc[1] = 0x12;
5:     loc[2] = 0xaa;

6:     return 0;
7: }
```

compile

```
0000000100000f80 _main:
100000f80: 55
100000f81: 48 89 e5
100000f84: 31 c0
100000f86: c7 45 fc 00 00 00 00
100000f8d: b9 cd ab 34 12
100000f92: 48 89 4d f0
100000f96: 48 8b 4d f0
100000f9a: c6 01 89
100000f9d: 48 8b 4d f0
100000fa1: c6 41 01 12
100000fa5: 48 8b 4d f0
100000fa9: c6 41 02 aa
100000fad: 5d
100000fae: c3
```

put the binary executable into
The code segment

Address	Content
...	...
application stack end	...
...	...
application stack start	...
...	...
application code end	...
...	...
application code start	...
...	...

Stack segment

```
1: int main() {
2:     char* loc = (char*) 0x1234abcd;
3:     loc[0] = 0x89;
4:     loc[1] = 0x12;
5:     loc[2] = 0xaa;

6:     return 0;
7: }
```

Memory for main function local variable `loc`

Address	Content
...	...
application stack end	...
0xabcd 0003	...
0xabcd 0002	...
0xabcd 0001	...
0xabcd 0000	...
...	...
application stack start	...
...	...

Suppose &loc == 0xabcd 0000, meaning this local variable is stored at address 0xabcd 0000 in the stack.

Execution of line2

```
1: int main() {
2:     char* loc = (char*) 0x1234abcd;
3:     loc[0] = 0x89;
4:     loc[1] = 0x12;
5:     loc[2] = 0xaa;

6:     return 0;
7: }
```

Address	Content
...	...
application stack end	...
0xabcd 0003	0x 12
0xabcd 0002	0x 34
0xabcd 0001	0x ab
0xabcd 0000	0x cd
...	...
application stack start	...
...	...

Operating systems allow the user application to access memory addresses in its stack, so that modifying local variable `loc` will not cause fault.

Execution of line3

```
1: int main() {
2:     char* loc = (char*) 0x1234abcd;
3:     loc[0] = 0x89;
4:     loc[1] = 0x12;
5:     loc[2] = 0xaa;

6:     return 0;
7: }
```

The code will crash if 0x1234abcd is NOT within application code or stack segments.

Address	Content
...	...
application stack end	Access allowed
...	Access allowed
application stack start	Access allowed
...	...
application code end	Access allowed
...	Access allowed
application code start	Access allowed
...	...
0x1234 abcd	Access disallowed

Lesson1: the **minimal** requirement of program execution is **code & stack** segments in memory address space.

Correct line2

```
1: int main() {
2:     char* loc = (char*) malloc(3);
3:     loc[0] = 0x89;
4:     loc[1] = 0x12;
5:     loc[2] = 0xaa;

6:     return 0;
7: }
```

Malloc request a piece of memory (3 bytes in this case) from the OS. The newly allocated memory region is called heap segment.

Address	Content
...	...
application stack end	Access allowed
...	Access allowed
application stack start	Access allowed
...	...
application heap end	Access allowed
...	Access allowed
application heap start	Access allowed
...	...
application code end	Access allowed
...	Access allowed
application code start	Access allowed
...	...

Execution of line2

```
1: int main() {
2:     char* loc = (char*) malloc(3);
3:     loc[0] = 0x89;
4:     loc[1] = 0x12;
5:     loc[2] = 0xaa;

6:     return 0;
7: }
```

Suppose the return value of `malloc(3)` is `0x5555 6666`.

Address	Content
...	...
application stack end	Access allowed
0xabcd 0003	...
0xabcd 0002	...
0xabcd 0001	...
0xabcd 0000	...
application stack start	Access allowed
...	...
application heap end	…
0x5555 6668	Access allowed
0x5555 6667	Access allowed
0x5555 6666	Access allowed
application heap start	…
...	...

Execution of line2

```
1: int main() {
2:     char* loc = (char*) malloc(3);
3:     loc[0] = 0x89;
4:     loc[1] = 0x12;
5:     loc[2] = 0xaa;

6:     return 0;
7: }
```

Suppose `&loc == 0xabcd 0000`.

Address	Content
...	...
application stack end	Access allowed
0xabcd 0003	55
0xabcd 0002	55
0xabcd 0001	66
0xabcd 0000	66
application stack start	Access allowed
...	...
application heap end	…
0x5555 6668	Access allowed
0x5555 6667	Access allowed
0x5555 6666	Access allowed
application heap start	…
...	...

Execution of line3

```
1: int main() {
2:     char* loc = (char*) malloc(3);
3:     loc[0] = 0x89;
4:     loc[1] = 0x12;
5:     loc[2] = 0xaa;

6:     return 0;
7: }
```

Address	Content
...	...
application stack end	Access allowed
0xabcd 0003	55
0xabcd 0002	55
0xabcd 0001	66
0xabcd 0000	66
application stack start	Access allowed
...	...
application heap end	…
0x5555 6668	Access allowed
0x5555 6667	Access allowed
0x5555 6666	**0x 89**
application heap start	…
...	...

Execution of line4

```
1: int main() {
2:     char* loc = (char*) malloc(3);
3:     loc[0] = 0x89;
4:     loc[1] = 0x12;
5:     loc[2] = 0xaa;

6:     return 0;
7: }
```

Address	Content
...	...
application stack end	Access allowed
0xabcd 0003	55
0xabcd 0002	55
0xabcd 0001	66
0xabcd 0000	66
application stack start	Access allowed
...	...
application heap end	…
0x5555 6668	Access allowed
0x5555 6667	**0x 12**
0x5555 6666	0x 89
application heap start	…
...	...

Execution of line5

```
1: int main() {
2:     char* loc = (char*) malloc(3);
3:     loc[0] = 0x89;
4:     loc[1] = 0x12;
5:     loc[2] = 0xaa;

6:     return 0;
7: }
```

Address	Content
...	...
application stack end	Access allowed
0xabcd 0003	55
0xabcd 0002	55
0xabcd 0001	66
0xabcd 0000	66
application stack start	Access allowed
...	...
application heap end	…
0x5555 6668	**0x aa**
0x5555 6667	0x 12
0x5555 6666	0x 89
application heap start	…
...	...

Lesson2: when application requires dynamic memory allocation, OS will allocate the required amount in heap.

P0 Revisit, Q&A

EGOS demo

Question: how do operating systems run 2 user applications (multi-tasking)?

Note: we only talked about a single user application in all previous slides.

Multi-tasking (naïve)

- Suppose we have 2 user applications (**#1** and **#2**).
- The OS can run application **#1** first.

application**#1** stack end	…
…	…
application**#1** stack start	…
…	…
application**#1** code end	…
…	…
application**#1** code start	…
…	…

Multi-tasking (naïve)

- Suppose we have 2 user applications (**#1** and **#2**).
- The OS can run application **#1** first.
- And then run application **#2**.

…	…
application**#2** stack end	…
…	…
application**#2** stack start	…
…	…
application**#2** code end	…
…	…
application**#2** code start	…

Multi-tasking (naïve)

- Suppose we have 2 user applications (#1 and #2).
- The OS can run application #1 first.
- And then run application #2.
- This is called batch processing and it is the origin of operating systems (e.g., IBM 709 in 1960).
- OS was actually not computer code, but a real person called operator.

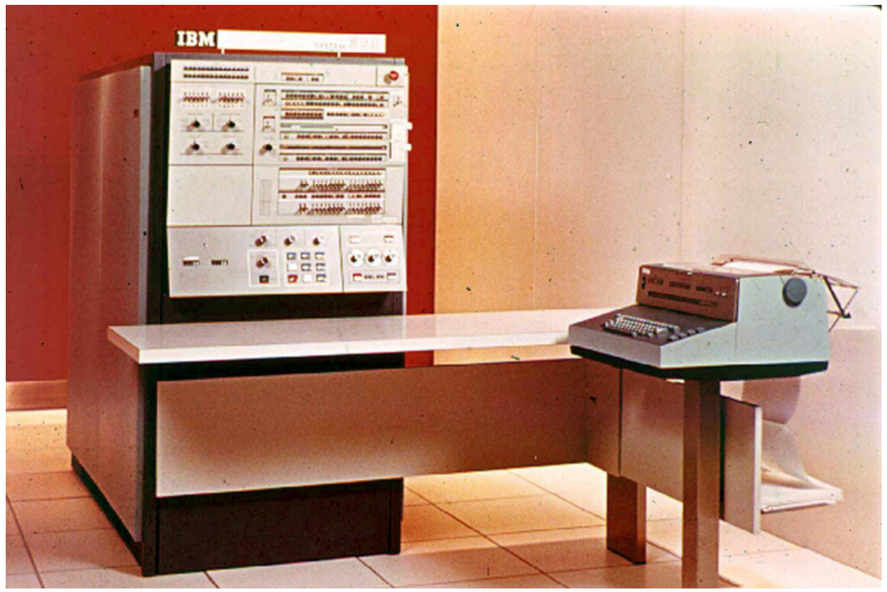
human operator

Human operator feeds application programs to the machine one-by-one.

*** Images from Computer History Museum: https://www.computerhistory.org/collections/catalog/102728984**

Multi-tasking (time-sharing)

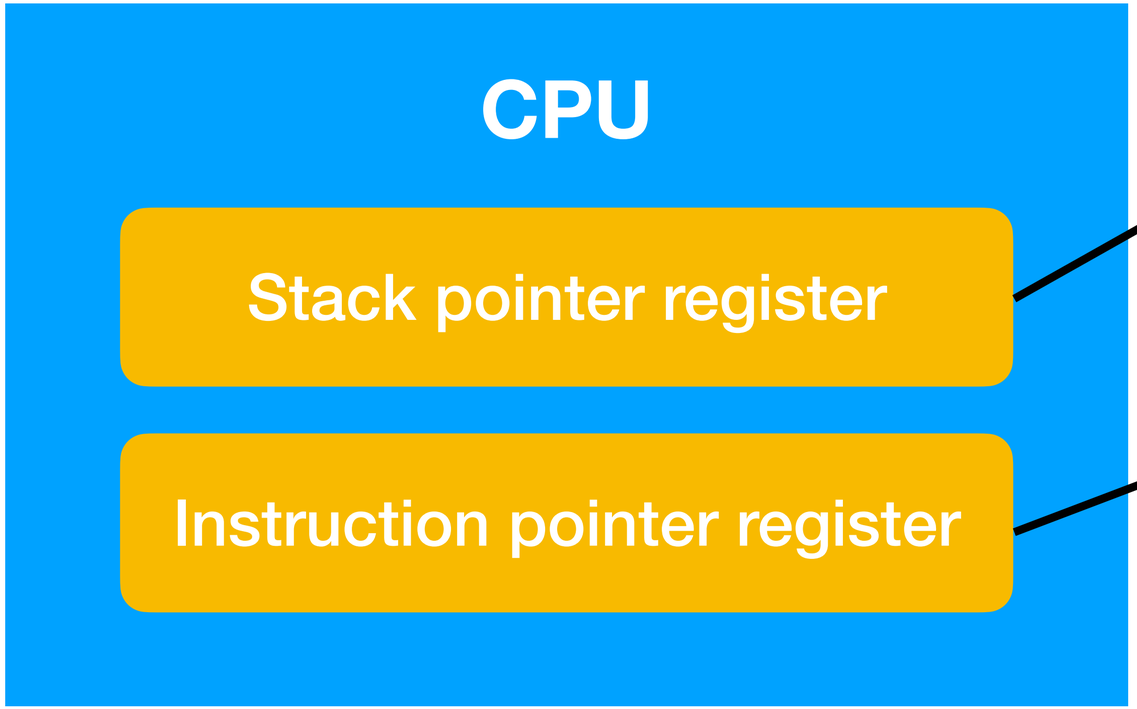
- Suppose we have 2 user applications (#1 and #2), **both of them** have code and stack segments in the memory.
- e.g., IBM 360 in 1967



application**#1** stack end	…
…	…
application**#1** stack start	…
…	…
application**#1** code end	…
…	…
application**#1** code start	…
…	…
application**#2** stack end	…
…	…
application**#2** stack start	…
…	…
application**#2** code end	…
…	…
application**#2** code start	…

*** Images from https://about.sourcegraph.com/blog/the-ibm-system-360-the-first-modular-general-purpose-computer/**

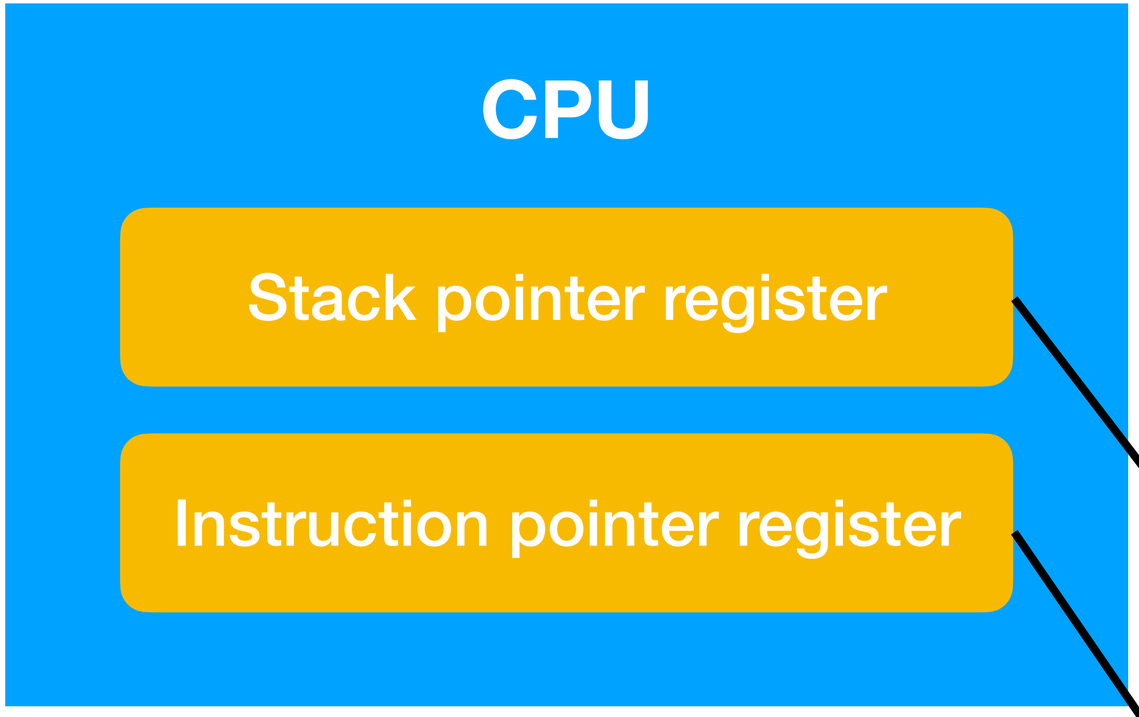
Running application #1



application#1 stack end	…
…	…
application#1 stack start	…
…	…
application#1 code end	…
…	…
application#1 code start	…
…	…
application#2 stack end	…
…	…
application#2 stack start	…
…	…
application#2 code end	…
…	…
application#2 code start	…

A CPU is **running application #1** if its stack pointer register and instruction pointer register hold memory addresses in the stack and code segment of application **#1.**

Running application #2



application#1 stack end	…
…	…
application#1 stack start	…
…	…
application#1 code end	…
…	…
application#1 code start	…
…	…
application#2 stack end	…
…	…
application#2 stack start	…
…	…
application#2 code end	…
…	…
application#2 code start	…

A CPU is **running application #2** if its stack pointer register and instruction pointer register hold memory addresses in the stack and code segment of application **#2**.

Lesson3: memory address space + stack pointer + instruction pointer = context; context defines which program the CPU is executing.

Lesson in next lecture: context-switch is switching stack pointer and instruction pointer registers to different stack and code segments.

Homework

- We have released P1 today and P1 is due on Oct 2.
- Implement the concepts of thread, context-switch and synchronization of threads.
- We will introduce more about these concepts in the next two lectures.