# CS 475
# Operating Systems

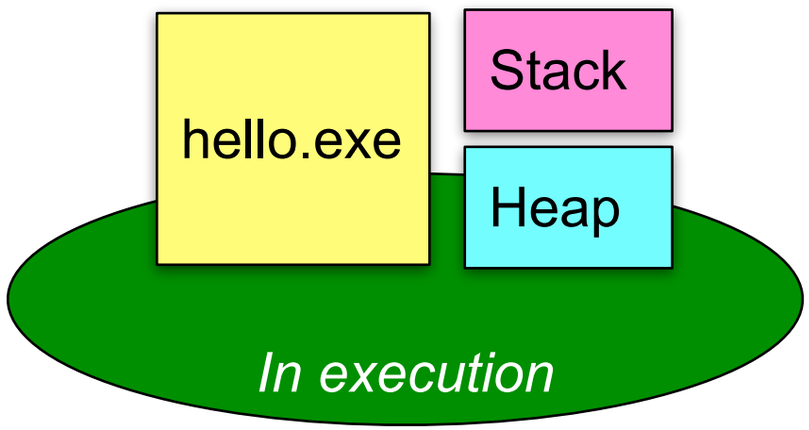

Department of Mathematics
and Computer Science

Lecture 3
Process Management

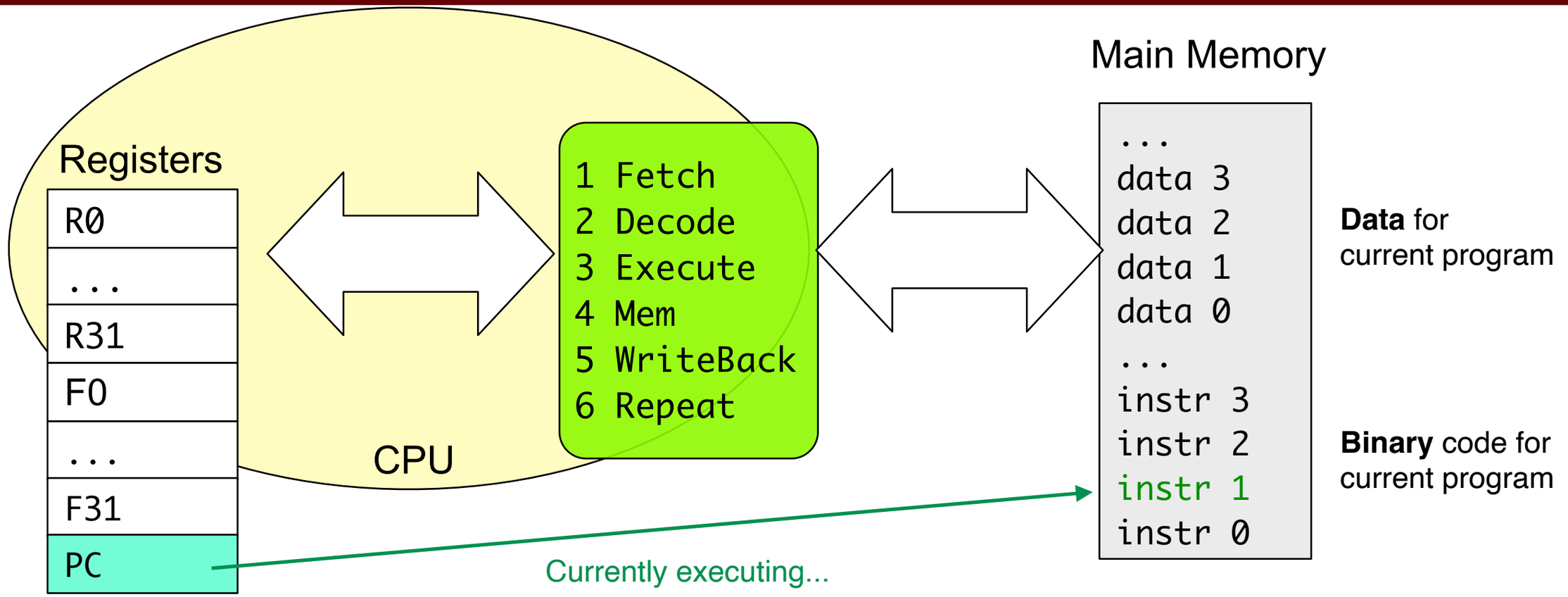
# Definition: Process vs. Program

- A *program* is a *passive* entity:
  - A compiled executable (binary) file stored somewhere on disk


- A *process* is an *active* entity:
  - A process is a program that is currently running
  - It is loaded into memory and executing its instructions (or waiting to)



# CPU Execution Cycle




- CPU execution sequence:
  1. Fetch Instruction at address referenced in PC register (program counter)
  2. Decode Instruction
  3. Execute (possibly using registers for intermediate storage)
  4. Write results (into registers or memory)
  5. PC = NextInstruction(PC)
  6. Repeat

# Goals for This Lecture

- Motivation: Concurrency
- Address Space
- Processes and the PCB
- Process Management in C
  - Creation: fork()
  - Basic Synchronization: wait()
  - Running another program: exec()

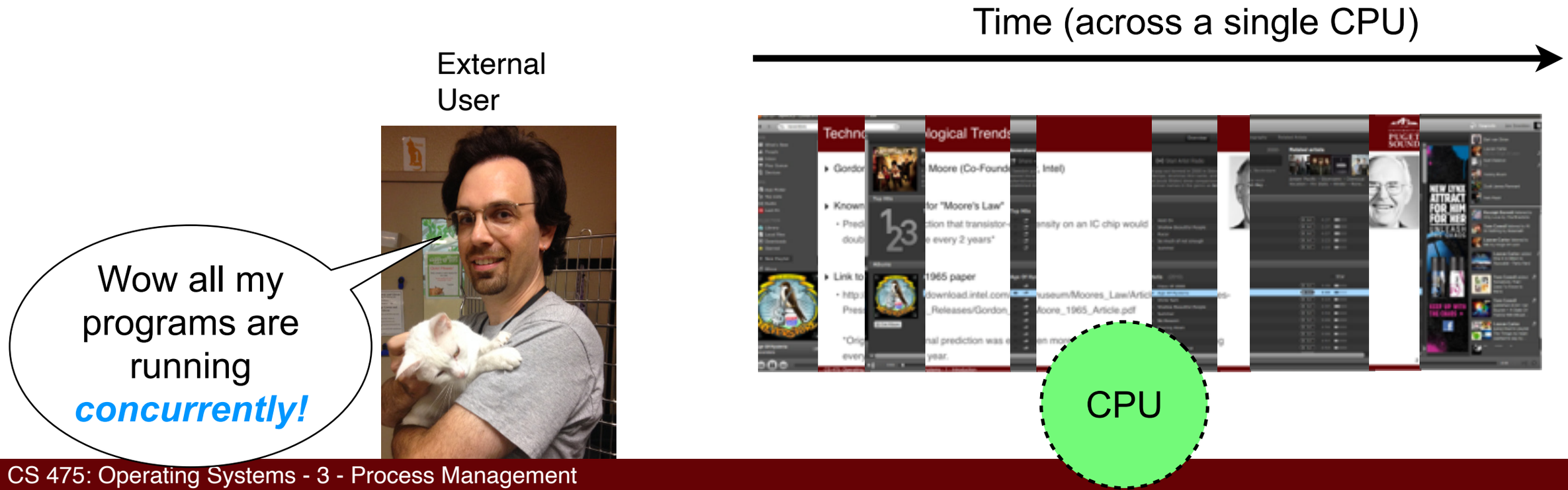
# "Job Concurrency Increases CPU Utilization"

- In the 1950's, batch processing ruled the computing world!!!!
  - A computer runs **one job** at a time
    - IBM 701, MS-DOS, early Macintosh
  - Great for OS developers
    - Simplified OS design if we didn't have to worry about multiple programs running!

- 1960's-70's: Timesharing to Make User More Efficient
  - Humans are becoming more expensive. Keep them busy by multitasking!
  - Execute > 1 job at a time
    - *But 1 CPU meant that only 1 job can be really be running at any instant*
    - *(Can we fake it?)*

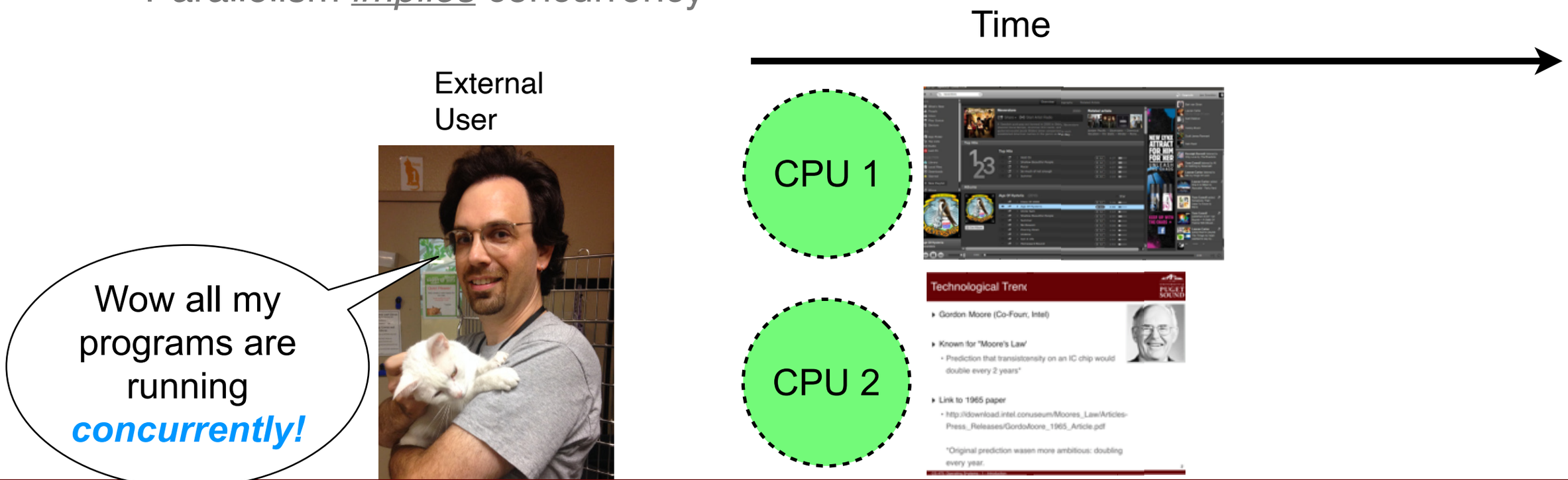
# Definition: Concurrency vs. Parallelism

- Concurrency vs. Parallelism
  - **Concurrency** is *giving the perception* of simultaneous job execution
    - You can have one CPU multiplexing between different programs
    - Or, you can have multiple CPUs all running different programs
  - Parallelism is *actually performing* simultaneous job execution
    - You can have multiple CPUs all running different programs
    - Parallelism *implies* concurrency



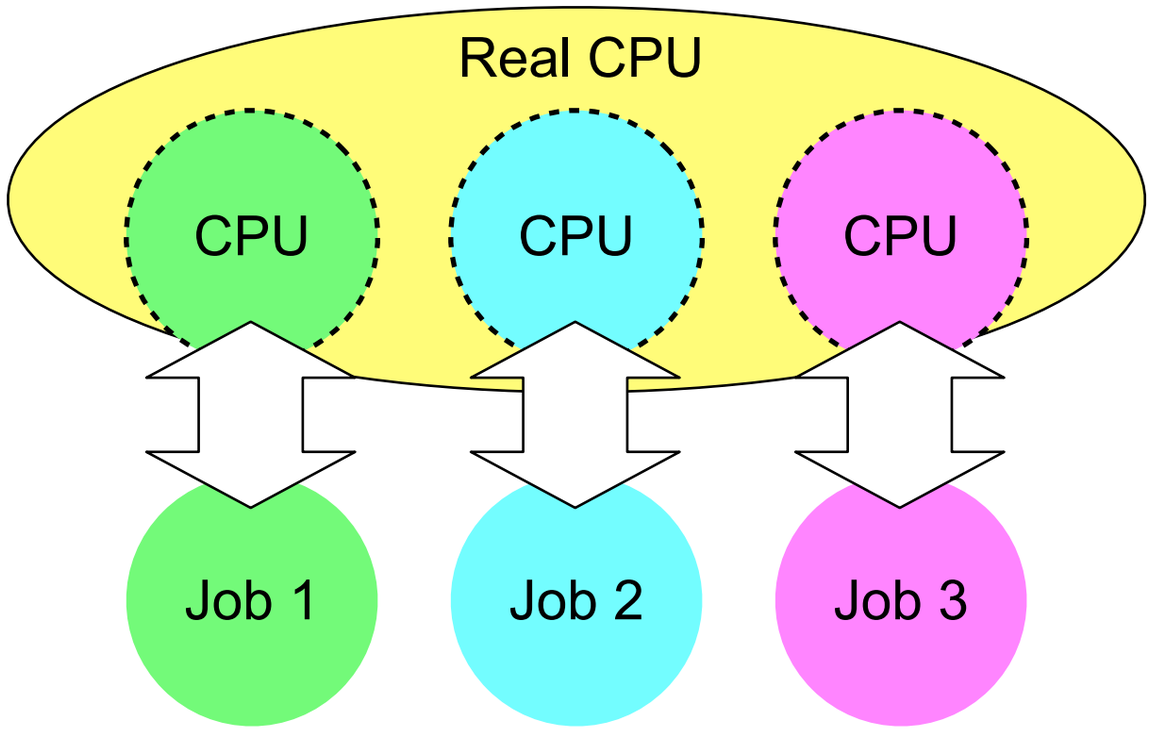
# Definition: Concurrency vs. Parallelism

- Concurrency vs. Parallelism
  - **Concurrency** is *giving the perception* of simultaneous job execution
    - You can have one CPU multiplexing between different programs
    - Or, you can have multiple CPUs all running different programs
  - **Parallelism** is *actually performing* simultaneous job execution
    - You can have multiple CPUs all running different programs
    - Parallelism *implies* concurrency



# Policy: CPU Virtualization

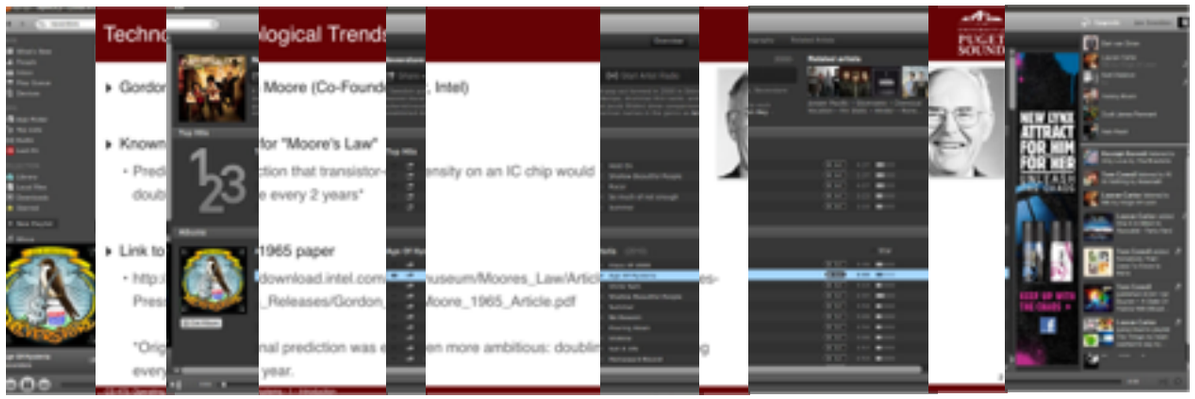
- *Policy: We want to concurrently execute multiple jobs on 1 CPU*
  - This policy is called *"CPU Virtualization"*
  - We want each job to *think* they have ownership of its own CPU
    - (Can you imagine the alternative? You'd have to request allocation of some usage time before you can run your program!)



# Mechanism? Context Switching

## Mechanism: Context Switching

- **Save** the *current state* of a job to main memory
- **Load** the state of another job state from main memory



Time (across a single CPU)

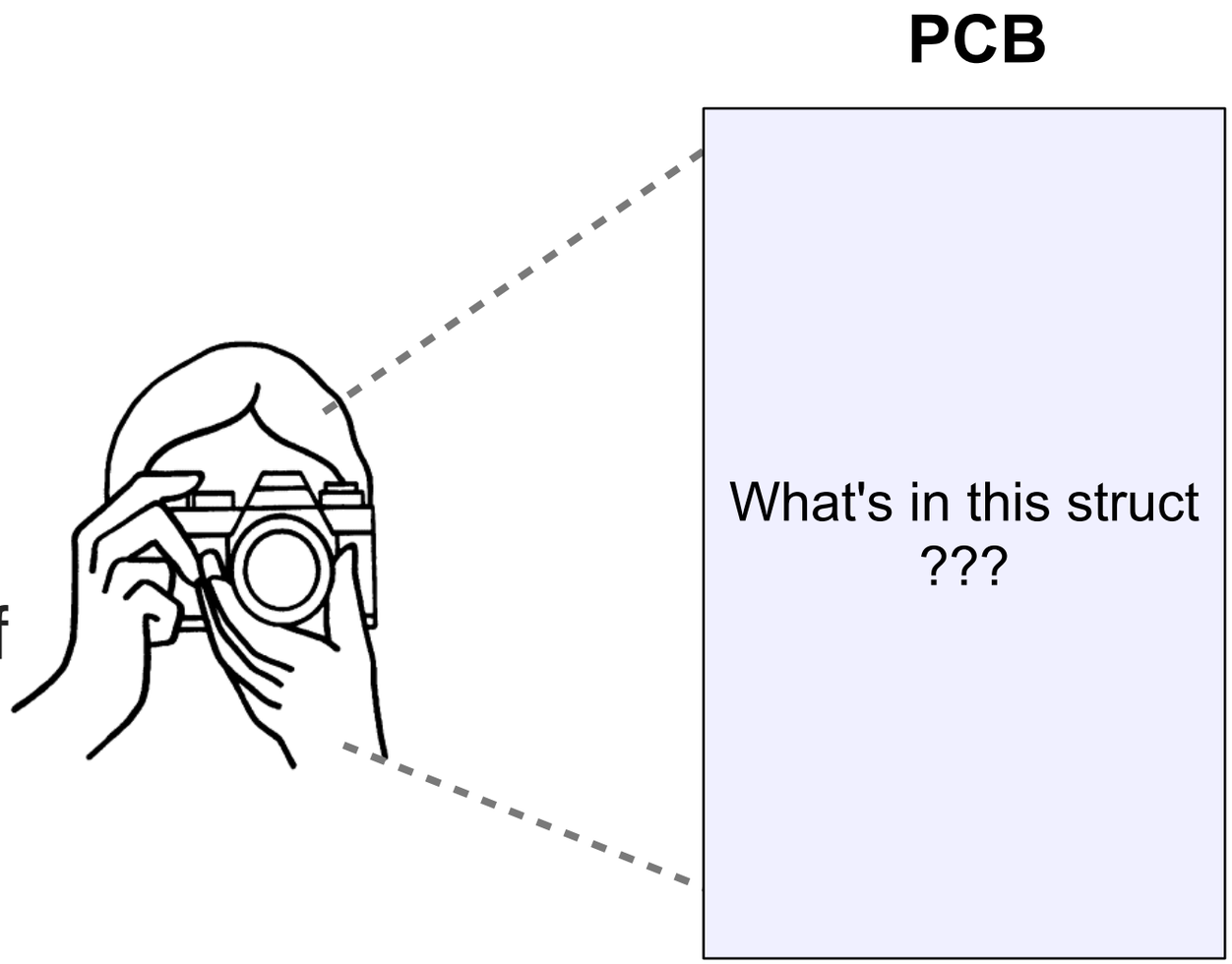
- To pause and play, we need to a way to *store the state* of any process.

# Process Control Block (PCB)

- Think games: What do we need in a *"save file"* for each process?

- The OS represents each process in a *Process Control Block (PCB)*

- We need to capture the snapshot of a process at arbitrary points in time
  - *We'll use a struct*
  - *What info might a PCB struct need to store?*



# Process Control Block (PCB)

- What's in a PCB Struct?

**PCB Struct (One per process)**

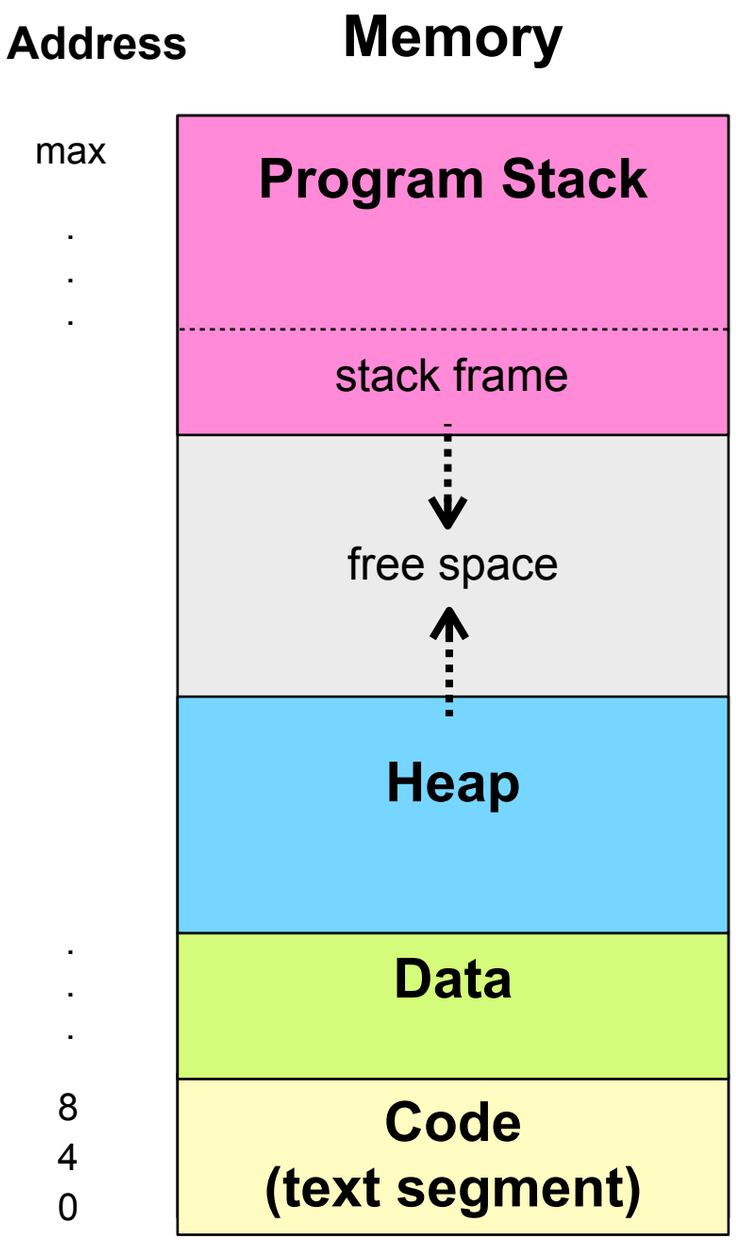
- pid (unique ID in the OS)
- uid (who created this proc?)
- Execution state (ready? running? done?)
- CPU register contents
- List of open files
- List of open sockets

**Address Space Pointers** *?????*
  - Base Pointer (bp) *?????*
  - Stack Pointer (sp) *?????*
  - Heap Pointer (brk) *?????*
  - Program Counter (pc) *?????*

# Goals for This Lecture

- Motivation: Concurrency
- Address Space
- Processes and the PCB
- Process Management in C
  - Creation: fork()
  - Basic Synchronization: wait()
  - Running another program: exec()

# Address Space (One per Process!)




**Stack: Keeps track of function calls**
Local variables, parameters, return address are stored here

A stack frame contains all the local variables and input arguments belonging to a function call.

**Heap:** Dynamically allocated data stored here (*e.g.*, `malloc()` in C; `new` in Java and C++)

**Data:** Global and static variables stored here

**Code:** Compiled binary sits in here.

Assumes a 32-bit architecture, so each CPU word is 4 bytes (= 32bits)

# Process Control Block (PCB)

- What's in a PCB Struct?

**PCB Struct (One per process)**

- pid (unique ID in the OS)
- uid (who created this proc?)
- Execution state (ready? running? done?)
- CPU register contents
- List of open files
- List of open sockets

**Address Space Pointers** *?????*
  - Base Pointer (bp) *?????*
  - Stack Pointer (sp) *?????*
  - Heap Pointer (brk) *?????*
  - Program Counter (pc) *?????*

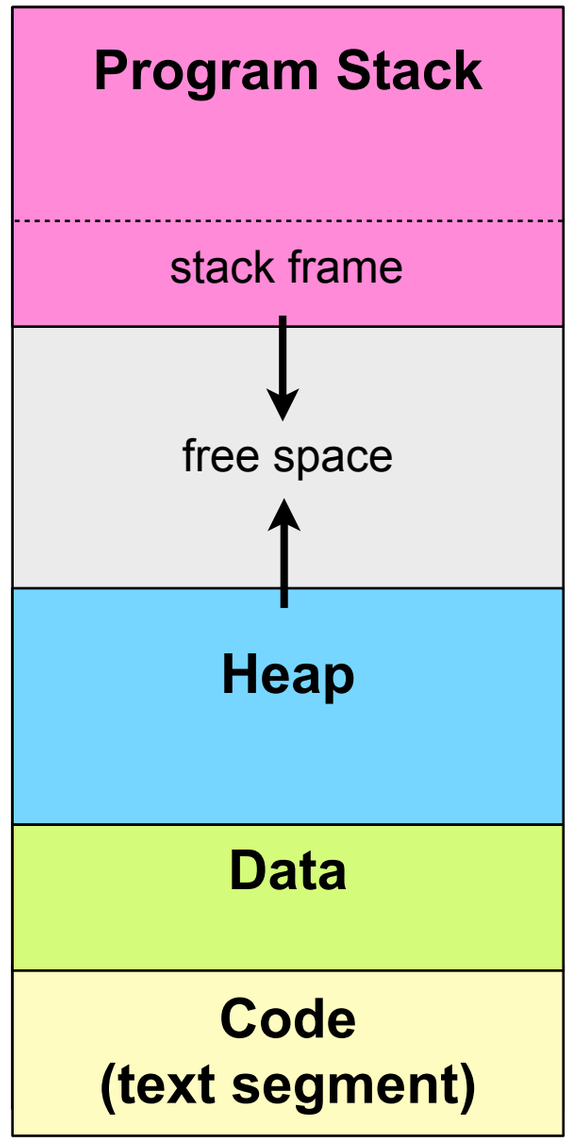
# Process Control Block (PCB)

- What's in a PCB Struct?

**PCB Struct (One per process)**

- pid (unique ID in the OS)
- uid (who created this proc?)
- Execution state (ready? running? done?)
- CPU register contents
- List of open files
- List of open sockets

**Address Space Pointers**
  - Base Pointer (bp)
  - Stack Pointer (sp)
  - Heap Pointer (brk)
  - Program Counter (pc)



# Example: Intel x86 Architecture (32-Bit)

- x86 register names (32-bit architecture)

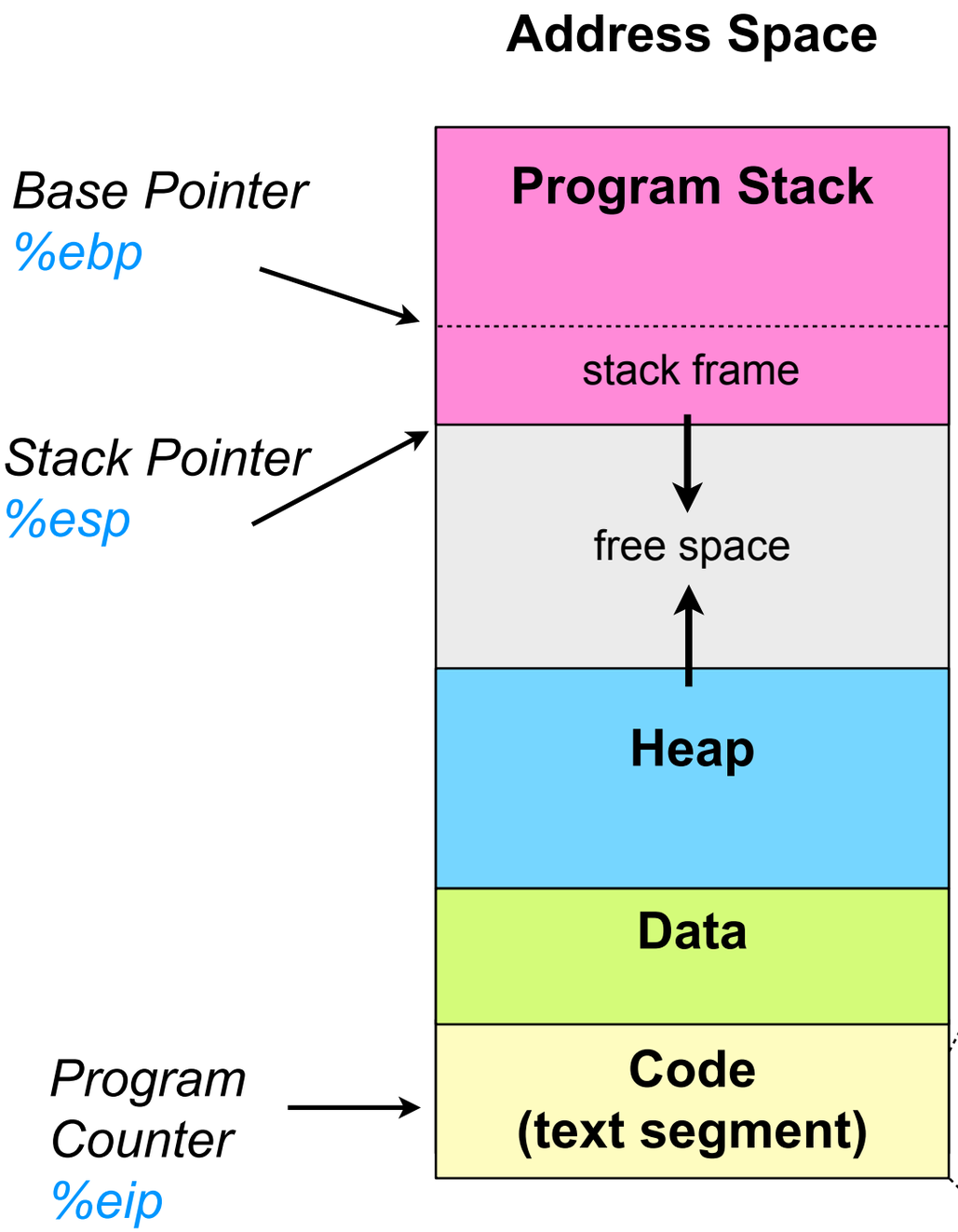
General Purpose Registers

```
%eax (usually holds a function's return value)
%ecx
%edx
%ebx
%esi
%edi
%esp (pointer to top of stack)
%ebp (pointer to stack frame base address)
```

Special Purpose

```
%eip (instruction pointer or Program Counter)
```

# Stack Management: x86 and C Example



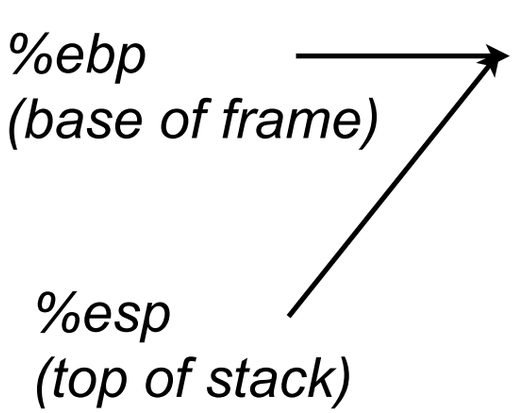

In code.c

```c
int main() {
   int a = 5, b = 10;
   sum(a,b);
}

int sum(int x, int y) {
   int z = x + y;
   return z;
}
```

$ gcc -S -o sum code.c

```
main:
  pushl $5
  pushl $10
  pushl 8(%eip)
  jmp   sum
  print %eax
sum:
  pushl %ebp
  movl  %esp, %ebp
  movl  12(%ebp), %eax
  addl  8(%ebp), %eax
  popl  %ebp
  popl  %esi
  jmp   (%esi)
```

# Stack Management: x86 Example




free space

```
  jmp    (%esi)
  popl   %esi
  popl   %ebp
  addl   8(%ebp), %eax
  movl   12(%ebp), %eax
  movl   %esp, %ebp
  pushl  %ebp
sum:
  print  %eax
  jmp    sum
  pushl  8(%eip)
  pushl  $10
  pushl  $5
main:
```

**Code**



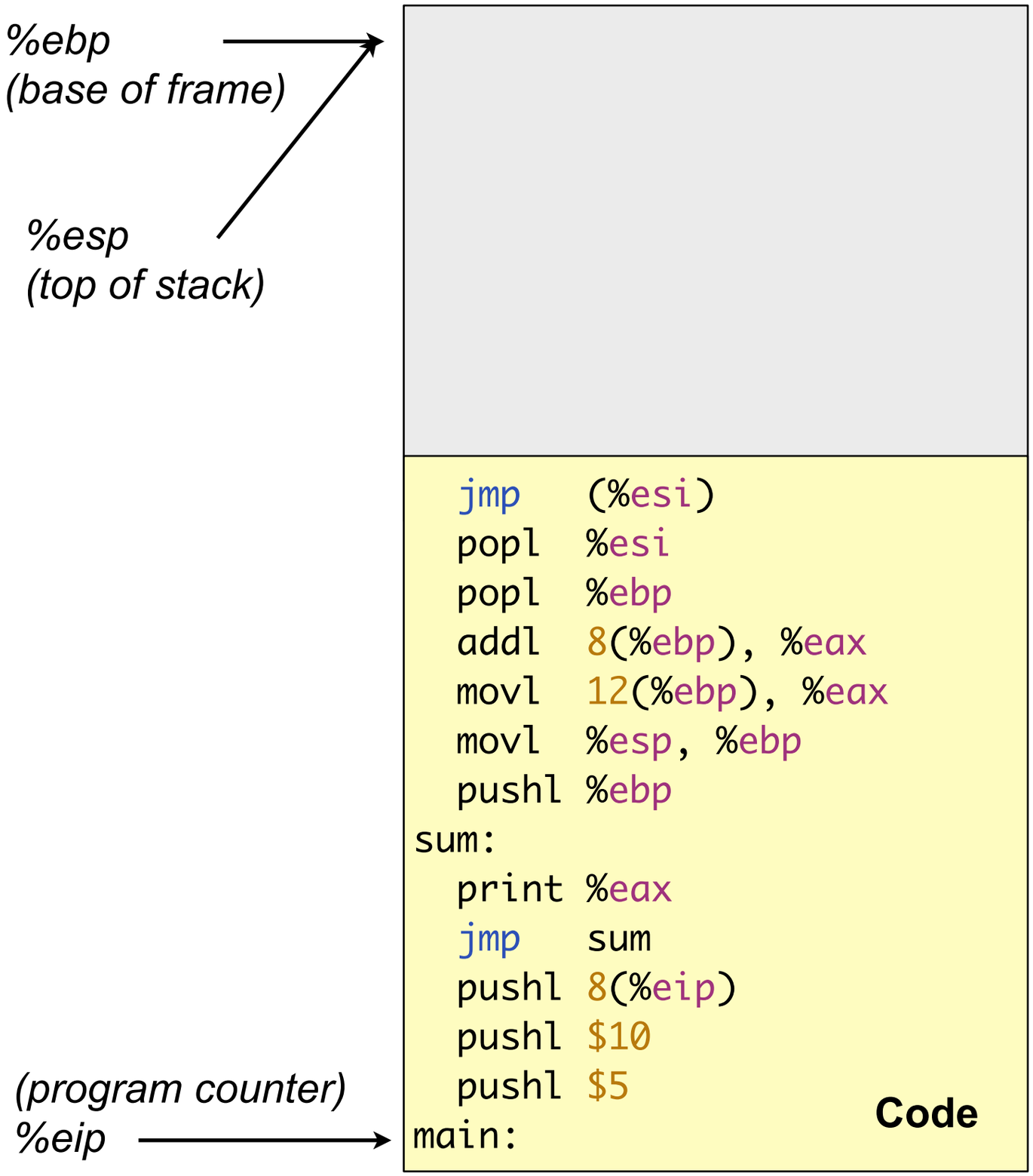

```
maxAddr
  .
  .
  .
  .
  .
  .
36
32
28
24
20
16
12
8
4
0
```

**Read this:**

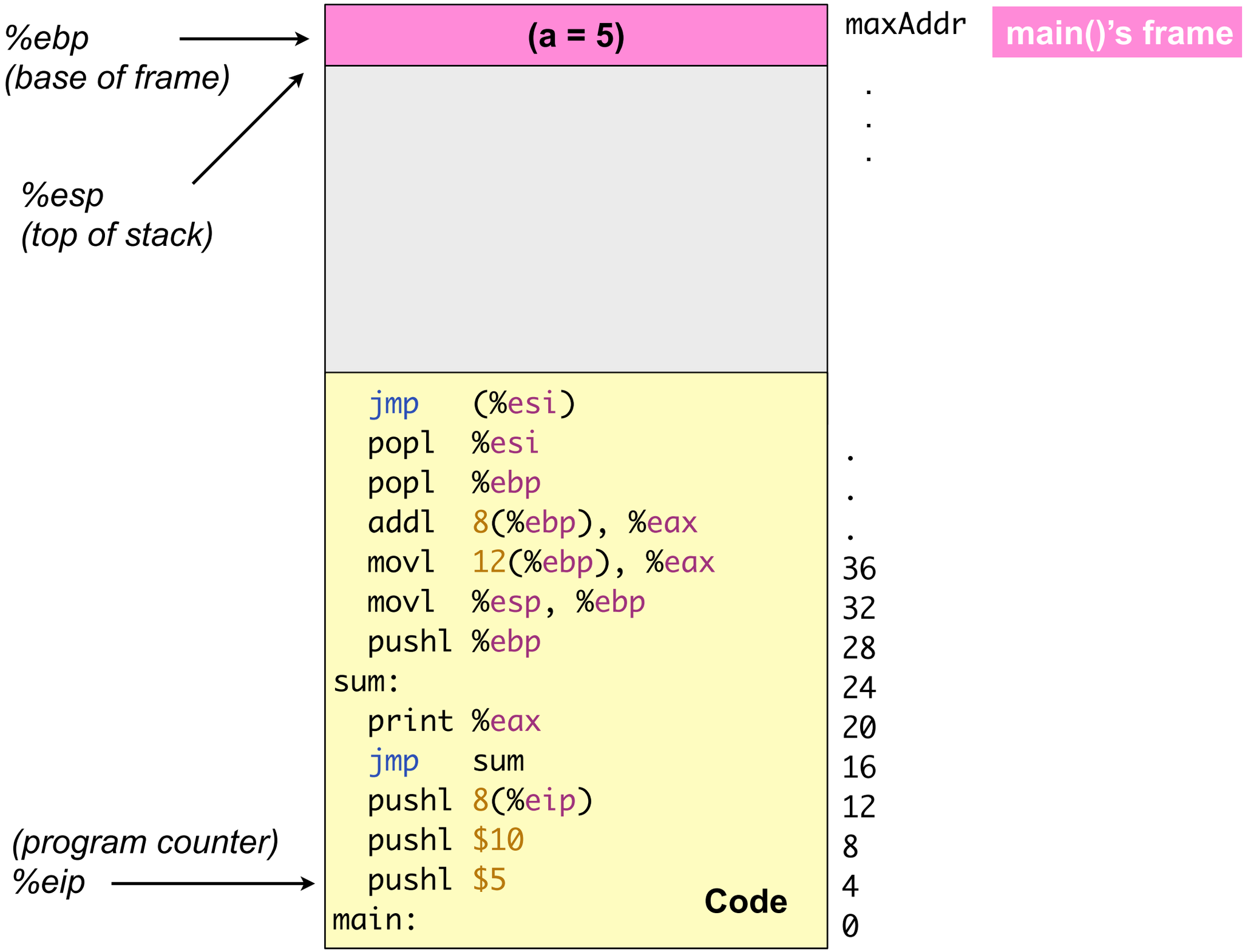
* The code is stored bottom-up since addresses increase upward in the fig.

* In reality, code would be in binary, not assembly.

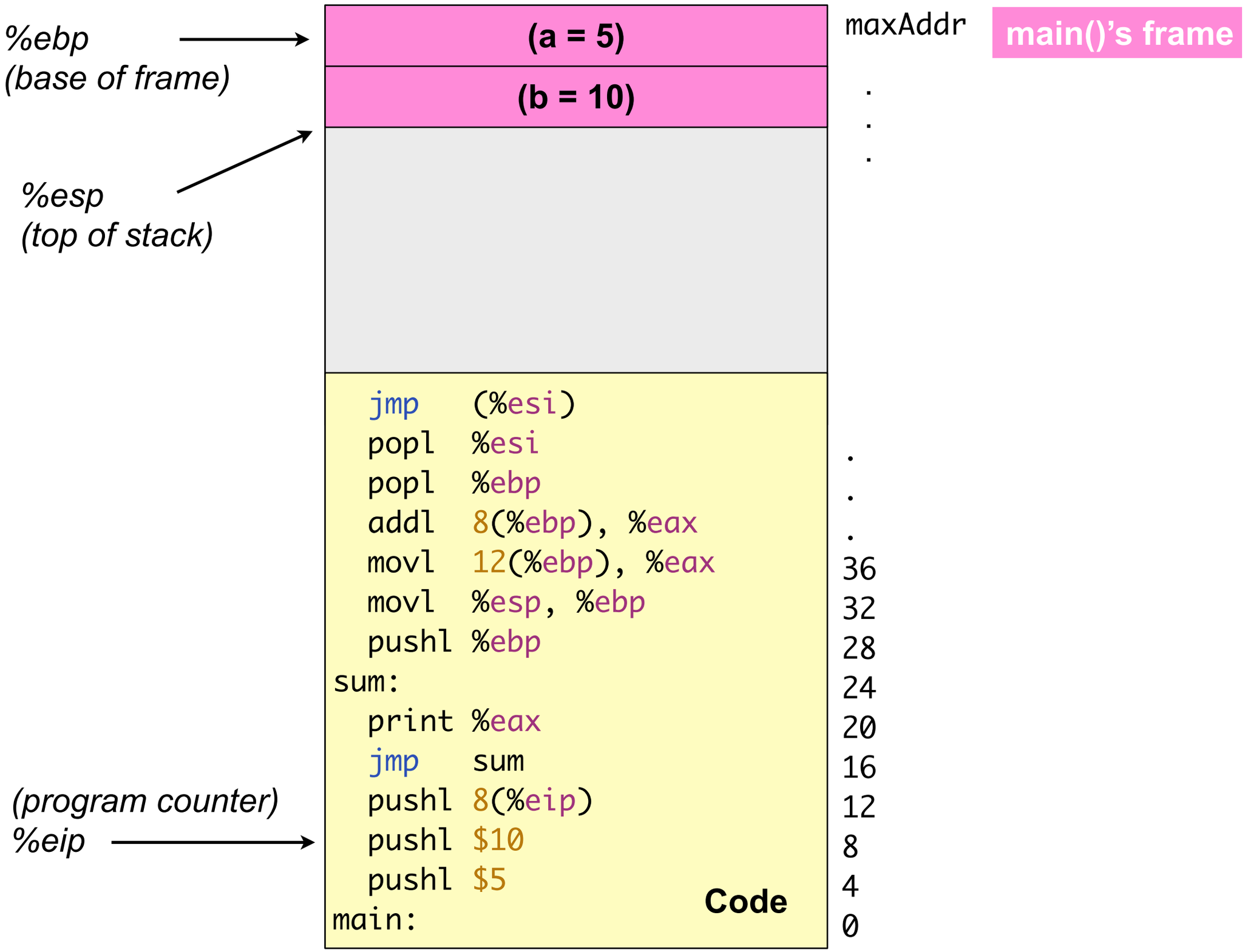
# x86 Example (1)



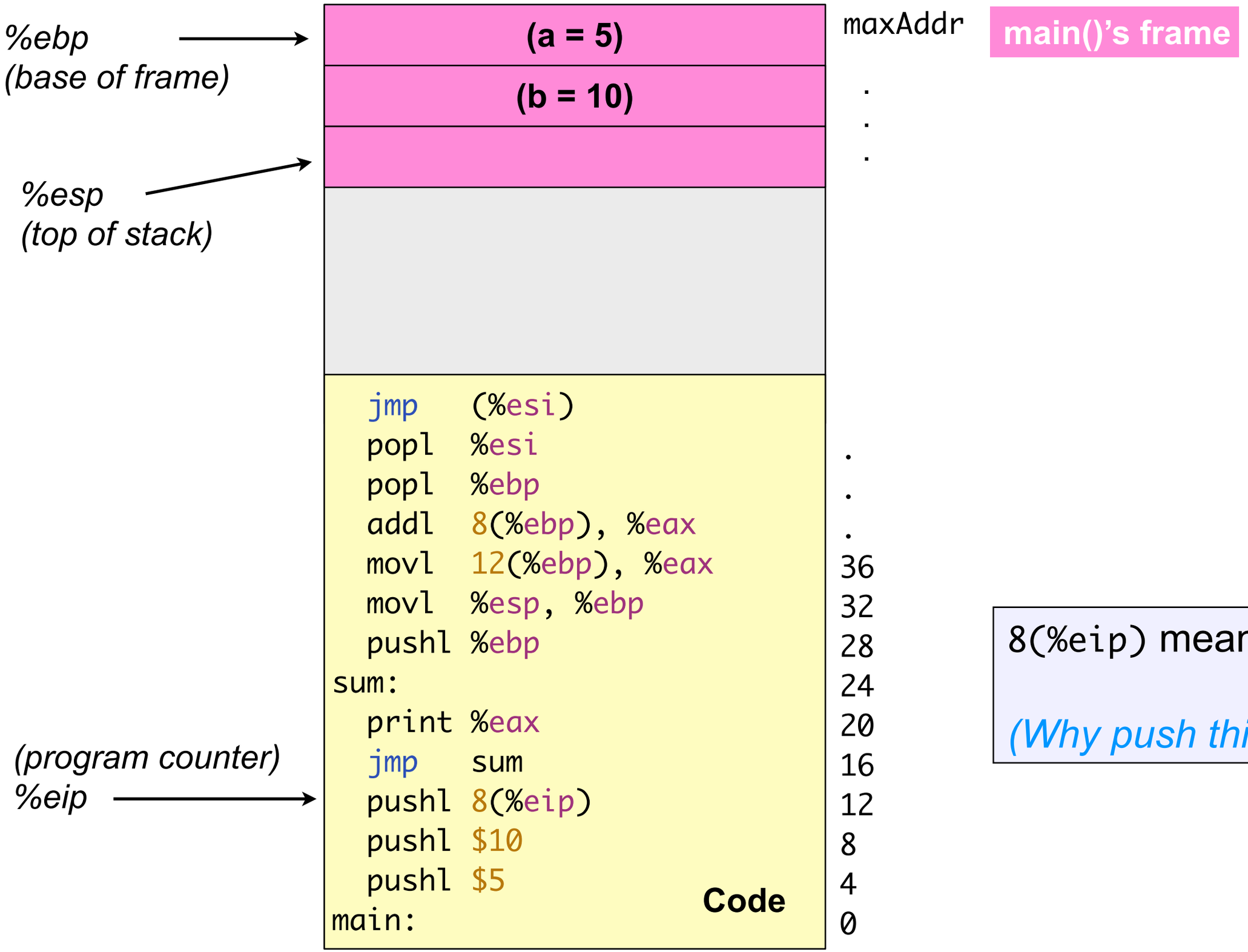
# x86 Example (2)



# x86 Example (3)



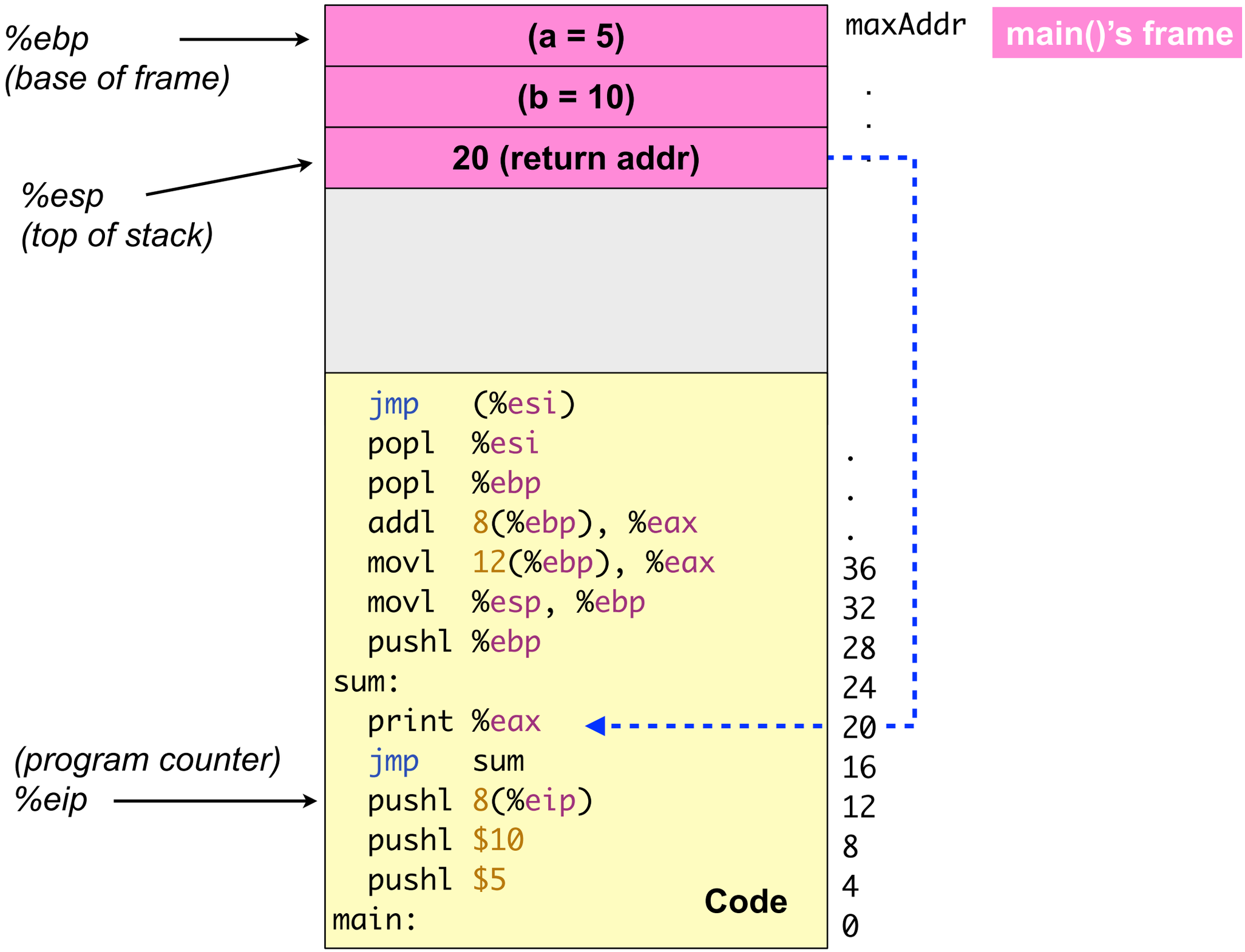
# x86 Example (4)



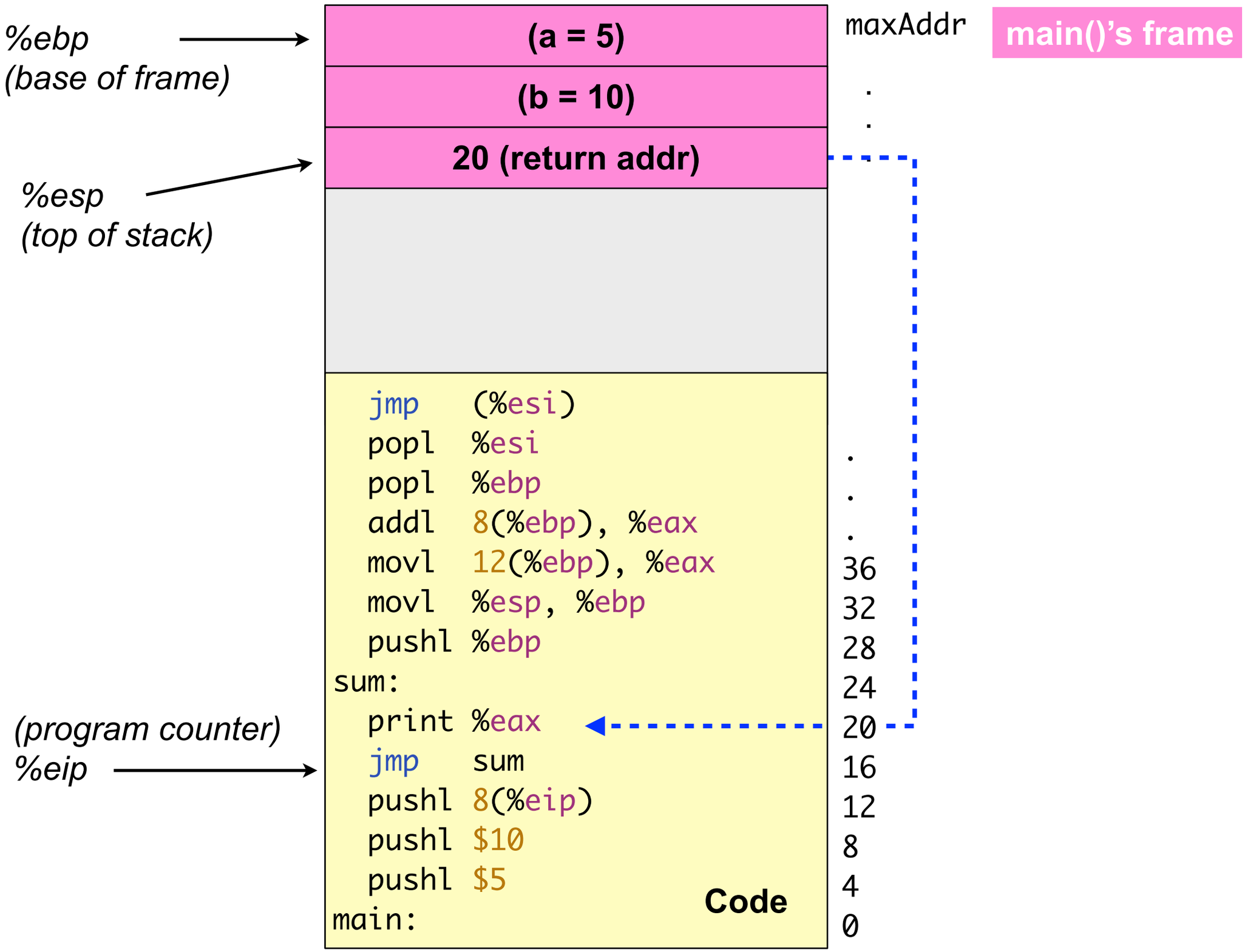

8(%eip) means %eip + 8

*(Why push this address?)*

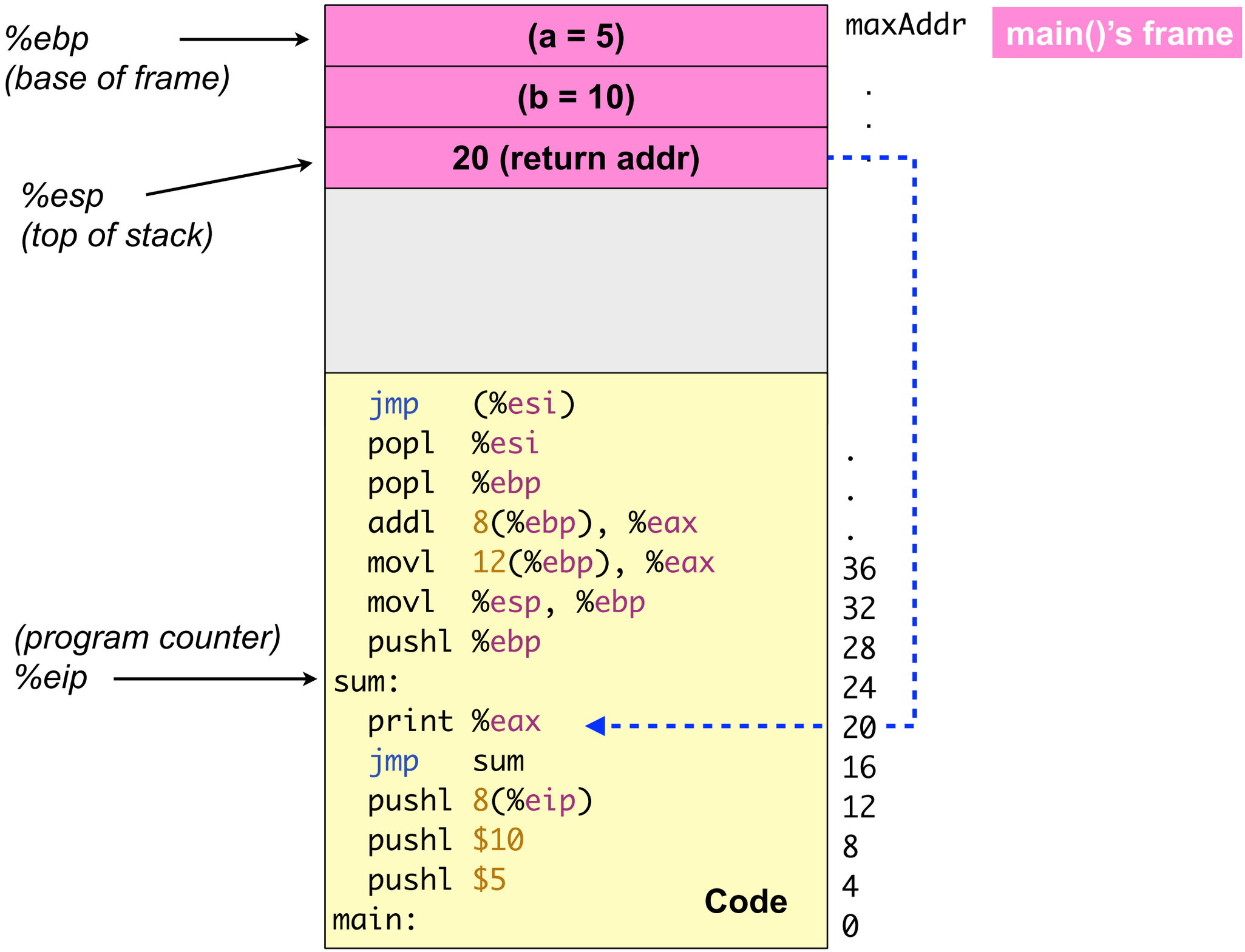
# x86 Example (5)



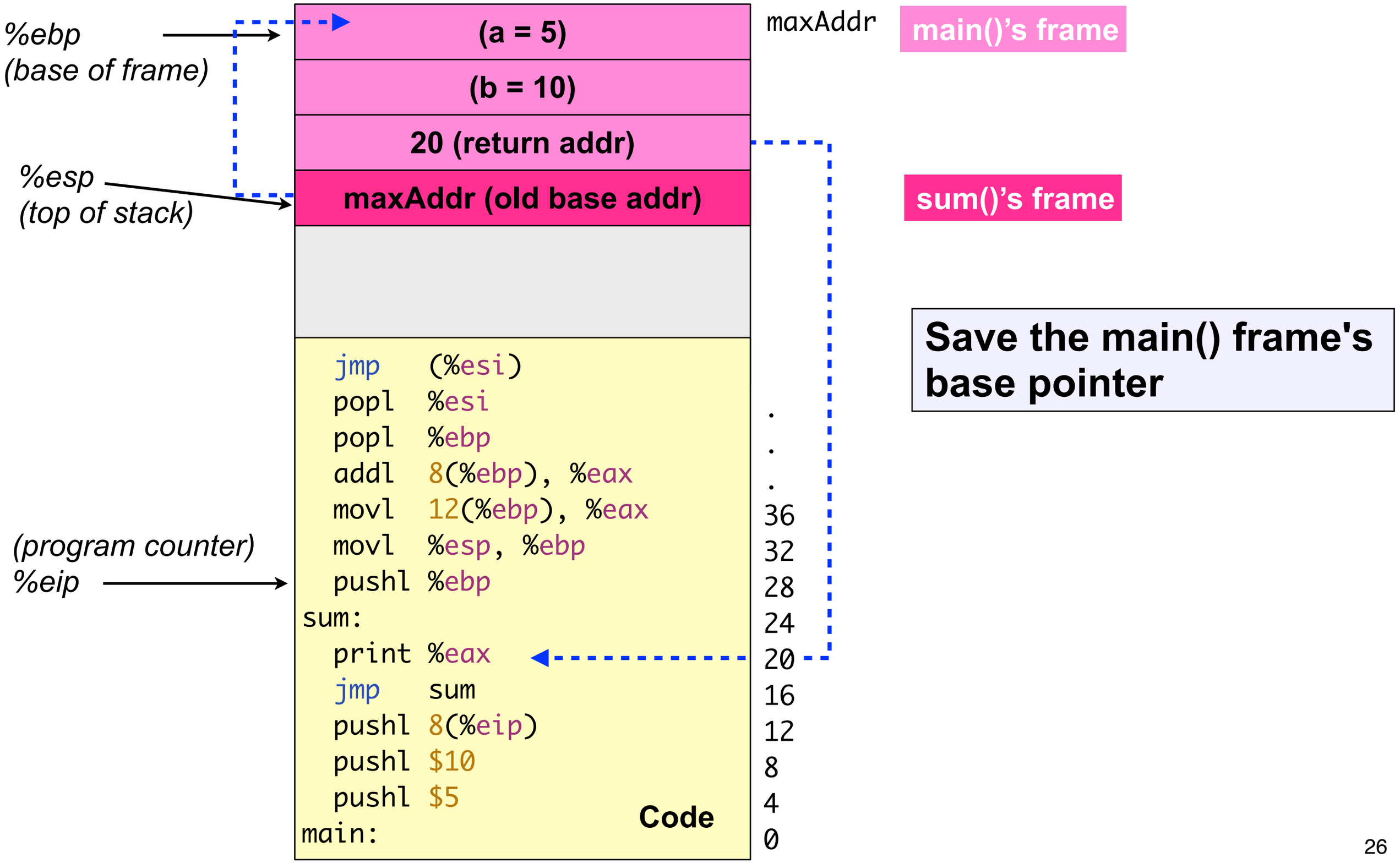
# x86 Example (6)



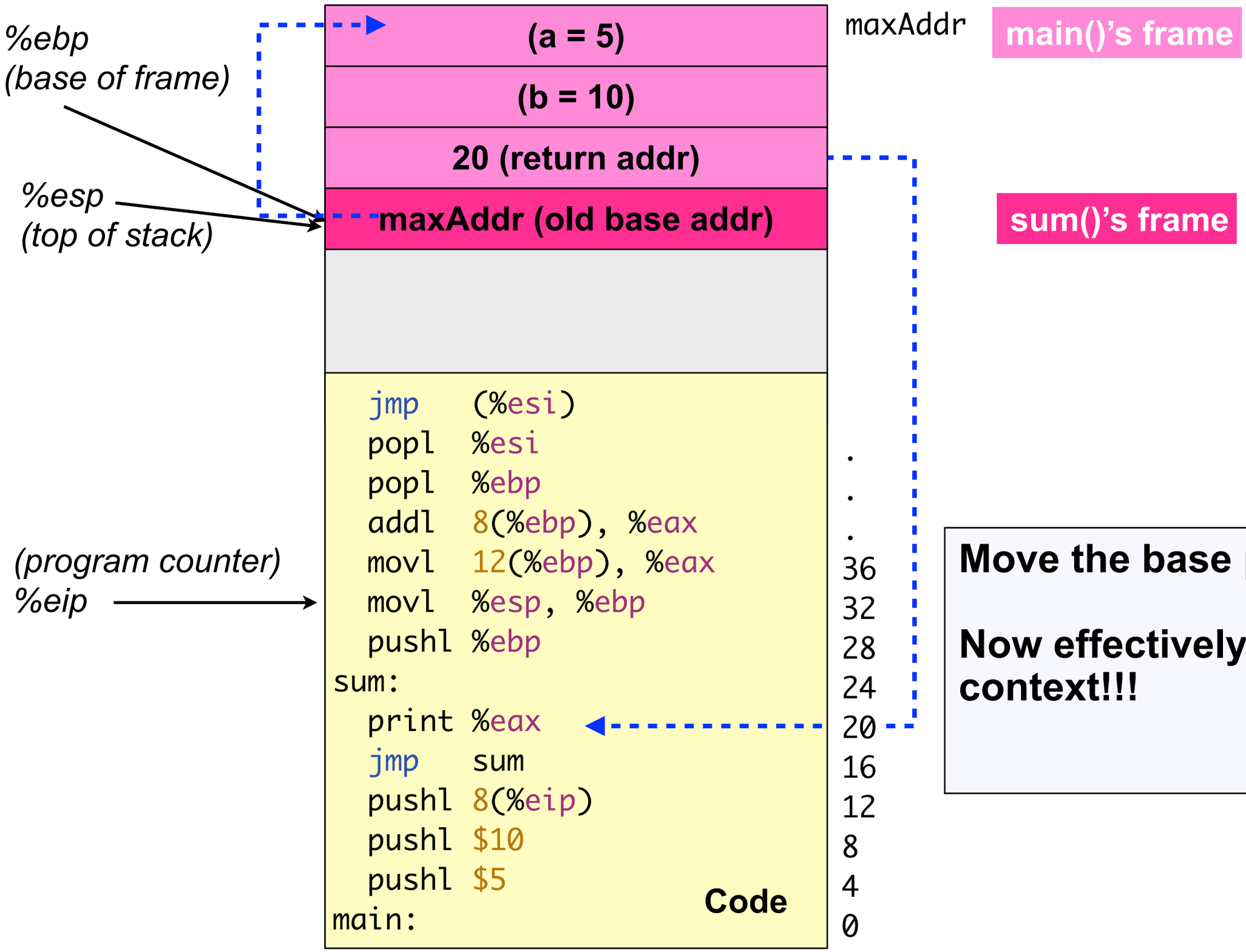
# x86 Example (7)



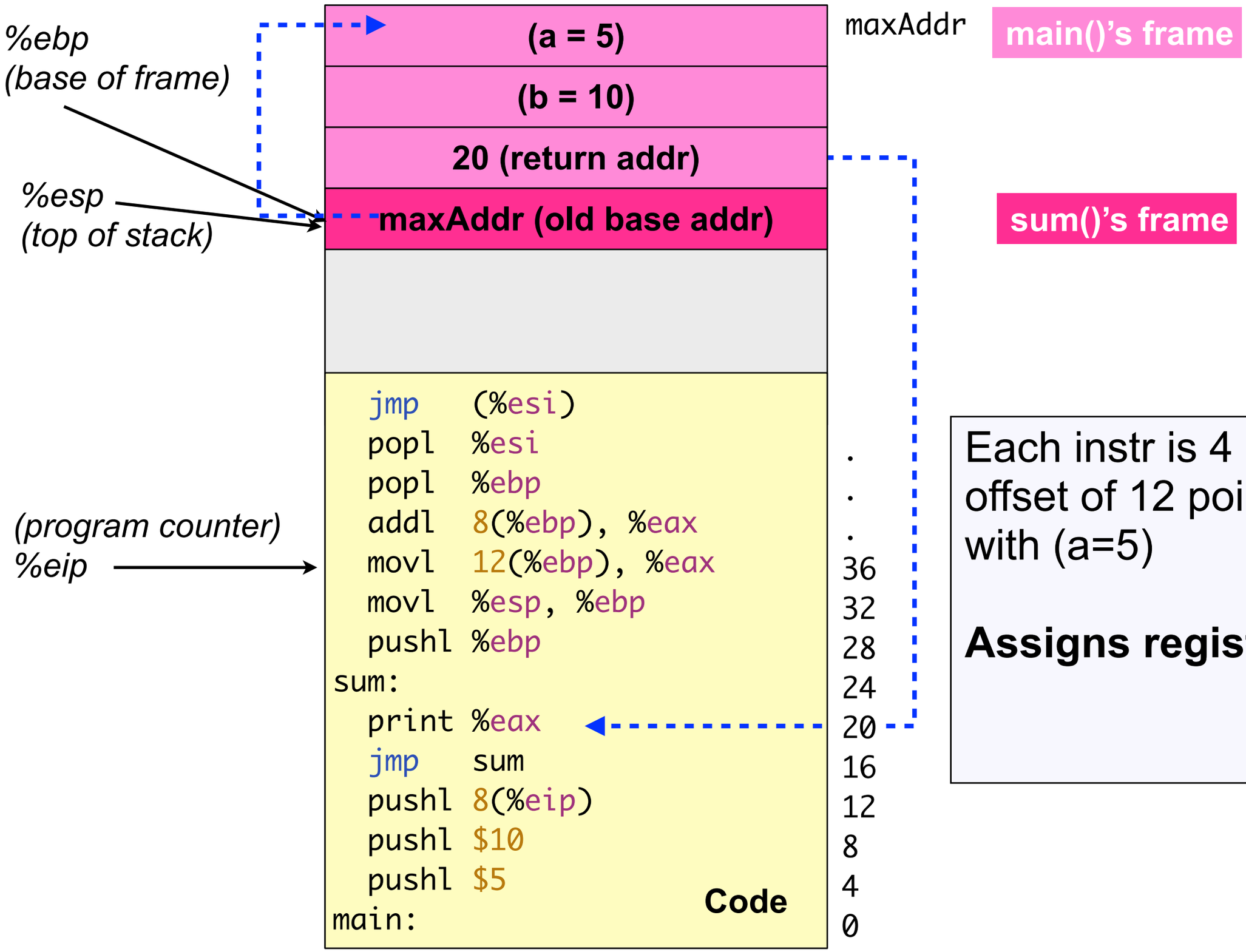
# x86 Example (8)



# x86 Example (9)



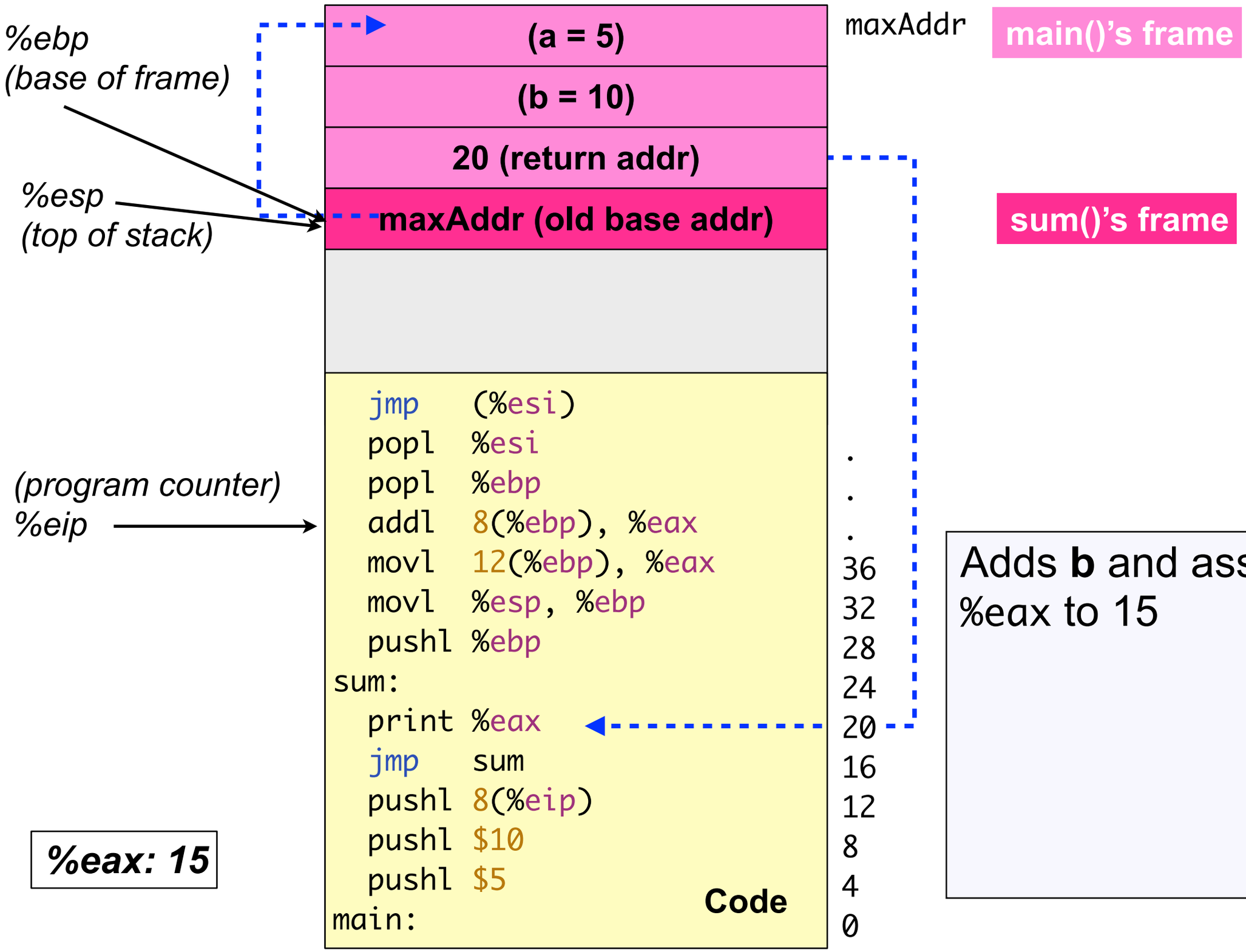
# x86 Example (10)



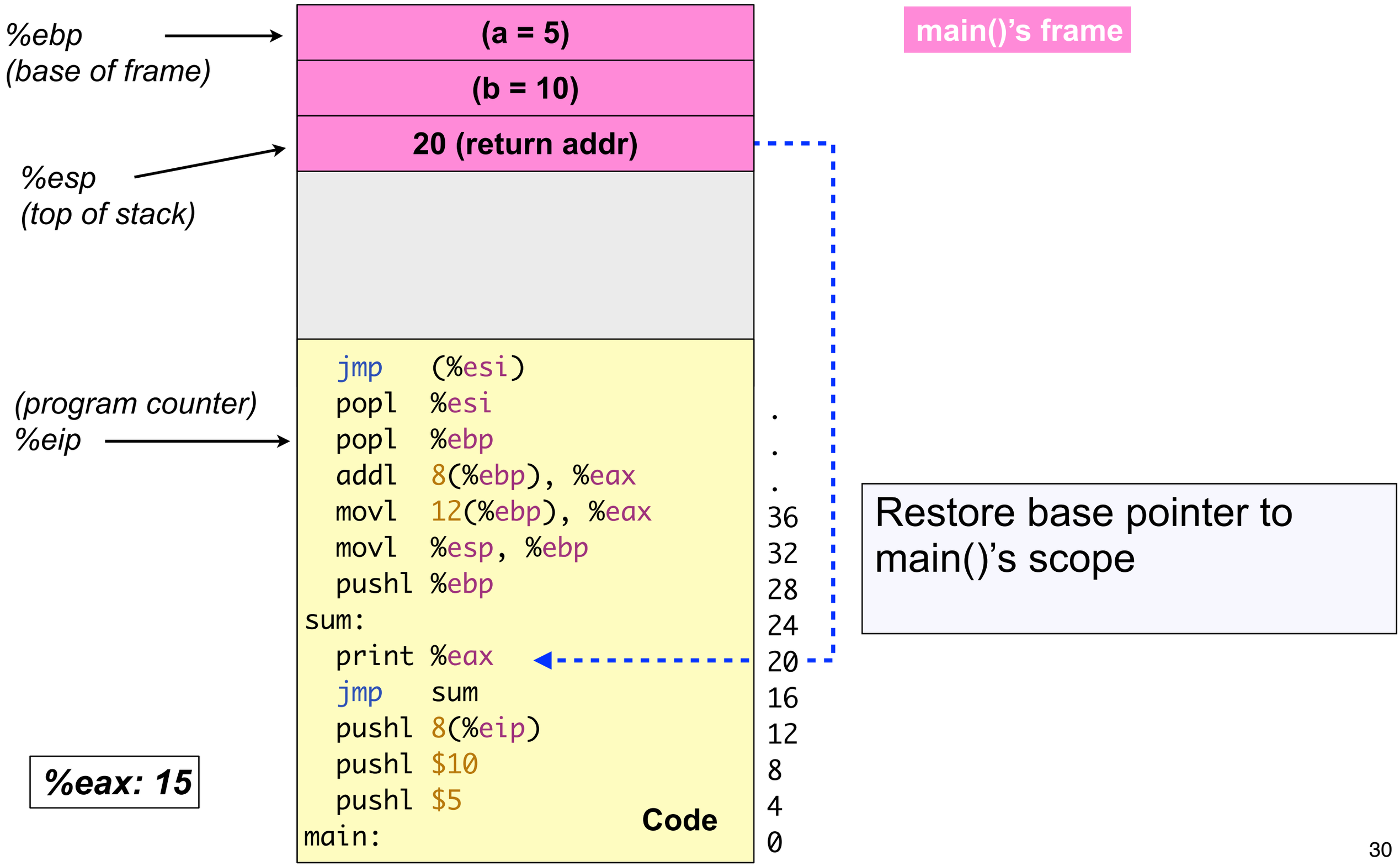

Each instr is 4 bytes, so an offset of 12 points to location with (a=5)

**Assigns register** %eax **to 5**

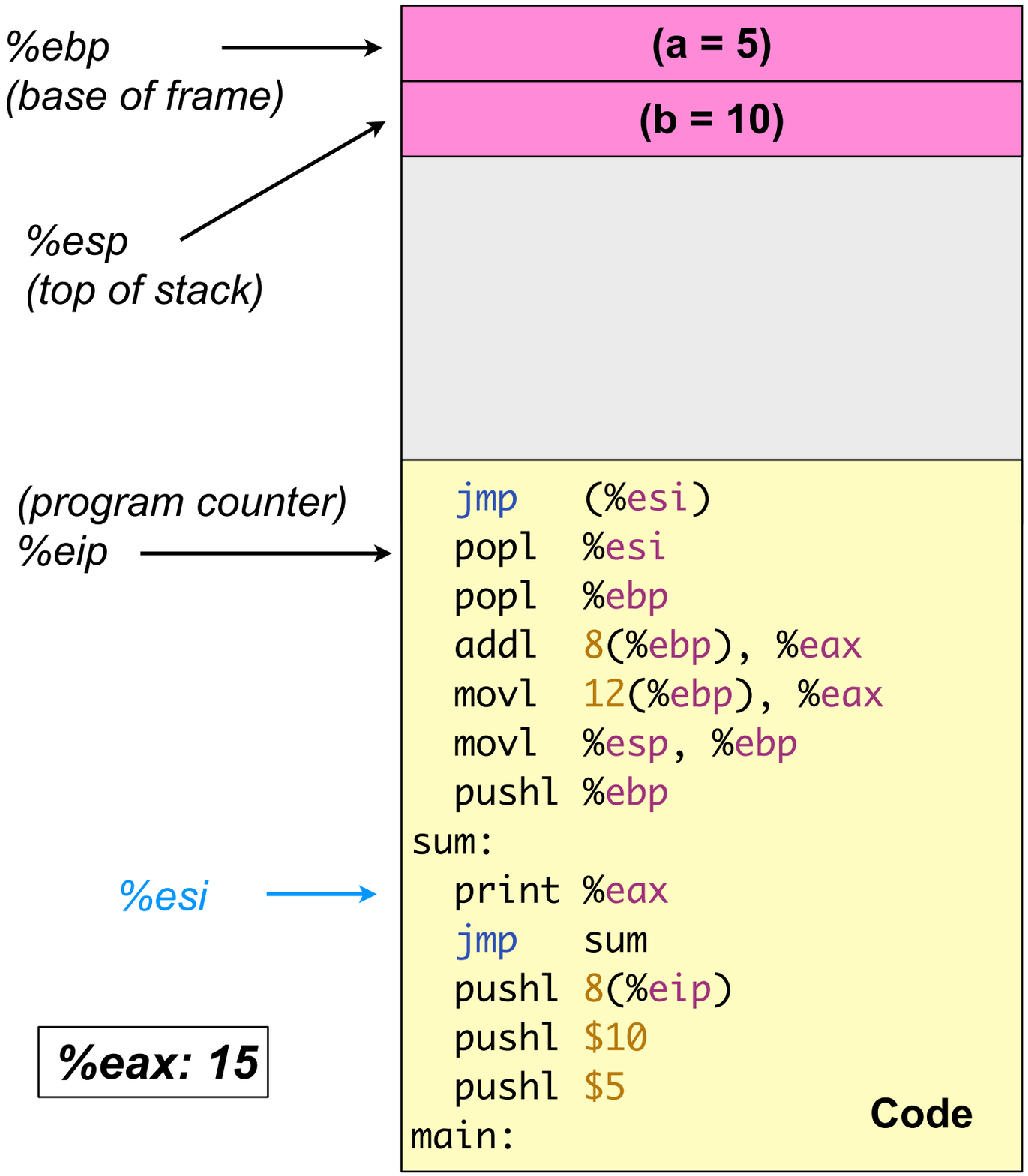
# x86 Example (11)



# x86 Example (12)



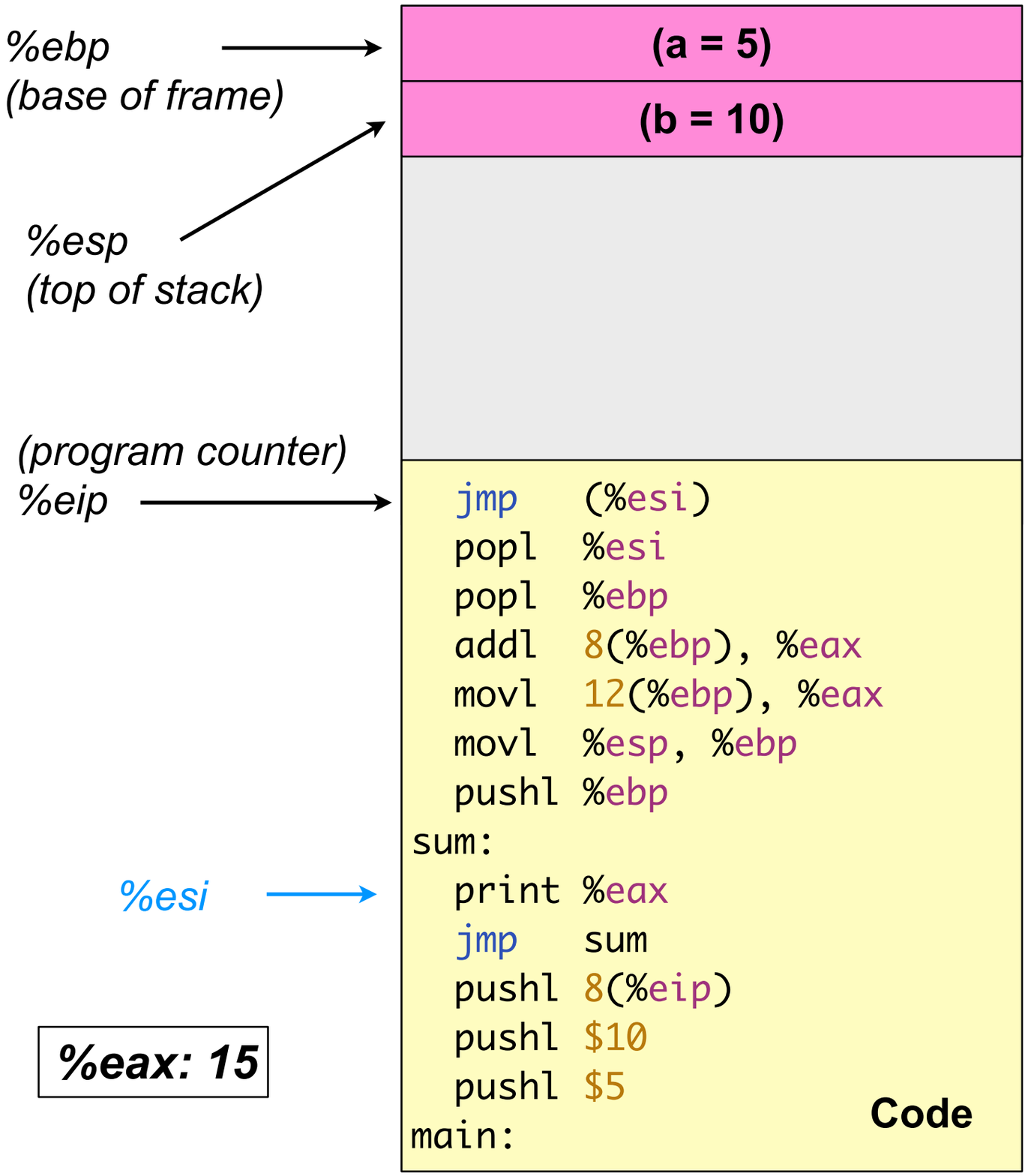
# x86 Example (13)




main()'s frame

Store return address (20) in %esi

# x86 Example (14)

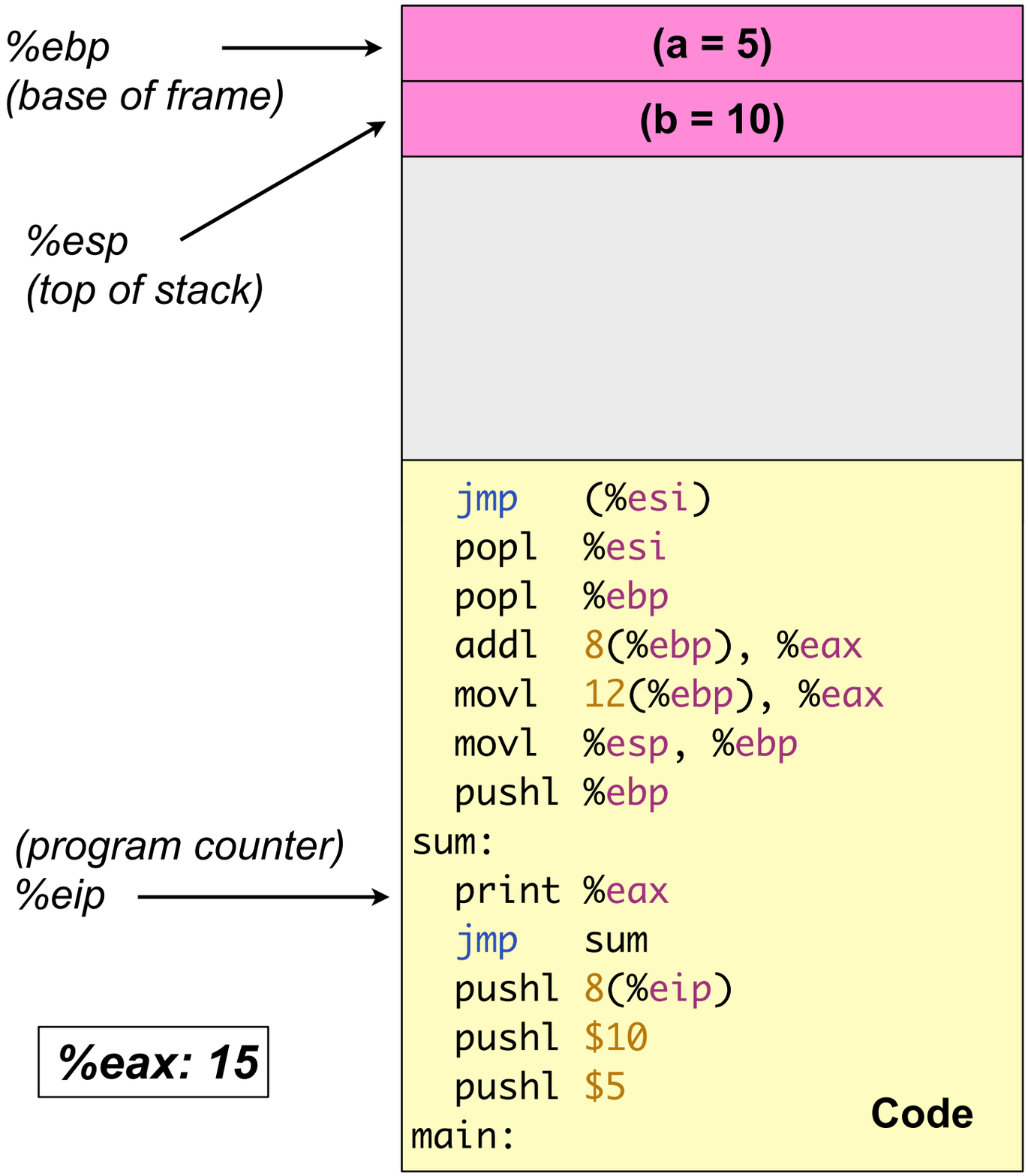



**main()'s frame**

Jump to return address

***%eax: 15***

# x86 Example (15)




**main()'s frame**

Continue from where we left off in main()

**Print string in by** %eax

# Aside: Buffer Overflow Exploit

- One of the most common security vulnerabilities (still!)

```c
void foo() {
    char s[8];    // buffer to store input
    getInput(s);  // put stuff in my buffer
}

void getInput(char s[]) {
    char c;
    int i = 0;
    // read keyboard input until EOF is reached
    while ((c = getchar()) != EOF) {
        s[i] = c;
        i++;
    }
    s[i] = '\0';  //terminate string
}
```

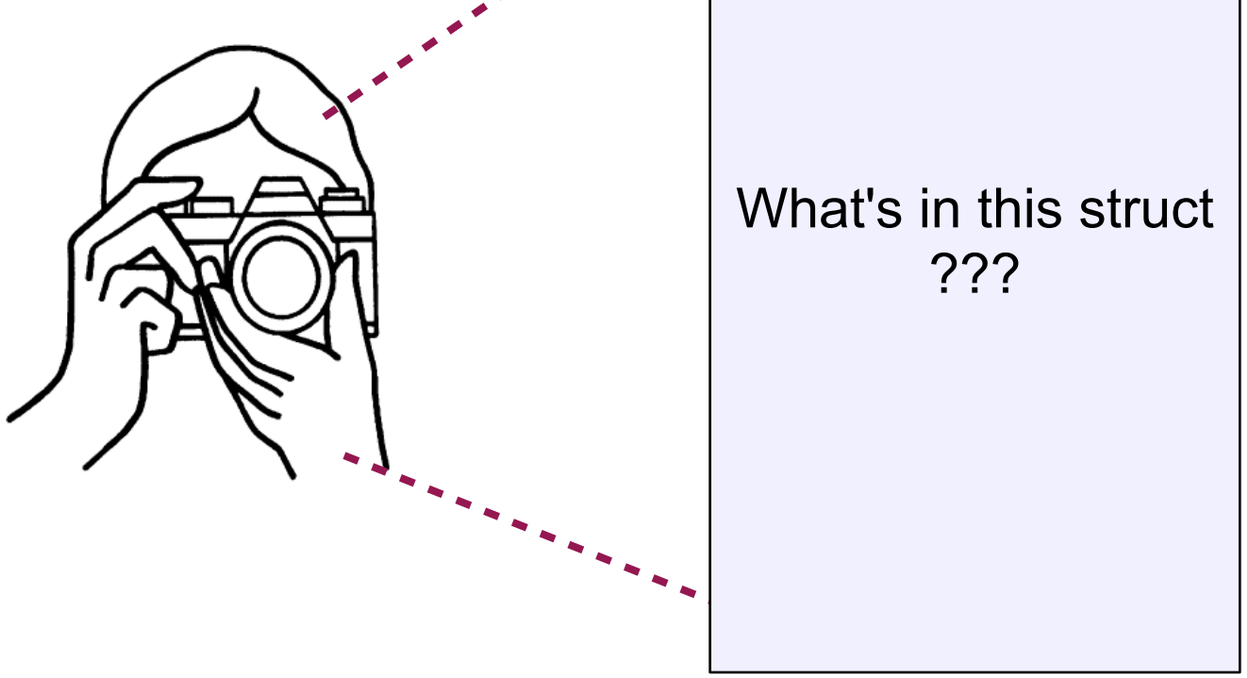
- How to avoid this? Burden is on *the programmer*...
  - Program defensively
  - How would *you* fix `getInput()`?

# Goals for This Lecture

- Motivation: Concurrency
- Address Space
- Processes and the PCB
- Process Management in C
  - Creation: fork()
  - Basic Synchronization: wait()
  - Running another program: exec()

# Process Control Block (PCB)

- We need to capture the state of processes at arbitrary points in time
  - *How is process state stored?*
- OS represents each process as a *Process Control Block (PCB)*
  - A "snapshot" of process execution
  - What info would a PCB need to store?



# Context Switch: Abstractly...

```
void contexSwitch(PCB *currentPCB, PCB *nextPCB) {

    /* save state of current PCB */
    currentPCB->r0 = CPU[r0];
    currentPCB->r1 = CPU[r1];
    currentPCB->r2 = CPU[r2];
    //...
    currentPCB->sp = CPU[sp];  // save stack pointer
    currentPCB->pc = CPU[pc];  // save program counter

    /* save current address space to disk */
    copy_to_disk(currentPCB->addr_sp);


    /* load new address space from disk */
    copy_from_disk(nextPCB->addr_sp);


    /* restore state of next PCB */
    CPU[r0] = nextPCB->r0;
    CPU[r1] = nextPCB->r1;
    CPU[r2] = nextPCB-.r2;
    //...
    CPU[sp] = nextPCB->sp;
    CPU[pc] = nextPCB->pc;

}
```

# Context Switch: Food for Thought

- Saving and Loading PCBs aren't too much work on their own...

- Context switches don't seem like much code but they *are* expensive!
  - First, switches triggered by either a timer interrupt or a trap (a process yields the CPU), so it's slow to handle.
  - Also, picking next process to run (called scheduling) is expensive too.

- A single context switch in Linux on a modern CPU:
  - Requires ~10,000 CPU cycles
    - In an ideal situation, 1 instruction could be executed per cycle!
    - That’s a lot of squandered CPU cycles!

# PCB in Linux (No you don't have to know this)

```
struct task_struct {
    long state;
    long counter;
    long priority;
    unsigned long signal;
    unsigned long blocked;
    unsigned long flags;
    int errno;
    long debugreg[8];
    struct exec_domain *exec_domain;

     /* various fields */
    struct linux_binfmt *binfmt;
    struct task_struct *next_task, *prev_task;
    struct task_struct *next_run,  *prev_run;
    unsigned long saved_kernel_stack;
    unsigned long kernel_stack_page;
    int exit_code, exit_signal;
    unsigned long personality;
    int dumpable:1;
    int did_exec:1;
    int pid;
    int pgrp;
    int tty_old_pgrp;
    int session;
    int leader;
    int groups[NGROUPS];
    struct task_struct *p_opptr, *p_pptr, *p_cptr, *p_ysptr, *p_osptr;
    struct wait_queue *wait_chldexit;
    unsigned short uid,euid,suid,fsuid;
    unsigned short gid,egid,sgid,fsgid;

    unsigned long timeout, policy, rt_priority;
    unsigned long it_real_value, it_prof_value, it_virt_value;
    unsigned long it_real_incr, it_prof_incr, it_virt_incr;
    struct timer_list real_timer;
    long utime, stime, cutime, cstime, start_time;

    /* mm fault and swap info: */
        unsigned long min_flt, maj_flt, nswap,
              cmin_flt, cmaj_flt, cnswap;
        int swappable:1;
        unsigned long swap_address;
        unsigned long old_maj_flt;
        unsigned long dec_flt;
        unsigned long swap_cnt;
    /* limits */
        struct rlimit rlim[RLIM_NLIMITS];
        unsigned short used_math;
        char comm[16];
    /* file system info */
        int link_count;
        struct tty_struct *tty;
    /* ipc stuff */
        struct sem_undo *semundo;
        struct sem_queue *semsleeping;

        struct desc_struct *ldt;
        struct thread_struct tss;
        struct fs_struct *fs;

    /* open file information */
        struct files_struct *files;

    /* memory management info */
        struct mm_struct *mm;

    /* signal handlers */
        struct signal_struct *sig;
};
```

Each process is in one of several states (next)

Each process has a unique ID

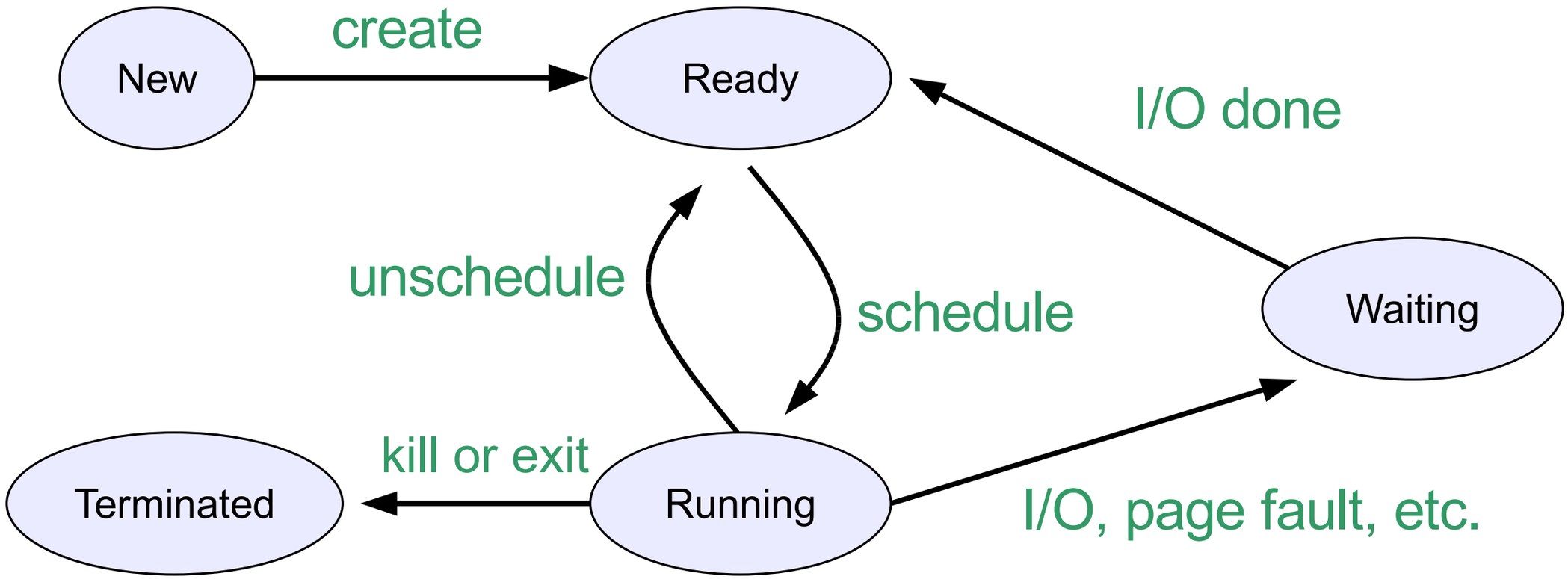
Each process may have a list of open files.

Start time; time in user mode time in kernel mode; etc.

Each process has a pointer to its address space. Also SP, BP, PC, all stored in *mm struct

# The Life Cycle of Processes: Process States

- As a process run, it transitions among these states
  - A process *cannot* be in two states at once!

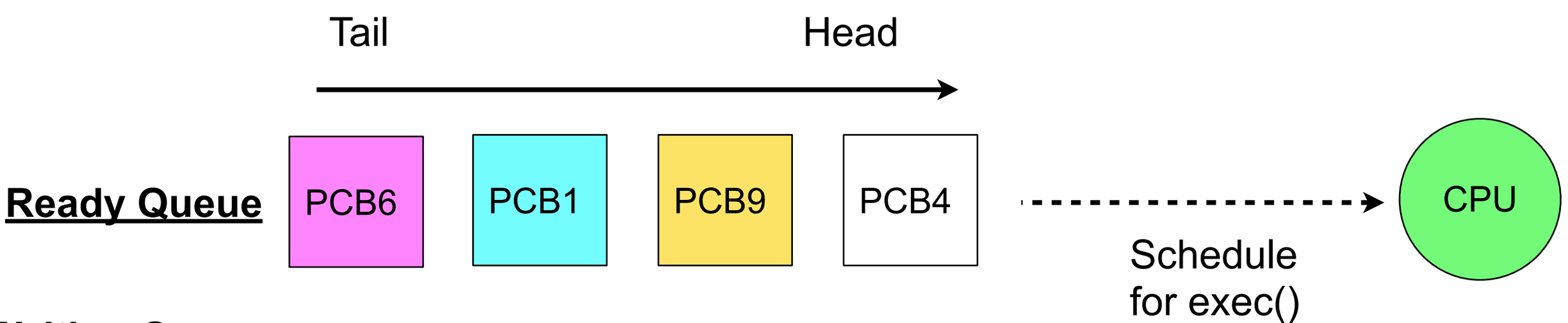


# Implementing State Transition with Queues

- An OS uses *Queues* to implement process states
  - Each queue contains pointers to PCBs
  - PCBs move from queue to queue as they change state

- Separate queues for *ready* and *waiting* states
  - The ready queue used for processes that are ready to receive CPU time
    - *Why separate ready and waiting states? They seem similar!*
  - Plus a wait queue for each I/O device
    - Disk Queue, Network Queue, Print Queue, *etc.*

# State Transitions: FIFO example



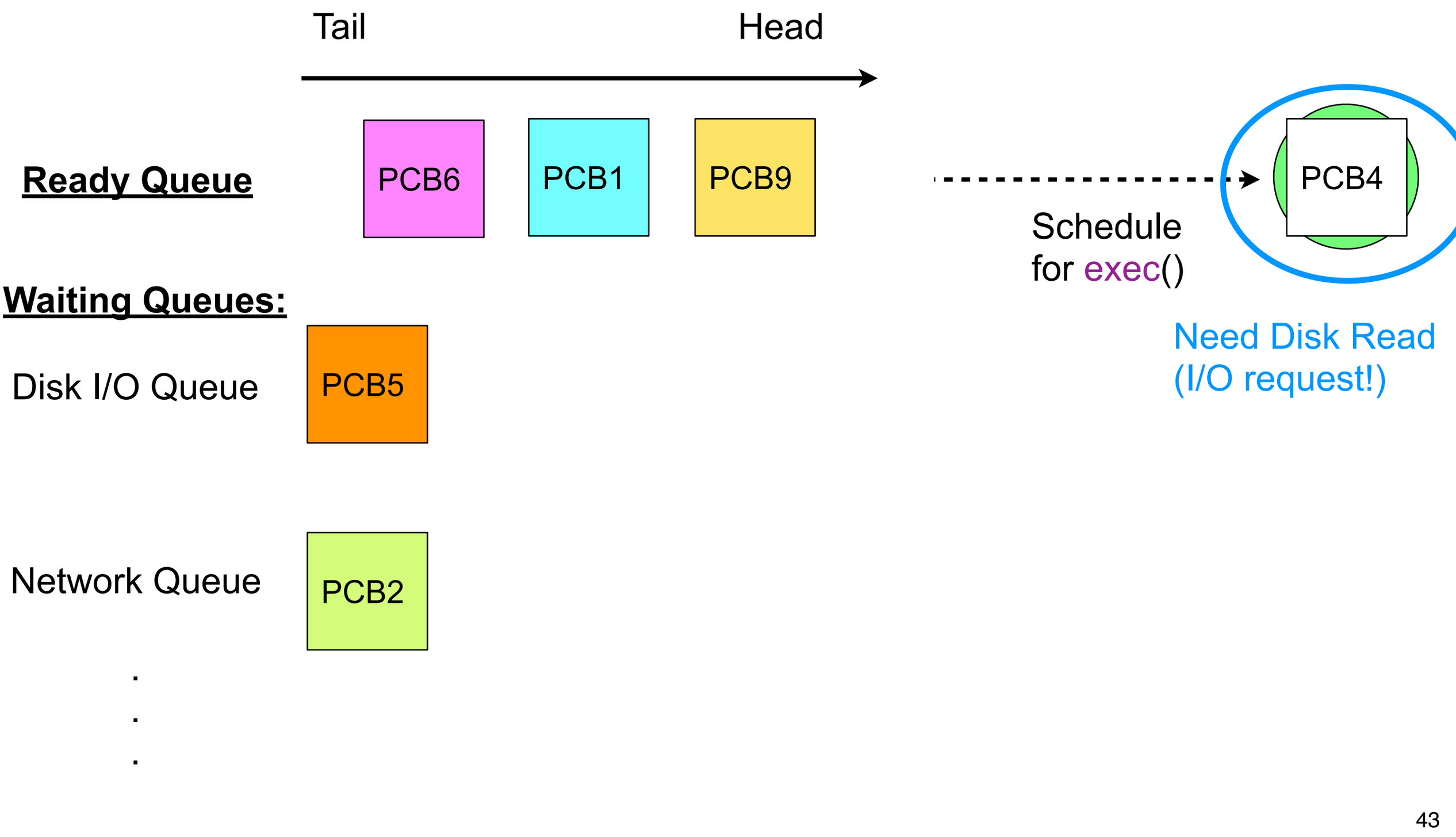

**Waiting Queues:**

Disk I/O Queue — PCB5

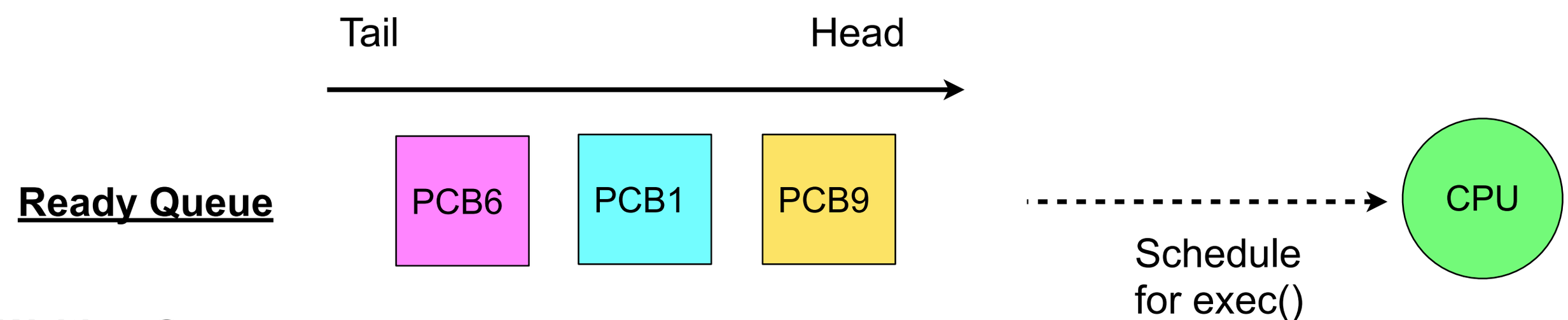
Network Queue — PCB2

.
.
.

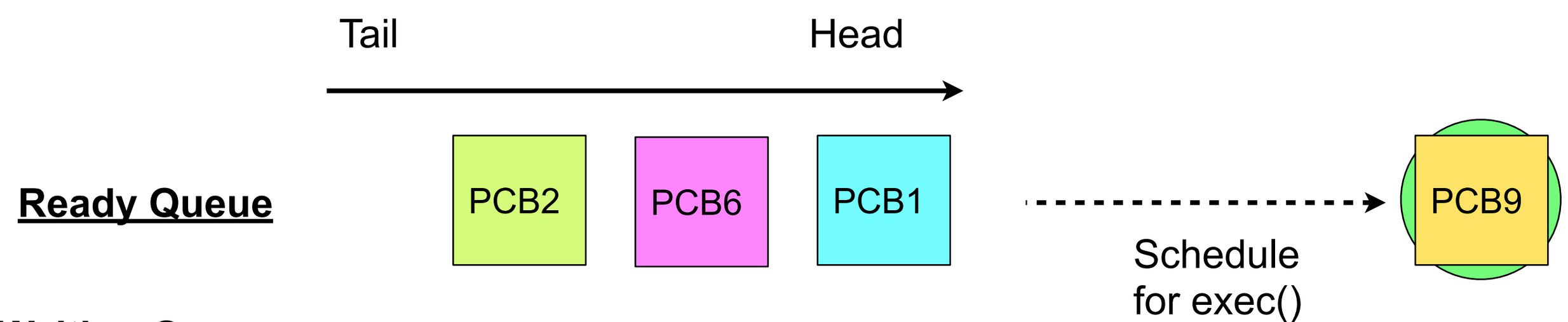
# State Transitions: FIFO example



# State Transitions: FIFO example



# State Transitions: FIFO example



# Example: When You Run 'ps au' in linux

***Process State (Just focus on the leading term)***

```
$ ps au

USER    PID  %CPU %MEM      VSZ    RSS   TT  STAT STARTED      TIME COMMAND
root  93801   0.0  0.0 34272300   1760 s001  R+    1:49PM   0:00.00 ps au
dchiu 93794   0.0  0.0 34406240   1908 s001  S     1:49PM   0:00.01 -bash
root  93793   0.0  0.0 34447976   5136 s001  Ss    1:49PM   0:00.03 login -fp d
dchiu 99996   0.0  0.0 33794880   1780 s000  Ss+  23Jan23   0:00.32 /bin/bash -
```

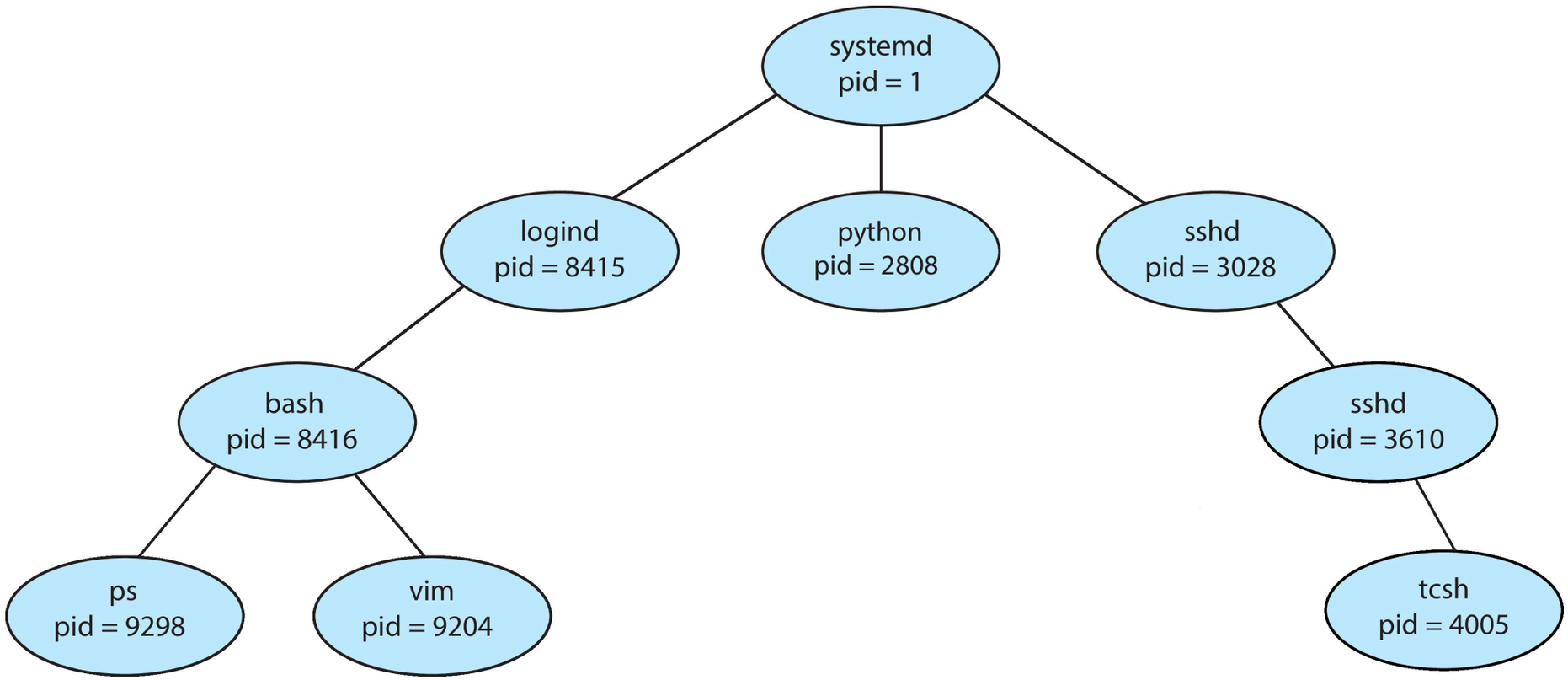
R = Running or Ready
S = Sleeping (Waiting)

# Goals for This Lecture

- Motivation: Concurrency
- Address Space
- Processes and the PCB
- Process Management in C
  - Creation: fork()
  - Basic Synchronization: wait()
  - Running another program: exec()

# Process Tree

- Except for the OS (the first process), all processes are created by another process
  - The lineage of each process is represented with a process tree



# Process Creation: `fork()`

- To create a new process on the OS, use the system call `fork()`
  - The calling process is called the ***parent** process*
  - The new process is called the ***child** process*

# Process Functions in C

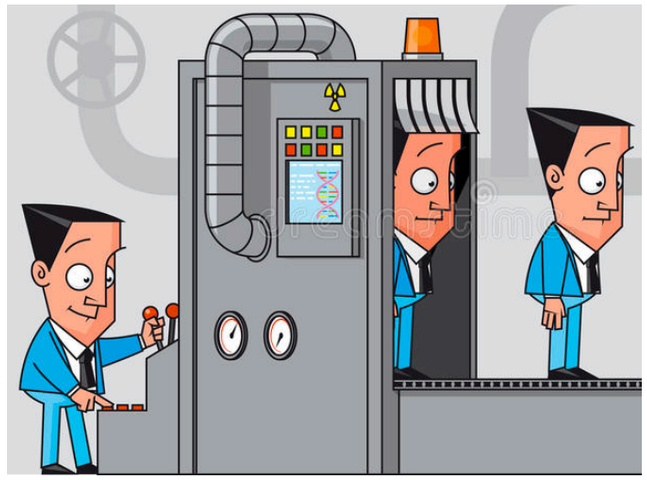
- Syntax in C:
  - Need to `#include <unistd.h>`
  - `pid_t` is just a typedef for `int`

```
/**
 * @return 0 if it's the child, the child's pid if parentâ€šÀè *
otherwise -1 on error
 */
pid_t fork();

/**
 * @return pid of the calling process
 */
pid_t getpid();

/**
 * @return pid of the calling process' parent
 */
pid_t getppid();
```

# Process Creation (Cont.)

- The behavior of `fork()` is a bit odd...

- `fork()` creates a *near-clone* of the parent process
  - Allocate a new PCB and Address Space for child
    - Copy parent's address space to child's
    - Copy parent's register contents (including PC, SP, BP...)
    - Copy parent's I/O state (*e.g.*, open file and network handles)
  - Child and parent then execute concurrently
  - Then `fork()` returns twice in your code!
    - *(What??)* Once for parent, once for child



# Process Creation (Cont.)

- Questions about `fork()`:
  - Does child really need a copy of parent's address space?
    - Seems wasteful and expensive to copy!
    - Just share until child makes a change to memory (*copy-on-write policy*)
  - "Nearly a clone?" What are some things that are private to the child?
    - What good is a clone? (We'll address that later)
- How do we get the child to run as an independent process?
- When might fork() return -1
  - (Indicating error during process creation)?

# Process Creation (Cont.)

- Child's execution starts where the parent's `fork()` returns
  - This is important, because it leads to confusing program traces!

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

int main(int argc, char *argv[]) {

    for (int i = 0; i < 10; i++) {
        printf("Process %d: value is %d\n", getpid(), i);

        //fork a child on the 6th iteration
        if (5 == i) {
            printf("Process %d: About to fork...\n", getpid());
            fork();
        }
    }
}
```

# Example Output: Non-Deterministic Execution!

```c
int main(int argc, char *argv[]) {
  int i;
  for (i = 0; i < 10; i++) {
    printf("Process %d: value is %d\n", getpid(), i);

    if (5 == i) {
        printf("Process %d: About to fork...\n", getpid());
        fork();
    }
  }
}
```

```
Process 4530: value is 0
Process 4530: value is 1
Process 4530: value is 2
Process 4530: value is 3
Process 4530: value is 4
Process 4530: value is 5
Process 4530: About to fork...
Process 4531: value is 6
Process 4530: value is 6
Process 4531: value is 7
Process 4530: value is 7
Process 4531: value is 8
Process 4530: value is 8
Process 4531: value is 9
Process 4530: value is 9
```

```
Process 4530: value is 0
Process 4530: value is 1
Process 4530: value is 2
Process 4530: value is 3
Process 4530: value is 4
Process 4530: value is 5
Process 4530: About to fork...
Process 4531: value is 6
Process 4531: value is 7
Process 4531: value is 8
Process 4531: value is 9
Process 4530: value is 6
Process 4530: value is 7
Process 4530: value is 8
Process 4530: value is 9
```

```
Process 4530: value is 0
Process 4530: value is 1
Process 4530: value is 2
Process 4530: value is 3
Process 4530: value is 4
Process 4530: value is 5
Process 4530: About to fork...
Process 4531: value is 6
Process 4530: value is 6
Process 4530: value is 7
Process 4530: value is 8
Process 4530: value is 9
Process 4531: value is 7
Process 4531: value is 8
Process 4531: value is 9
```

# Your Turn!

- How many Hellos get printed??

```c
int main() {
    fork();
    fork();
    printf("Hello\n");
    return 0;
}
```

```c
int main() {
    fork();
    fork();
    fork();
    printf("Hello\n");
    return 0;
}
```

# Telling Parents and Children Apart in Code

- The return value of `fork()` is the child's pid in the parent!
- The return value of `fork()` is 0 in the child!

```
int child_pid = fork();

if (0 != child_pid) {
   // parent process
   printf("I am the parent proc. My child's proc is %d\n", child_pid);
}
else {
   // child process
   printf("I am the child proc, parent proc is %d\n", getppid());
}
```

# Goals for This Lecture

- Motivation: Concurrency
- Address Space
- Processes and the PCB
- Process Management in C
  - Creation: fork()
  - Basic Process Synchronization: wait()
  - Running another program: exec()

# Synchronizing between Parent and Children

- Need to `#include <sys/wait.h>` and `#include <sys/types.h>`
- The `wait()` system call makes the parent wait for its child to terminate.

```
pid_t wait(int *status)
```

> `@param *status` Output parameter to the status of the terminating child (can be input as `NULL`)
>
> `@return` pid of the terminating child process

- Sometimes the parent wants to know *why* the child terminated: Normal exit? Was it terminated by OS?
  - More on exit status: man7.org/linux/man-pages/man2/wait.2.html

# Synchronizing between Parent and Children

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

int main(int argc, char *argv[]) {

    for (int i = 0; i < 10; i++) {

        printf("Process %d: value of i is %d\n",getpid(), i);
        if (5 == i) {
            printf("Process %d about to do a fork!\n",getpid());
            pid_t child_pid = fork();

            //the parent waits for child to finish
            if (0 != child_pid) {
                wait(NULL);
            }
        }
    }
}
```

# More C: A More General `wait()`

- If a parent created multiple children, can we specify which child to wait for?

```
pid_t waitpid(pid_t pid, int *status, int options);
```

`@param pid` a child's pid, or -1 for *any* child

`@param *status` output param to the status of the terminating child (can be `NULL`)

`@param options` Generally 0 (see man7.org/linux/man-pages/man2/wait.2.html)

`@return` pid of the child process

# Synchronizing between Parent and Children

```c
int main(int argc, char *argv[]) {

    for (int i = 0; i < 10; i++) {
        printf("Process %d: value of i is %d\n",getpid(), i);
        if (5 == i) {
            printf("Process %d about to do a fork!\n",getpid());
            pid_t child_pid = fork();

            //the parent waits for child to finish
            if (0 != child_pid) {
                wait(NULL);
            }
        }
    }
}
```

Parent waits here while child runs!

Child Terminates

Parent resumes

```
Process 12312: value of i is 0
Process 12312: value of i is 1
Process 12312: value of i is 2
Process 12312: value of i is 3
Process 12312: value of i is 4
Process 12312: value of i is 5
Process 12312 about to do a fork!
Process 12313: value of i is 6
Process 12313: value of i is 7
Process 12313: value of i is 8
Process 12313: value of i is 9
Process 12312: value of i is 6
Process 12312: value of i is 7
Process 12312: value of i is 8
Process 12312: value of i is 9
```

# Linux: Zombie Processes (Z State)

- What if the child terminates before the parent?
  - Recall all processes supply a **status code** on exit.
  - If **status** has not been read, OS keeps the PCB around until its parent runs `wait()` to read it (called *reaping*)

```c
pid_t child_pid = fork(); //fork a child

if (0 != child_pid) {
  sleep(60);  // parent sleeps for a minute
}
else {
  // This is the child process. Exit immediately.
  exit(0);
}
```

**zombify.c**

# Example: Zombies

```
$ gcc -o zombify zombify.c
$ ./zombify &
[1] 17788

$ ps au
USER    PID  %CPU %MEM      VSZ    RSS   TT  STAT STARTED      TIME COMMAND
root  17790   0.0  0.0  2444452   1136 s000  R+   11:21AM   0:00.01 ps au
dchiu 17788   0.0  0.0  2434856    784 s000  S    11:21AM   0:00.00 ./zombify
dchiu 10939   0.0  0.0  2467220    952 s000  Ss   Wed01PM   0:00.11 -/bin/bash
dchiu 17789   0.0  0.0        0      0 s000  Z    11:21AM   0:00.00 (zombify)
```

*60 seconds later....*

```
$ ps au
USER    PID  %CPU %MEM      VSZ    RSS   TT  STAT STARTED      TIME COMMAND
root  17809   0.0  0.0  2445476   1144 s000  R+   11:23AM   0:00.00 ps au
dchiu 10939   0.0  0.0  2467220    952 s000  Ss   Wed01PM   0:00.11 -/bin/bash
```

*But in our code, parent never calls **wait()** before it exits, how did the Zombie get reaped?*

*(First, we have to understand orphaning and adoption)*

# Linux: Orphaned Processes

- What if the parent terminates *before* its child terminates?

orphan.c

```c
int main(int argc, char *argv[])  {
    pid_t child_pid = fork(); //fork a child
    if (0 != child_pid) {
      sleep(1); //parent sleeps only a second before terminating
      exit(0);
    }
    else {
      //child lives on after the parent terminates
      for (int i = 0; i < 5; i++) {
          printf("mypid=%d, parentpid=%d\n", getpid(), getppid());
          sleep(3);
      }
    }
    return 0;
}
```

```
$ ./orphan
mypid=10383, parentpid=10382
$
mypid=10383, parentpid=1
mypid=10383, parentpid=1
mypid=10383, parentpid=1
```

Parent exits here!
(Command Prompt returns)

Child gets a new parent (`pid=1`)

New parent calls `wait()` on any adopted child, so no zombies!

# Recall: Process Tree

```
$ pstree -p
systemd(1)─┬─NetworkManager(2267)─┬─dhclient(2417)
           |                      └─{NetworkManager}(2418)
           ├─abrt-dump-oops(2726)
           ├─abrtd(2718)
           ├─acpid(2404)
           ├─atd(2802)
           ├─automount(2581)─┬─{automount}(2582)
           |                 ├─{automount}(2583)
           |                 ├─{automount}(2586)
           |                 └─{automount}(2589)
           ├─avahi-daemon(2279)───avahi-daemon(2280)
           ├─bonobo-activati(3169)───{bonobo-activat}(3170)
           ├─console-kit-dae(3068)─┬─{console-kit-da}(3069)
           |                       ├─{console-kit-da}(3070)
```

In Linux, all processes are descendants of the `systemd` process, which is the last step in the boot.

systemd then reads the system's initialization scripts (in /etc/init.d) and executes more programs, eventually completing the boot

# Example From Earlier: Who Reaped the Zombie? The OS Did!

```c
int main () {
  pid_t child_pid = fork(); //fork a child
  if (0 != child_pid) {
    sleep(60); //parent sleeps for a minute
  }
  else {
    // This is the child process. Exit immediately.
    exit(0);
  }
  return 0;
}
```

```
$ ps au
USER    PID  %CPU %MEM      VSZ    RSS   TT  STAT STARTED      TIME COMMAND
root  17790   0.0  0.0  2444452   1136 s000  R+   11:21AM   0:00.01 ps au
dchiu 17788   0.0  0.0  2434856    784 s000  S    11:21AM   0:00.00 ./zombify
dchiu 17789   0.0  0.0        0      0 s000  Z    11:21AM   0:00.00 (zombify)
```

*60 seconds later....*

```
$ ps au
USER    PID  %CPU %MEM      VSZ    RSS   TT  STAT STARTED      TIME COMMAND
root  17809   0.0  0.0  2445476   1144 s000  R+   11:23AM   0:00.00 ps au
dchiu 10939   0.0  0.0  2467220    952 s000  Ss   Wed01PM   0:00.11 -/bin/bash
```

***Rationale:*** *The OS adopted the Zombie (**17789**) process and reaped it once its parent (**17788**) exited without reaping*

# Goals for This Lecture

- Motivation: Concurrency
- Address Space
- Processes and the PCB
- Process Management in C
  - Creation: fork()
  - Basic Synchronization: wait()
  - Running another program: exec()

# Why Clone Processes?

- When might cloning be good?
  - Web Servers like apache `fork()` a child process to deal with every incoming connection
  - Parallel/Distributed Computing - `fork()` to do different parts of the same type of work
    - (*e.g.*, 2 children sorts two halves of a list, then parent merges the halves)

- Most programs want to spin off *different* programs though ***(next)***

# The `execv()` System Call

- We want to start a process that runs *a different* program

- Use `exec` family of functions:
  - Need to `#include <unistd.h>`

```
int execv(const char *filename, char *const argv[]);
```

`@param *filename` Path to the executable

`@param *argv[]` Input arguments for the executable

`@return` -1 if something went wrong, or doesn't return on normal execution

# `execv()` System Call (Cont.)

- ***** Important! Important! ******

- `execv()` *replaces* the *calling process'* address space and CPU state:
  - OS clears out the calling process' address space.
  - OS loads the given executable file into the code segment.
  - CPU State: Starts over from the beginning of the given executable.

# Example: Running `ls -l`

```c
#include <unistd.h>
#include <string.h>

int main(int argc, char *argv[]) {
        /** setting up to run exec() **/
        char *command = "/bin/ls";
        char **args = (char**) malloc(10 * sizeof(char*));
        for (int i = 0; i < 10; i++) {
                args[i] = (char*) malloc(64 * sizeof(char));
        }

        strcpy(args[0], command);  //store the full path to the command
        strcpy(args[1], "-l");
        args[2] = NULL;         //denote end of args list
        execv(command, args); // exec new program


        /** exec done running **/
        printf("Here's more stuff I wanted to do!!!!");
        printf("Here's more stuff I wanted to do!!!!");
        printf("Here's more stuff I wanted to do!!!!");

        return 0;
}
```

**testExec.c**

# Example: Running ls -l (Output)

`execv("/bin/ls")` called immediately

```
$ ./execTest
total 21556
-rwxr-xr-x. 1 dchiu domain users     8640 Jun 30  2021 blockinglock
-rw-r--r--. 1 dchiu domain users     1157 Jun 30  2021 blockinglock.c
-rwxr-xr-x. 1 dchiu domain users     1261 Jun 30  2021 BoundedBuffer.java
-rw-r--r--. 1 dchiu domain users      352 Jun 30  2021 collatz.c
-rwxr-xr-x. 1 dchiu domain users     1315 Jun 30  2021 Consumer.java
-rw-r--r--. 1 dchiu domain users      926 Feb 13 14:19 correct_execv.c
-rw-r--r--. 1 dchiu domain users     1428 Jun 30  2021 deadlock.c
-rw-r--r--. 1 dchiu domain users      804 Feb 13 14:26 execv.c
-rw-r--r--. 1 dchiu domain users      441 Jun 30  2021 fork2.c
-rw-r--r--. 1 dchiu domain users       82 Jun 30  2021 forkbomb.c
-rw-r--r--. 1 dchiu domain users      539 Jun 30  2021 fork.c
-rw-r--r--. 1 dchiu domain users      290 Jun 30  2021 fork_wait.c
-rwxr-xr-x. 1 dchiu domain users      698 Jun 30  2021 highlight.css
-rwxr-xr-x. 1 dchiu domain users      220 Jun 30  2021 index.php
drwxr-xr-x. 2 dchiu domain users      113 Jun 30  2021 lec4
-rw-r--r--. 1 dchiu domain users      340 Jun 30  2021 movingAvg.c
-rw-r--r--. 1 dchiu domain users      475 Feb 10 14:22 orphan.c
-rwxr-xr-x. 1 dchiu domain users    18232 Feb  1 15:47 perf
-rw-r--r--. 1 dchiu domain users      742 Feb  1 15:47 perf_syscall.c
-rwxr-xr-x. 1 dchiu domain users     1309 Jun 30  2021 Producer.java
-rw-r--r--. 1 dchiu domain users      313 Jun 30  2021 rtclock.c
-rwxr-xr-x. 1 dchiu domain users      200 Jun 30  2021 rtclock.h
-rw-r--r--. 1 dchiu domain users      201 Jun 30  2021 segfault.c
-rwxr-xr-x. 1 dchiu domain users     8632 Jun 30  2021 spinlock
-rw-r--r--. 1 dchiu domain users     1176 Jun 30  2021 spinlock.c
-rw-r--r--. 1 dchiu domain users      109 Jun 30  2021 test2.c
-rw-r--r--. 1 dchiu domain users      751 Jun 30  2021 test.c
-rwxr-xr-x. 1 dchiu domain users 21889024 Jun 30  2021 test.out
-rw-r--r--. 1 dchiu domain users      253 Feb  8 14:25 zombify.c

$
```

**printfs never run!**
**/bin/ls replaced execTest code**

# execv() System Call (Cont.)

- **Important:** `execv("newProg", ...)` does *not* `fork()` a new process
  - 0) It *replaces* the calling process with `newProg`!
  - 1) You have to first explicitly call `fork()`
  - 2) Followed by `execv("newProg", ...)` in the **child** to start `newProg` as a separate process!

# Example: Parent Side

University of Puget Sound

**realTestExec.c**

```
int main(int argc, char *argv[]) {
    if (0 != fork()) {    //parent
        printf("pid=%d, Parent here... print to 999\n", getpid());
        int i;
        for (i=0; i<1000; i++)
            printf("pid=%d, %d\n",getpid(),i);
    }
```

**Focus on Parent code first**

```
    printf("pid=%d, Child proc should never get here\n",getpid());
}
```

# Example: Child Side

**realTestExec.c**

```
else {    //child
    printf("pid=%d, I'm the child. I'm going to run echo\n", getpid());
    char *command = "/bin/echo";
    char *newargs[3];
    newargs[0] = command;                  // full path to executable file!
    newargs[1] = "Random String";          // argument to /bin/echo
    newargs[2] = NULL;                     // denote end of args list
    execv(command, newargs); //spawn new program
}
```

# Putting It All Together

University of Puget Sound

**realTestExec.c**

```
int main(int argc, char *argv[]) {
    if (0 != fork()) {    //parent
        printf("pid=%d, Parent here... print to 999\n", getpid());
        int i;
        for (i=0; i<1000; i++)
            printf("pid=%d, %d\n",getpid(),i);
    }
    else {    //child
        printf("pid=%d, I'm the child. I'm going to run echo\n", getpid());
        char *command = "/bin/echo";
        char *newargs[3];
        newargs[0] = command;                  // full path to executable file!
        newargs[1] = "Random String";          // argument to /bin/echo
        newargs[2] = NULL;                     // denote end of args list
        execv(command, newargs); //spawn new program
    }
    printf("pid=%d, Child proc should never get here\n",getpid());
}
```

# Example Output of **realTestExec**

(1) Parent starts and forks child

(2) Parent keeps processing

.

.

(3) Child starts running too

(4) Child fires `execv()` here

(5) Child finishes

(6) Parent finishes

```
pid=12754, Parent here... print to 999
pid=12754, 0
pid=12754, 1
.
.
pid=12754, 24
pid=12755, I'm the child. I'm going to run echo
pid=12754, 25
.
.
pid=12754, 311
Random String
.
.
pid=12754, 998
pid=12754, 999
pid=12754, Child proc should never get here
```

# Goals for This Lecture

- Motivation: Concurrency
- Address Space
- Processes and the PCB
- Process Management in C
  - Creation: fork()
  - Basic Synchronization: wait()
  - Running another program: exec()
- Conclusion

# In Conclusion...

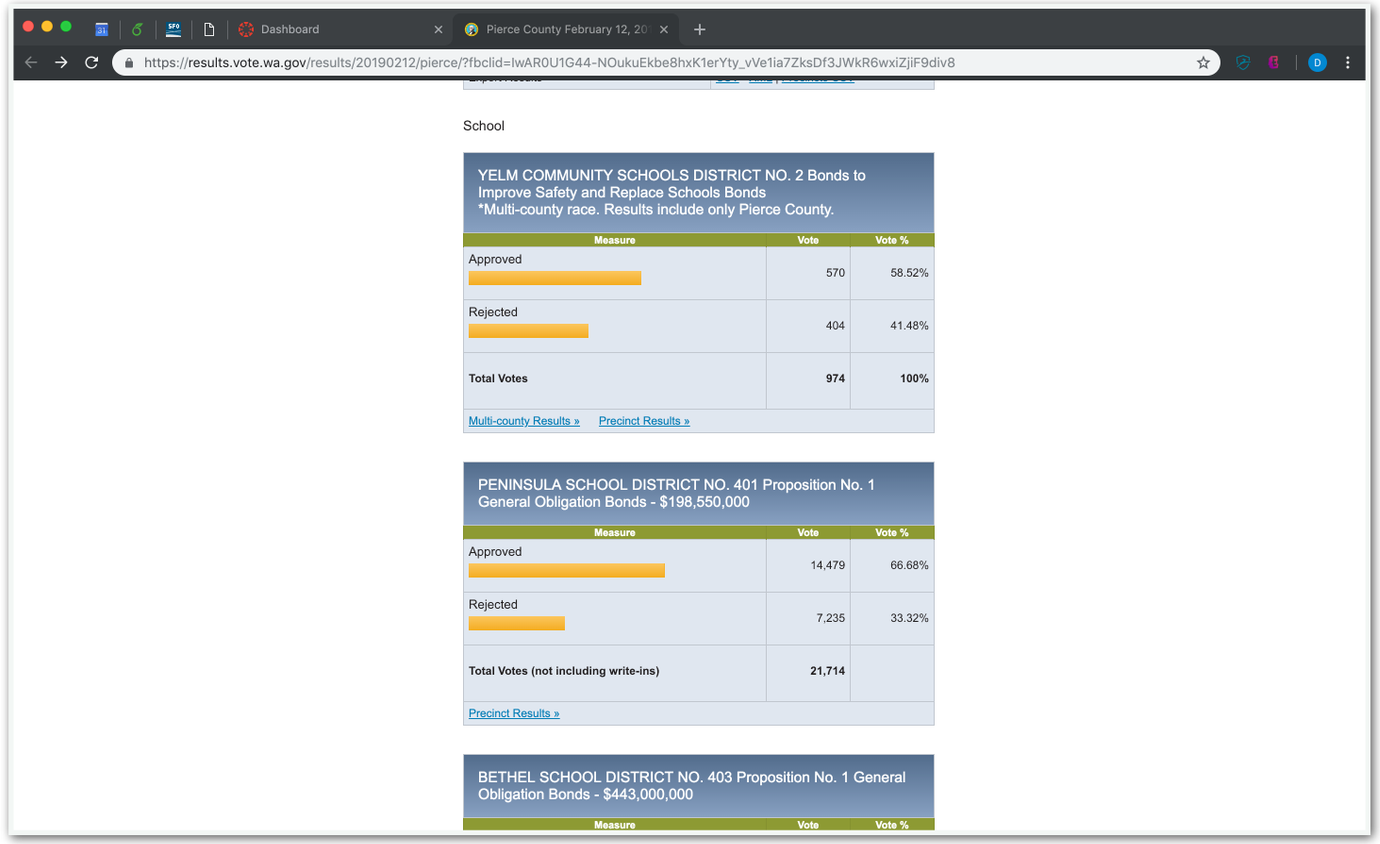
- To multiprogram and timeshare a computer system, we must allow multiple programs to co-exist and share the CPU
  - Process control blocks (PCB) stores a snapshot of each process state
  - PCBs can be various states of execution in its lifecycle
  - Pause and play among the PCBs to multi-task

# `fork()` and `exec()` in Java?

- In Java, closest thing to `fork()` is the `ProcessBuilder` class
- Can invoke an arbitrary program
  - Pass in arguments via the standard input
  - Read results via the standard output
- Pro - Very easy to use
- Con - Has a big startup overhead
- API: https://docs.oracle.com/javase/8/docs/api/java/lang/ProcessBuilder.html

# Next Time...

- Problem: Processes are inefficient: fork+exec is expensive
  - Open Chrome (one process), download HTML (fork), download images/ css from various locations (fork for each), ...




- Next Time: Processes are expensive. *(Threads)*

# Administrivia 2/6

- Announcements
  - Hwk 3 due tonight
  - Start reading Chapter 3.1-3.5 (dinosaur book)
- Last time:
  - Finished system calls and traps
- Today:
  - Process Management
  - Address Spaces
    - Program Stack

# Administrivia 2/8

- Reminders
  - Hwk 4 assigned, due Mon 2/20
- Last time...
  - Policy: CPU virtualization. Mechanism: Context Switch
  - PCBs and context switching
  - Address Spaces; How does the program stack manage scope?
- Today:
  - Process states
  - Process creation: fork() system call

# Administrivia 2/10

- Reminders
  - Hwk 4 assigned, due Mon 2/20
- Last time...
  - Process states
  - Transitions between states
  - Fork system call
- Today:
  - The wait() system call and reaping children processes
  - New terms: Zombies and Orphans

# Administrivia 2/13

- Reminders
  - Hwk 4 due Mon 2/20
- Last time...
  - The wait() system call
  - Zombie processes and reaping statuses
  - Orphaned processes and adoption
- Today
  - Finish Processes (Chap 3):
    - The even weirder exec() system call
  - Start threads (Chap 4 dinosaur book)