CSC 374/407: Computer Systems II

Lecture 3
Joseph Phillips
De Paul University

2016 September 6

Reading

- Bryant & O'Hallaron "*Computer Systems, 3rd Ed.*"
  - Chapter 8: Exception Control Flow (8.1 – 8.4)
- Hoover "*System Programming*"
  - System Calls 7.1-7.4

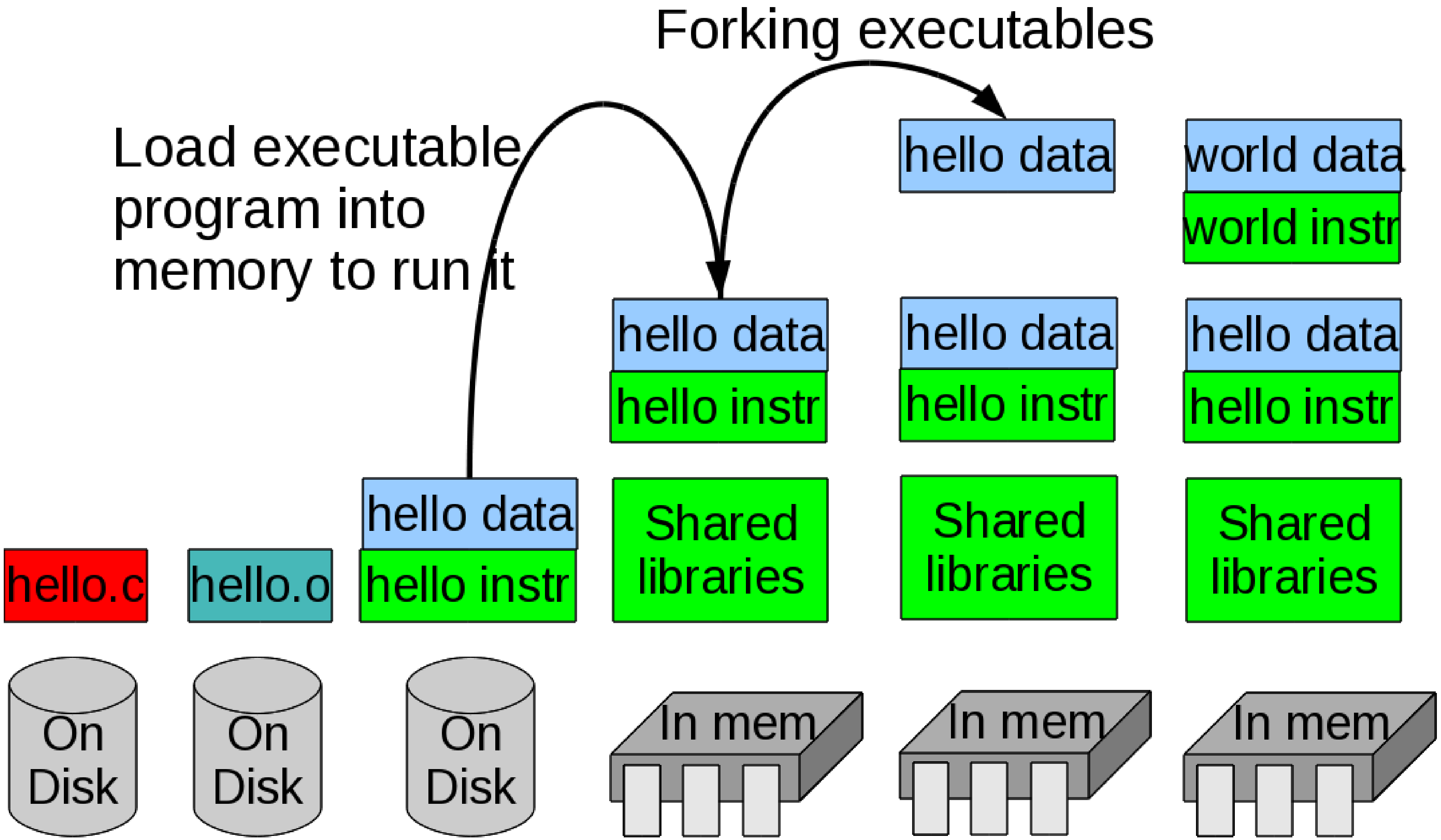
Topics

- Processes 1
  - Exceptions
    - Interrupts
    - Trap
    - Fault
    - Abort
  - Process lifecycle
  - System calls

- Processes 2
  - **`fork()`**
  - **`getpid()`** and **`getppid()`**
  - **`exit()`** and **`atexit()`**
  - **`execl()`**
  - **`int argc`** and **`char* argv[]`**
  - **`wait()`**, **`waitpid()`**
  - Zombies

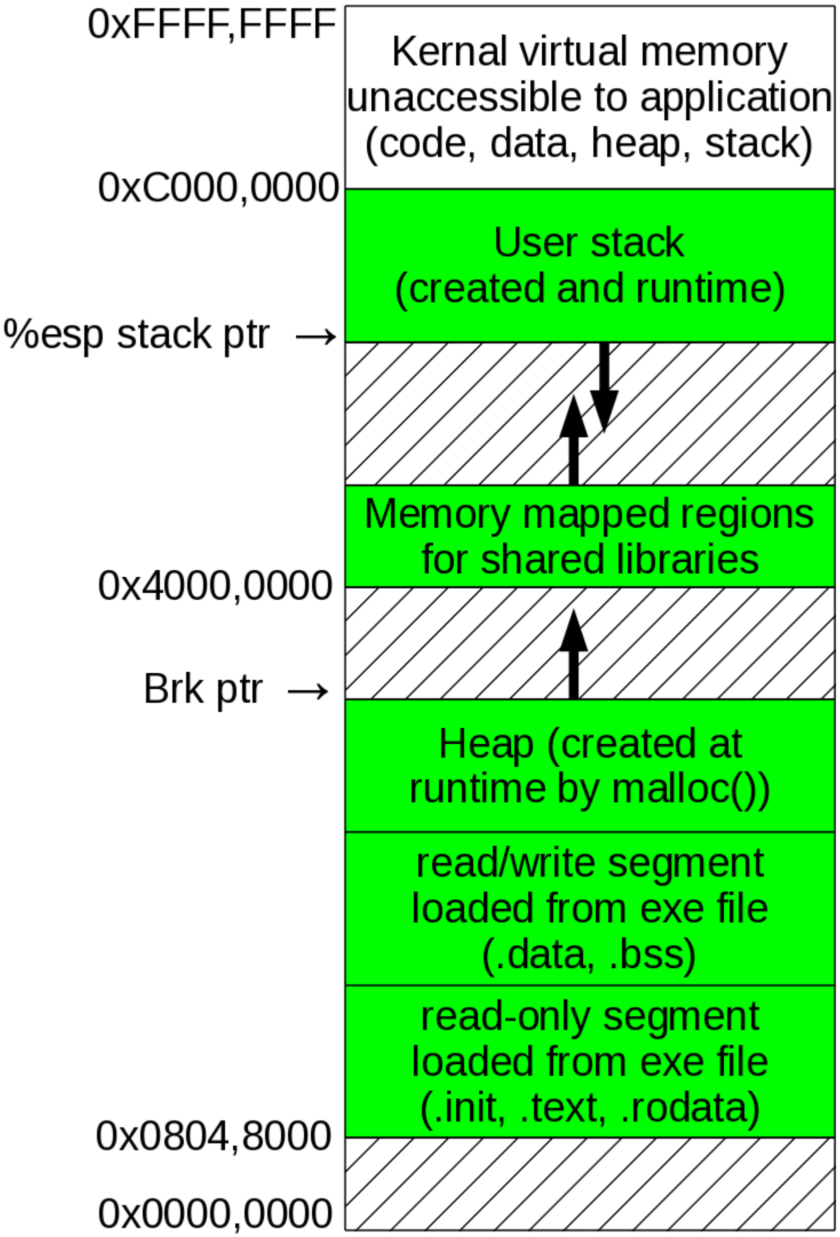
Today's topic (in time)

Running executable files and forking



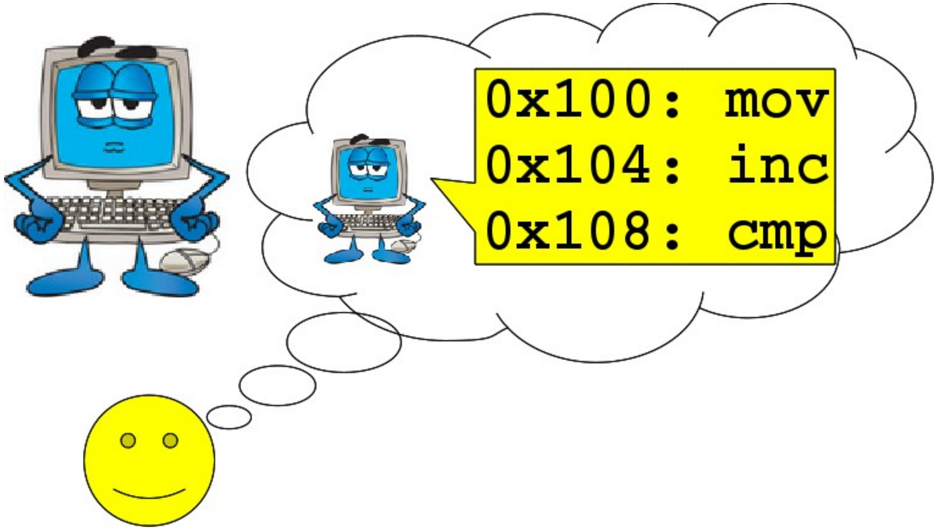
Today's topic (in space)

Loading or setting up everything:


0xFFFF,FFFF
Kernal virtual memory unaccessible to application (code, data, heap, stack)
0xC000,0000
User stack (created and runtime)
%esp stack ptr →
Memory mapped regions for shared libraries
0x4000,0000
Brk ptr →
Heap (created at runtime by malloc())
read/write segment loaded from exe file (.data, .bss)
read-only segment loaded from exe file (.init, .text, .rodata)
0x0804,8000
0x0000,0000

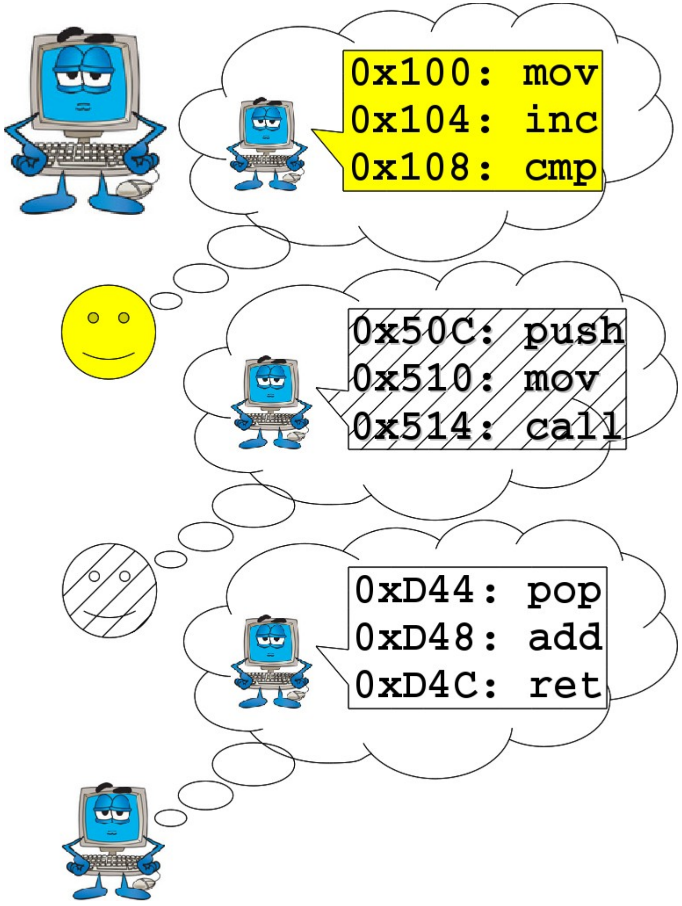
One user's view of a computer

- Computer spends all of its time doing instructions for *my* program



Of course computers run multiple processes

- More than one user
- More than one process per user
- OS also handles hardware events (key press, network traffic, timer) and its own maintenance (disk defragmenting)

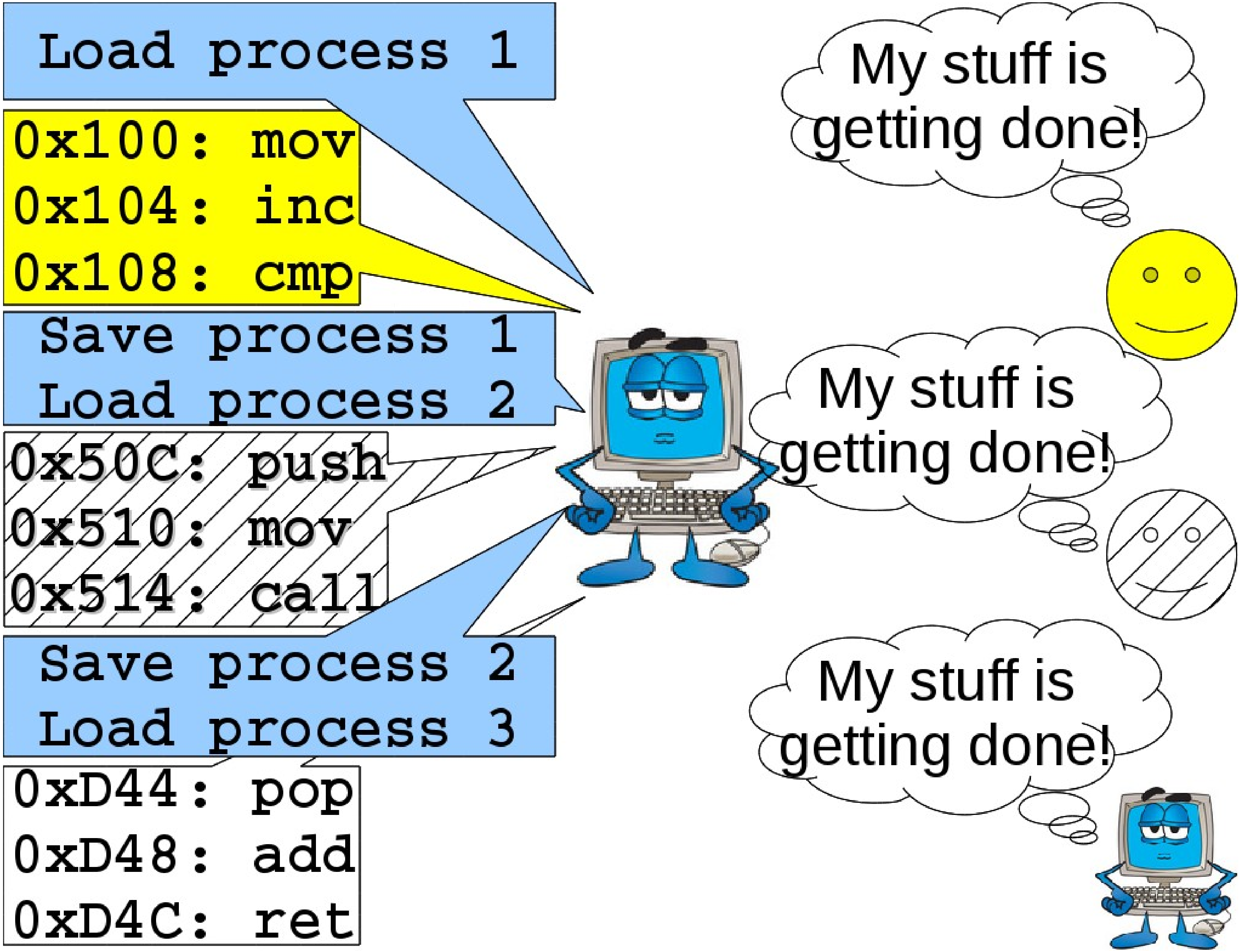


```
0x100: mov
0x104: inc
0x108: cmp
```

```
0x50C: push
0x510: mov
0x514: call
```

```
0xD44: pop
0xD48: add
0xD4C: ret
```

How computers do it



Load process 1

```
0x100: mov
0x104: inc
0x108: cmp
```

Save process 1
Load process 2

```
0x50C: push
0x510: mov
0x514: call
```

Save process 2
Load process 3

```
0xD44: pop
0xD48: add
0xD4C: ret
```

My stuff is getting done!
My stuff is getting done!
My stuff is getting done!

What's in a context switch?

- What's has to be loaded and saved?
  - Registers
  - Page table
  - Signal vector (more about that next week)
  - CPU and memory usage stats
  - List of open files
  - *etc.*
- Doing a **`jmp`** to your own address or a **`call`** to your own function handled by a process without outside help.
- ***Exceptions***, however needed to:
  - Switch between processes
  - Access system resources

4 types of exceptions

Class	Cause	Synchronicity	Return behavior
Interrupt	Signal from hardware	Async	Next instruct
Trap	Intentional call	Sync	Next instruct
Fault	Potentially recoverable error	Sync	*Maybe* next instruct
Abort	Unrecoverable error	Sync	Never returns

Interrupts

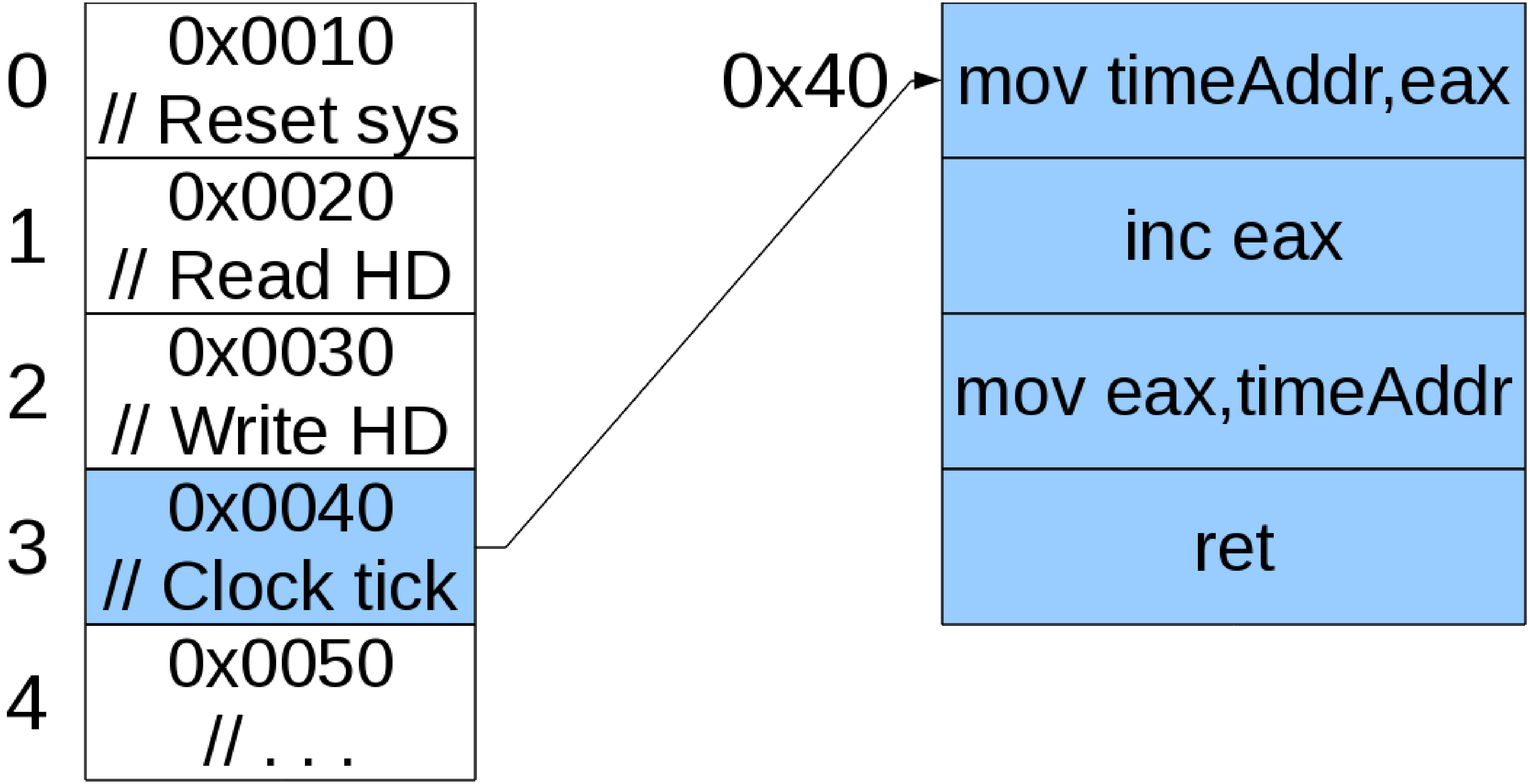
- Hardware says “I don't care which ordinary user's process you're running, ***attend to me now! (or at least soon)***”
  - A key is pressed by the user
  - A packet comes over the network
  - Computer's timer goes off

Doing an interrupt

1. An interrupt (identified by an integer) fires
2. Save state of running process
    - Common to push registers on stack
3. Go to interrupt table (*aka* vector table): array of pointers to code to handle interrupts
4. Get address from array
5. Go to that address
6. Do code
7. Re-load state of running process

Interrupt table or vector table

- Array of pointers to functions to handle specific interrupts



0	0x0010 // Reset sys
1	0x0020 // Read HD
2	0x0030 // Write HD
3	0x0040 // Clock tick
4	0x0050 // . . .

```
mov timeAddr,eax
inc eax
mov eax,timeAddr
ret
```

“IBM PC” i386 Interrupt Table (1)

- ROM BIOS by IBM (Microsoft?) circa 1981
- Now standardized and written by:
  - American Megatrends Inc
  - Micro Firmware
  - Phoenix Technologies

00h CPU: Executed after an attempt to divide by zero or when the quotient does not fit in the destination

01h CPU: Executed after every instruction while the trace flag is set

02h CPU: NMI, used e.g. by POST for memory errors

03h CPU: The lowest non-reserved interrupt, it is used exclusively for debugging, and the INT 03 handler is always implemented by a debugging program

“IBM PC” i386 Interrupt Table (2)

04h CPU: Numeric Overflow. Usually caused by the INTO instruction when the overflow flag is set.

05h Executed when Shift-Print screen is pressed, as well as when the BOUND instruction detects a bound failure.

06h CPU: Called when the Undefined Opcode (invalid instruction) exception occurs. Usually installed by the operating system.

07h CPU: Called when an attempt was made to execute a floating-point instruction and no numeric coprocessor was available.

08h IRQ0: Implemented by the system timing component; called 18.2 times per second (once every 55 ms) by the PIC.

09h IRQ1: Called after every key press and release (as well as during the time when a key is being held)

“IBM PC” i386 Interrupt Table (3)

0Ah Reserved for OS?

0Bh IRQ3: Called by serial ports 2 and 4 (COM2/4) when in need of attention

0Ch IRQ4: Called by serial ports 1 and 3 (COM1/3) when in need of attention

0Dh IRQ5: Called by hard disk controller (PC/XT) or 2nd parallel port LPT2 (AT) when in need of attention

0Eh IRQ6: Called by floppy disk controller when in need of attention

0Fh IRQ7: Called by 1st parallel port LPT1 (printer) when in need of attention

10h Video Services – installed by the BIOS or operating system; called by software programs

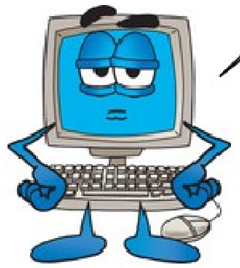
AH=00h: Set video mode;

AH=01h: Set Cursor shape

Etc.

Example interrupt (1)

0x100	mov
0x104	inc
0x108	cmp



mov
inc

Int table

0	0x0010
1	0x0020
2	0x0030
3	0x0040
4	0x0050

Stack

0xF18	0x010
0xF14	0x030
0xF10	???
0xF0C	???
0xF08	???

Registers

eip = 0x108
eax = 0x08
esp = 0xF10
int = ?

incTime()

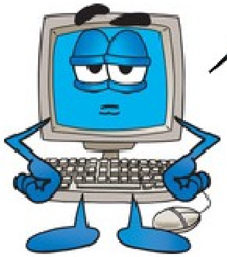
0x40	mov timeAddr,eax
0x44	inc eax
0x48	mov eax, timeAddr
0x4C	ret

timeAddr

12:00:00

Example interrupt (2)

0x100	mov
0x104	inc
0x108	cmp



Uhoh!
Int 3

Int table

0	0x0010
1	0x0020
2	0x0030
3	0x0040
4	0x0050

Stack

0xF18	0x010
0xF14	0x030
0xF10	???
0xF0C	???
0xF08	???

Registers

eip = 0x108
eax = 0x08
esp = 0xF10
int = *3*

timeAddr

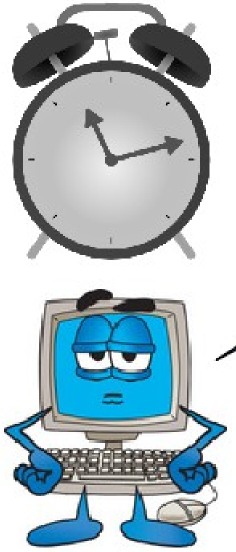
12:00:00

incTime()

0x40	mov timeAddr,eax
0x44	inc eax
0x48	mov eax, timeAddr
0x4C	ret

Example interrupt (3)

0x100	mov
0x104	inc
0x108	cmp



Push regsiters

Int table

0	0x0010
1	0x0020
2	0x0030
3	0x0040
4	0x0050

Stack

0xF18	0x010
0xF14	0x030
0xF10	0x108
0xF0C	0x08
0xF08	???

Registers

eip = 0x108
eax = 0x08
esp = 0xF08
int = *3*

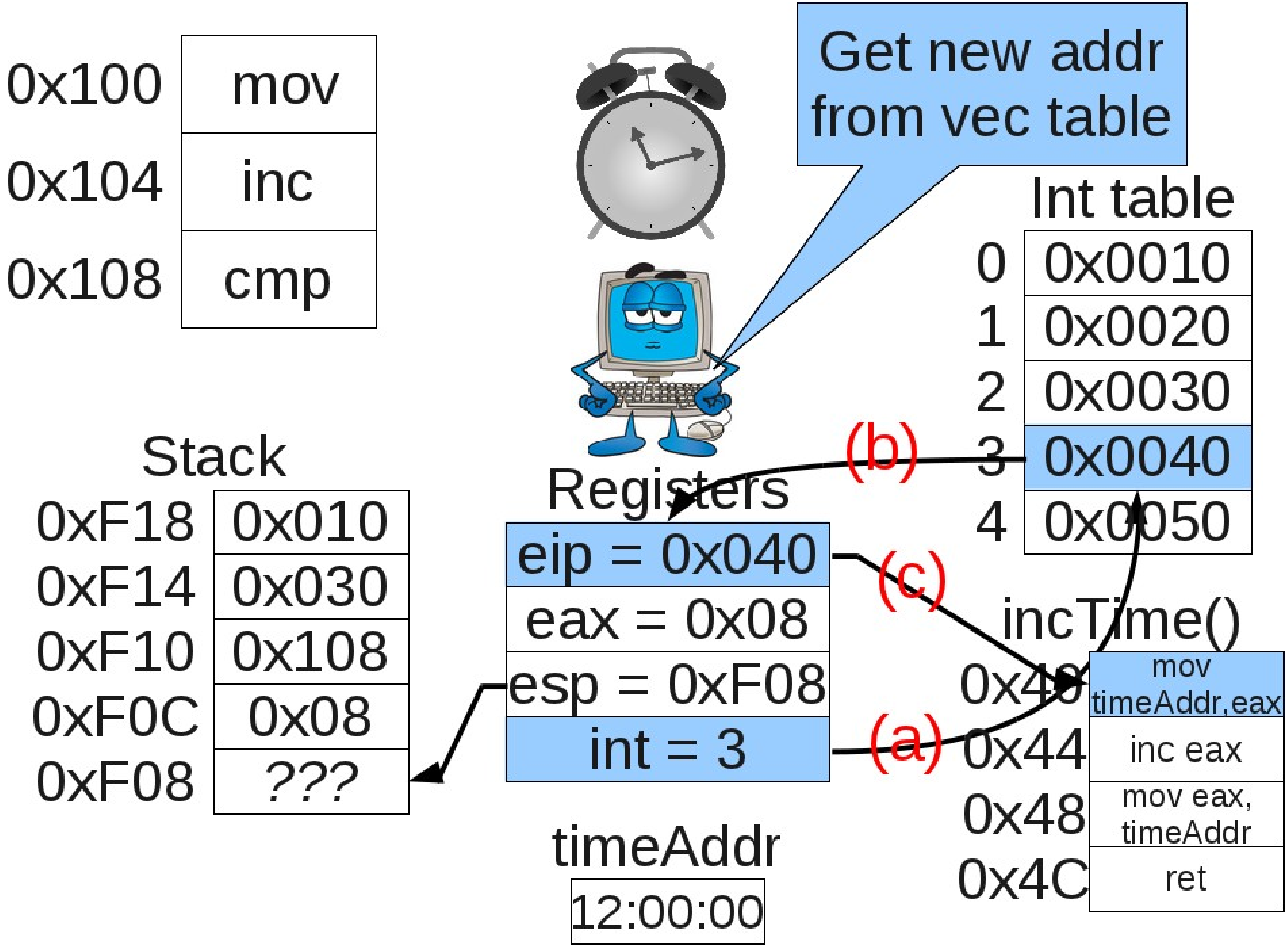
timeAddr

12:00:00

incTime()

0x40	mov timeAddr,eax
0x44	inc eax
0x48	mov eax, timeAddr
0x4C	ret

Example interrupt (4)



Get new addr from vec table

0x100	mov
0x104	inc
0x108	cmp

Int table

0	0x0010
1	0x0020
2	0x0030
3	0x0040
4	0x0050

Stack

0xF18	0x010
0xF14	0x030
0xF10	0x108
0xF0C	0x08
0xF08	???

Registers

eip = 0x040
eax = 0x08
esp = 0xF08
int = 3

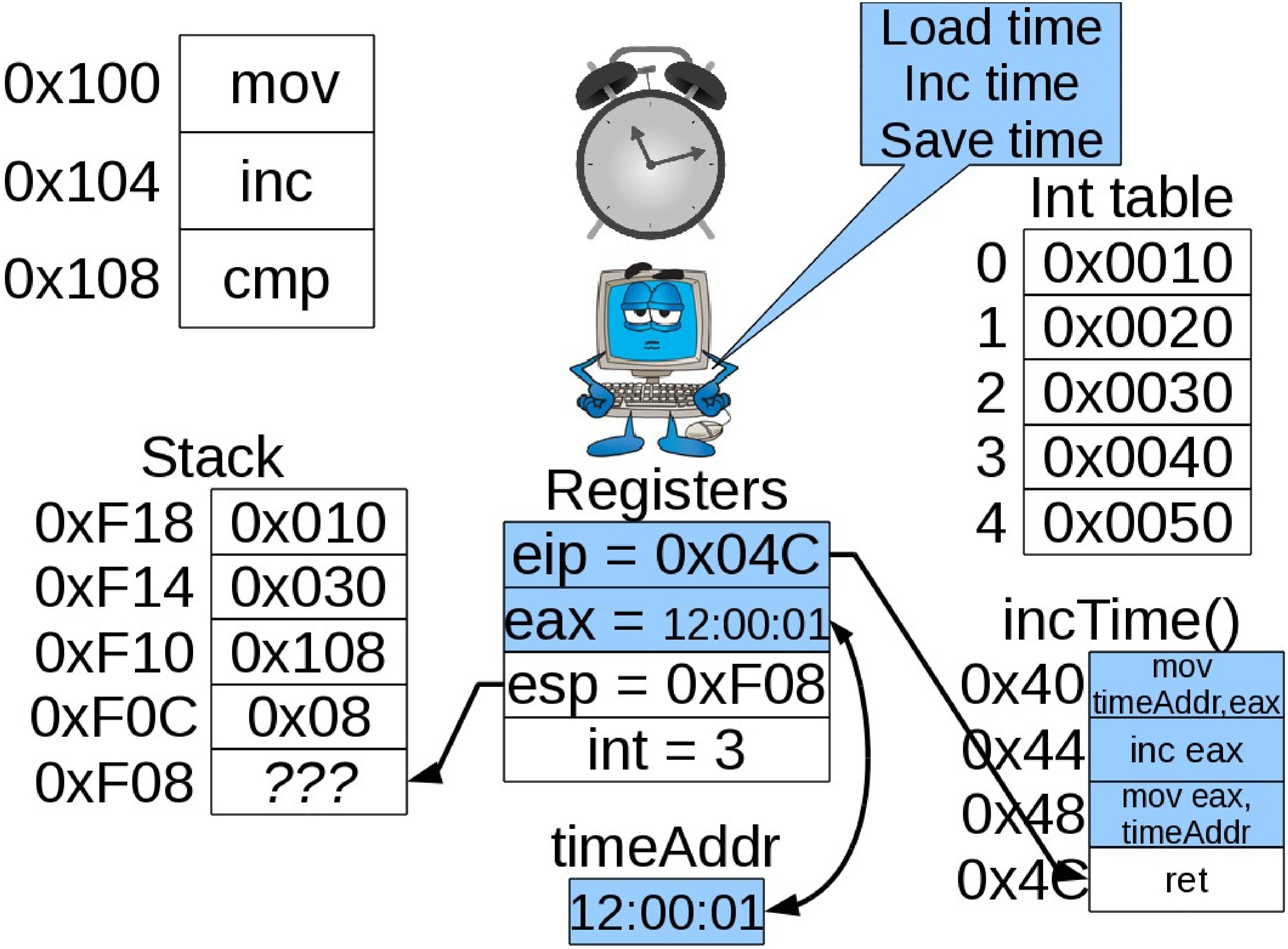
incTime()

0x40	mov timeAddr,eax
0x44	inc eax
0x48	mov eax, timeAddr
0x4C	ret

timeAddr

12:00:00

Example interrupt (5)



0x100	mov
0x104	inc
0x108	cmp

Load time
Inc time
Save time

Int table

0	0x0010
1	0x0020
2	0x0030
3	0x0040
4	0x0050

Stack

0xF18	0x010
0xF14	0x030
0xF10	0x108
0xF0C	0x08
0xF08	???

Registers

eip = 0x04C
eax = 12:00:01
esp = 0xF08
int = 3

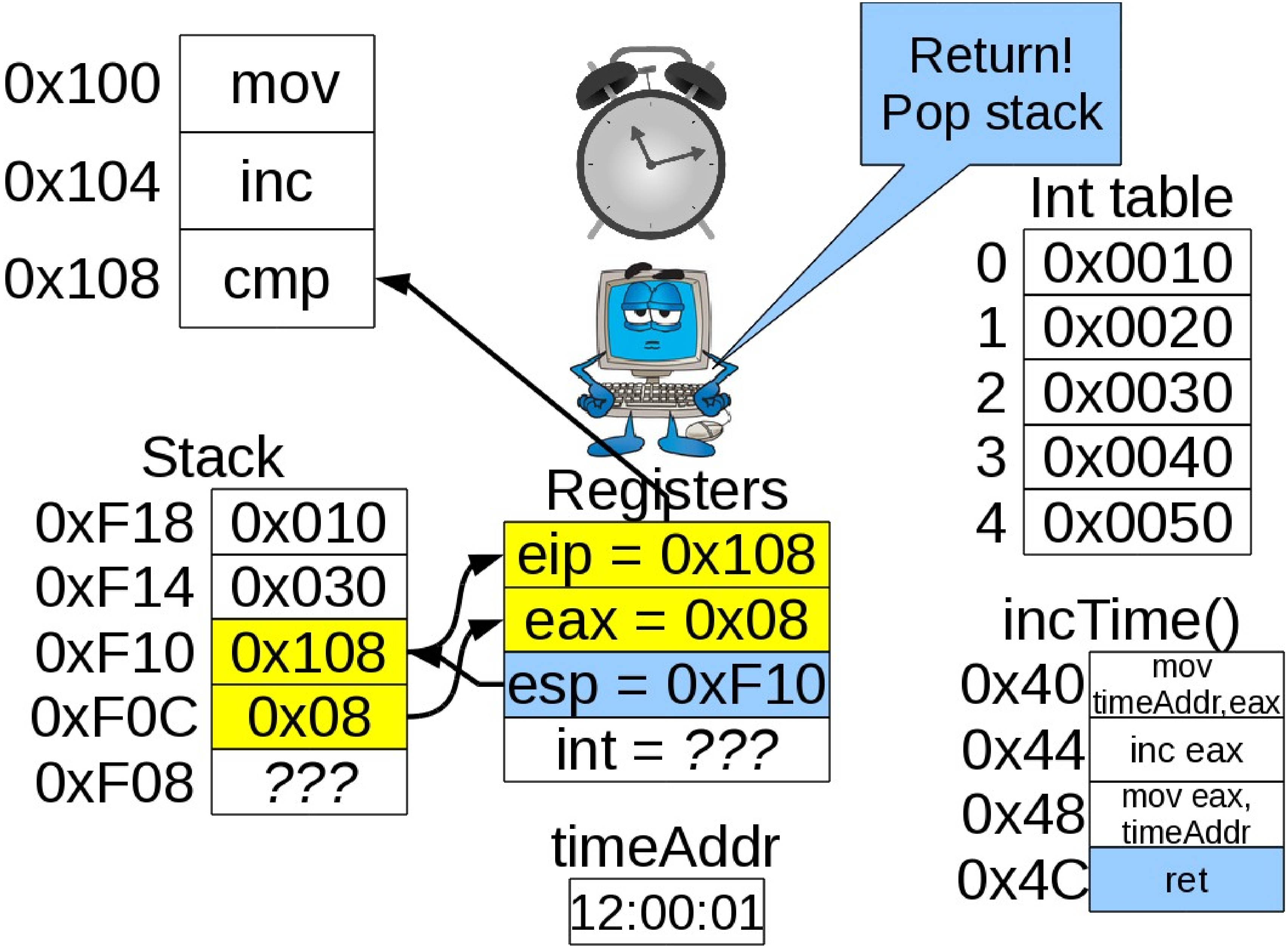
timeAddr

12:00:01

incTime()

0x40	mov timeAddr,eax
0x44	inc eax
0x48	mov eax, timeAddr
0x4C	ret

Example interrupt (6)



Return!
Pop stack

0x100	mov
0x104	inc
0x108	cmp

Stack

0xF18	0x010
0xF14	0x030
0xF10	0x108
0xF0C	0x08
0xF08	???

Registers

eip = 0x108
eax = 0x08
esp = 0xF10
int = ???

timeAddr

12:00:01

Int table

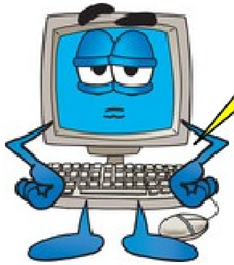
0	0x0010
1	0x0020
2	0x0030
3	0x0040
4	0x0050

incTime()

0x40	mov timeAddr,eax
0x44	inc eax
0x48	mov eax, timeAddr
0x4C	ret

Example interrupt (7)

0x100	mov
0x104	inc
0x108	cmp



Where was I?
Oh yeah . . .
cmp

Int table

0	0x0010
1	0x0020
2	0x0030
3	0x0040
4	0x0050

Stack

0xF18	0x010
0xF14	0x030
0xF10	0x108
0xF0C	0x08
0xF08	???

Registers

eip = 0x108
eax = 0x08
esp = 0xF10
int = ???

incTime()

0x40	mov timeAddr,eax
0x44	inc eax
0x48	mov eax, timeAddr
0x4C	ret

timeAddr

12:00:01

Your turn!

- The vector table on the original IBM PC was writable by any program.
- It also has a timer.
- You want to take some action every N seconds
- **Question 1**: How would you do so without messing up system time?
- **Question 2**: Is this an elegant solution?

Traps

- Intentional exceptions (*e.g.* system calls)
  - Linux uses ***int 0x80*** for all system calls

```
$ gcc -static hiWorld.c -o hiWorld
$ objdump -d -j .text hiWorld > textSeg.asm
$ grep "int " textSeg.asm
```

- Open up ***textSeg.asm*** and see interrupts:

```
8049109: 89 f1                 mov     %esi,%ecx
804910b: 89 fa                 mov     %edi,%edx
804910d: b8 92 00 00 00        mov     $0x92,%eax
8049112: cd 80                 int     $0x80
```

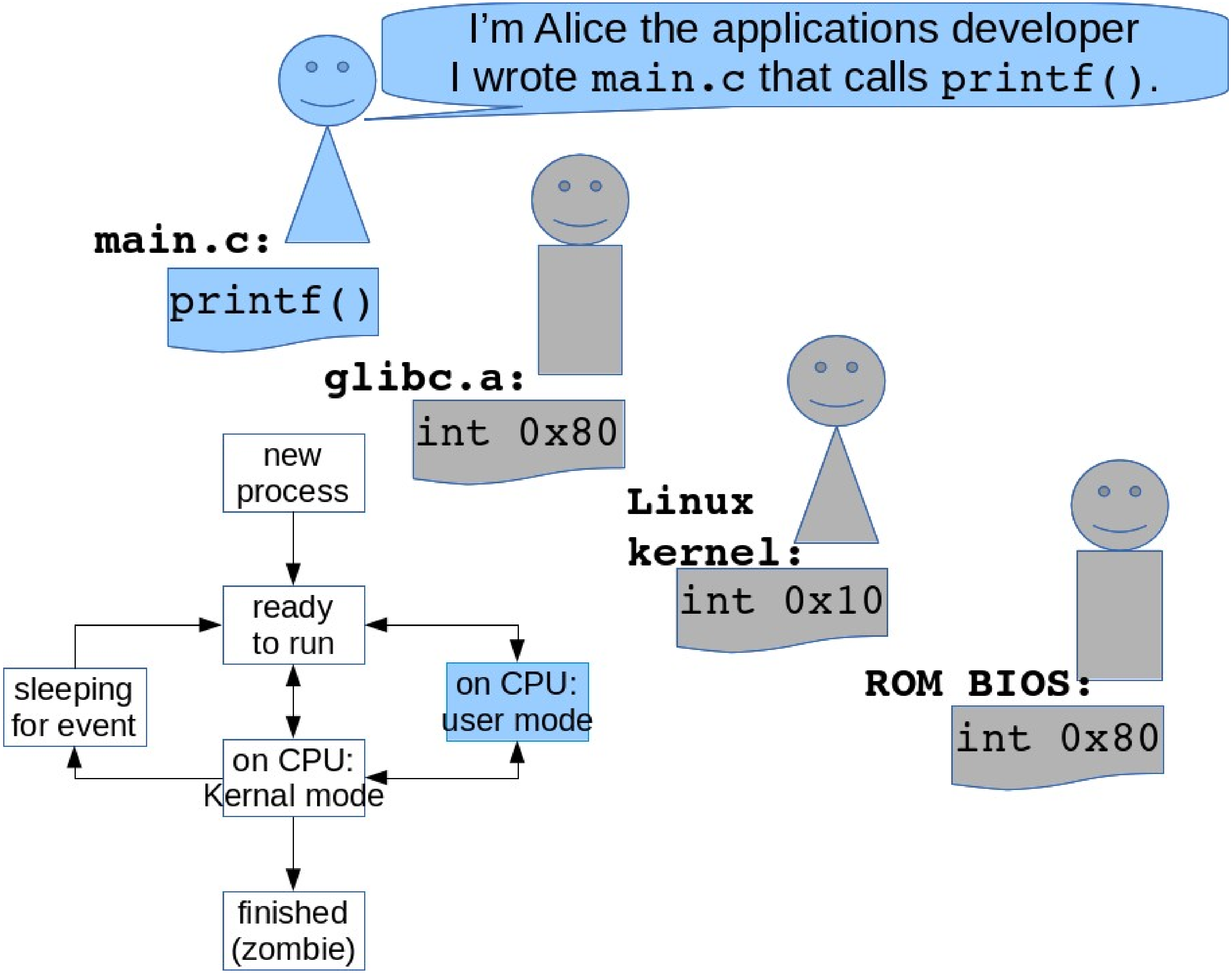
- **Question**: If Linux always uses **`int 0x80`** how does it know what service to perform?
  - **Hint**: Look carefully at its preceding code

Traps, cont'd

- Look in:
  - **`/usr/include/sys/syscall.h`**, redirected to
  - **`/usr/include/asm/unistd.h`**, finally to
  - **`/usr/include/asm/unistd_32.h`**
- There you'll see:

```
#define __NR_restart_syscall      0
#define __NR_exit                 1
#define __NR_fork                 2
#define __NR_read                 3
#define __NR_write                4
#define __NR_open                 5
#define __NR_close                6
#define __NR_waitpid              7
#define __NR_creat                8
. . . .
```

Doing a trap (system call): The same process in kernel mode



I'm Alice the applications developer
I wrote `main.c` that calls `printf()`.

`main.c:`

```
printf()
```

`glibc.a:`

```
int 0x80
```

`Linux kernel:`

```
int 0x10
```

`ROM BIOS:`

```
int 0x80
```

new process

ready to run

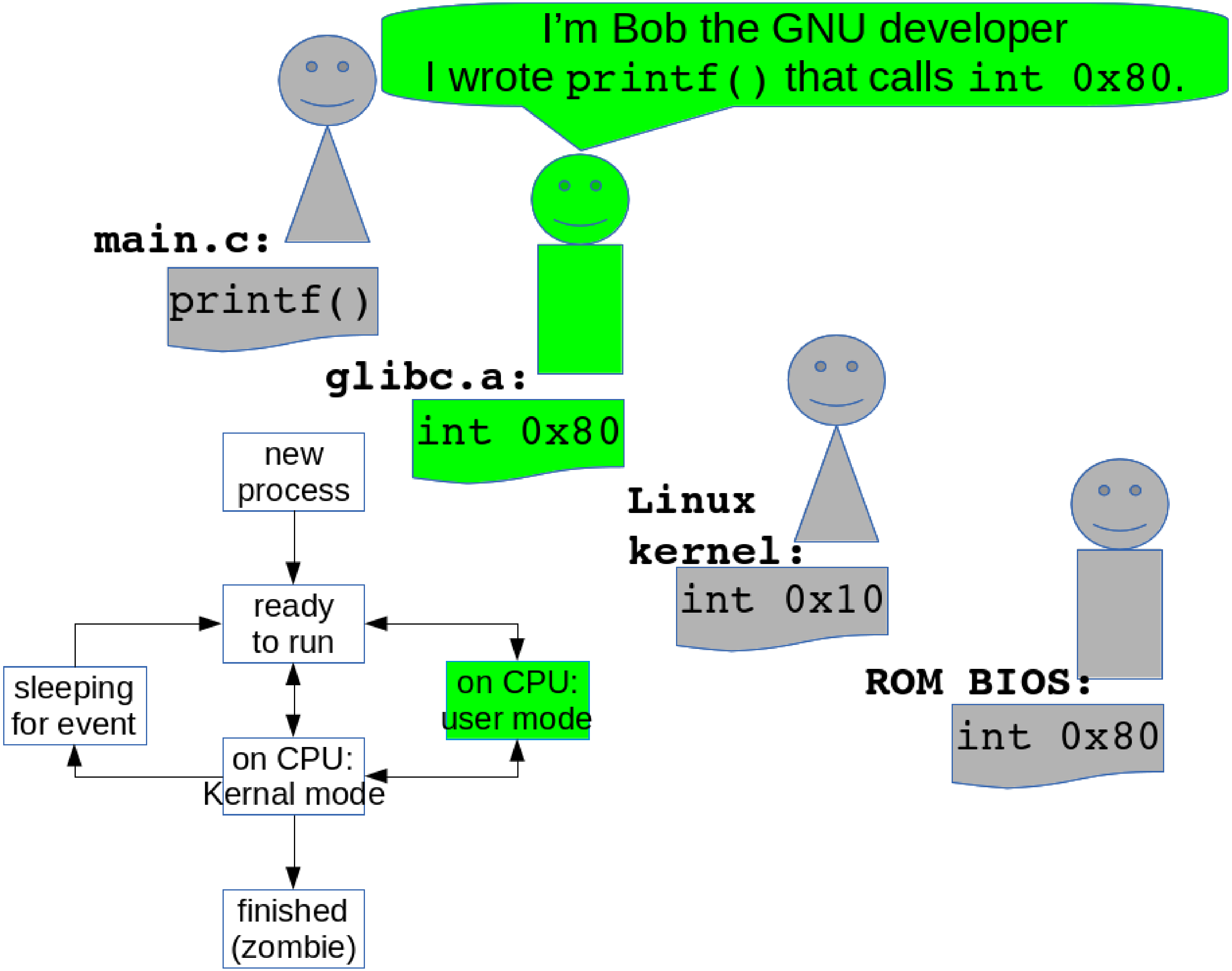
sleeping for event

on CPU: user mode

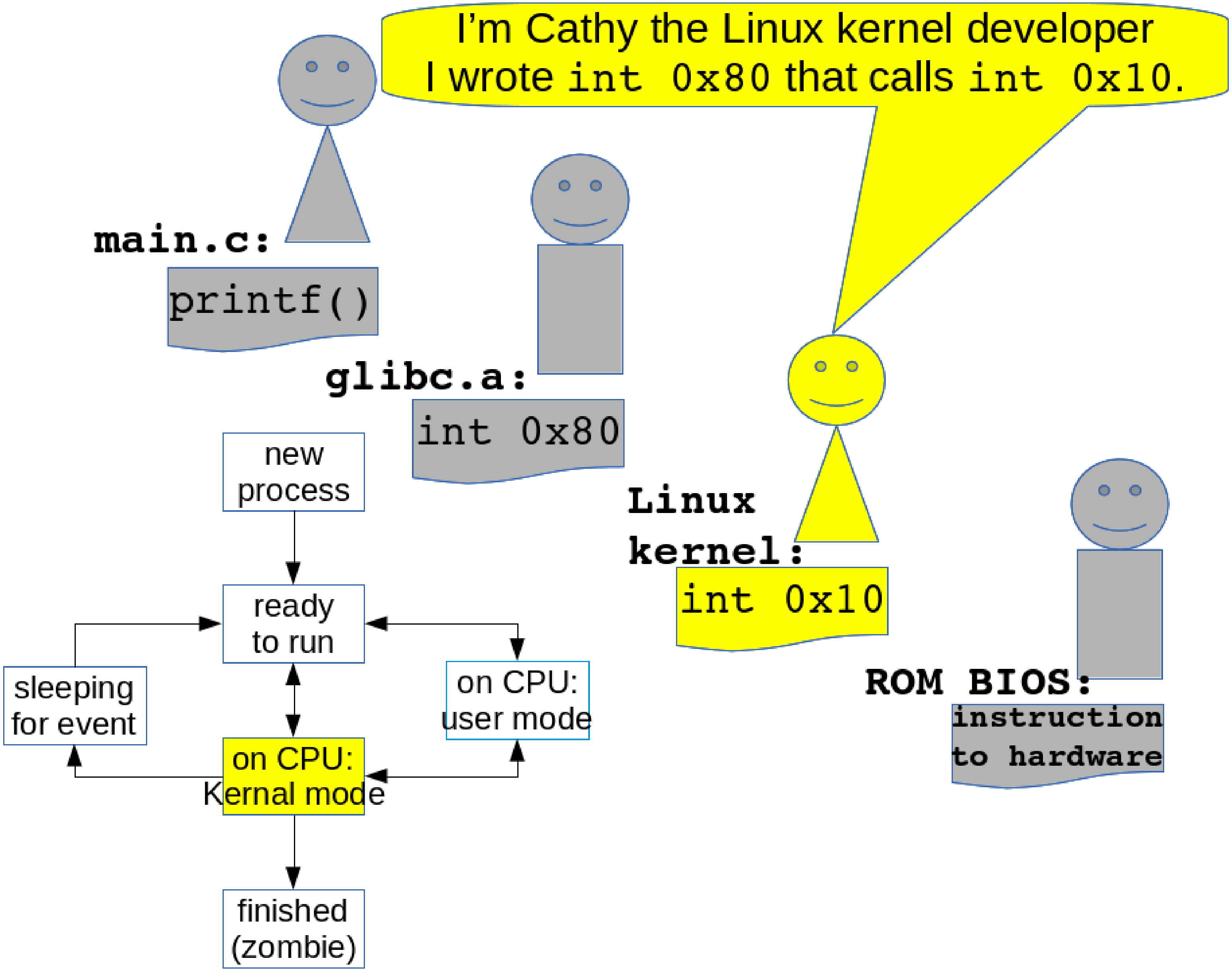
on CPU: Kernal mode

finished (zombie)

Doing a trap (system call): The same process in kernel mode



Doing a trap (system call): The same process in kernel mode



I’m Cathy the Linux kernel developer
I wrote `int 0x80` that calls `int 0x10`.

main.c:

```
printf()
```

glibc.a:

```
int 0x80
```

Linux kernel:

```
int 0x10
```

ROM BIOS:

instruction to hardware

new process

ready to run

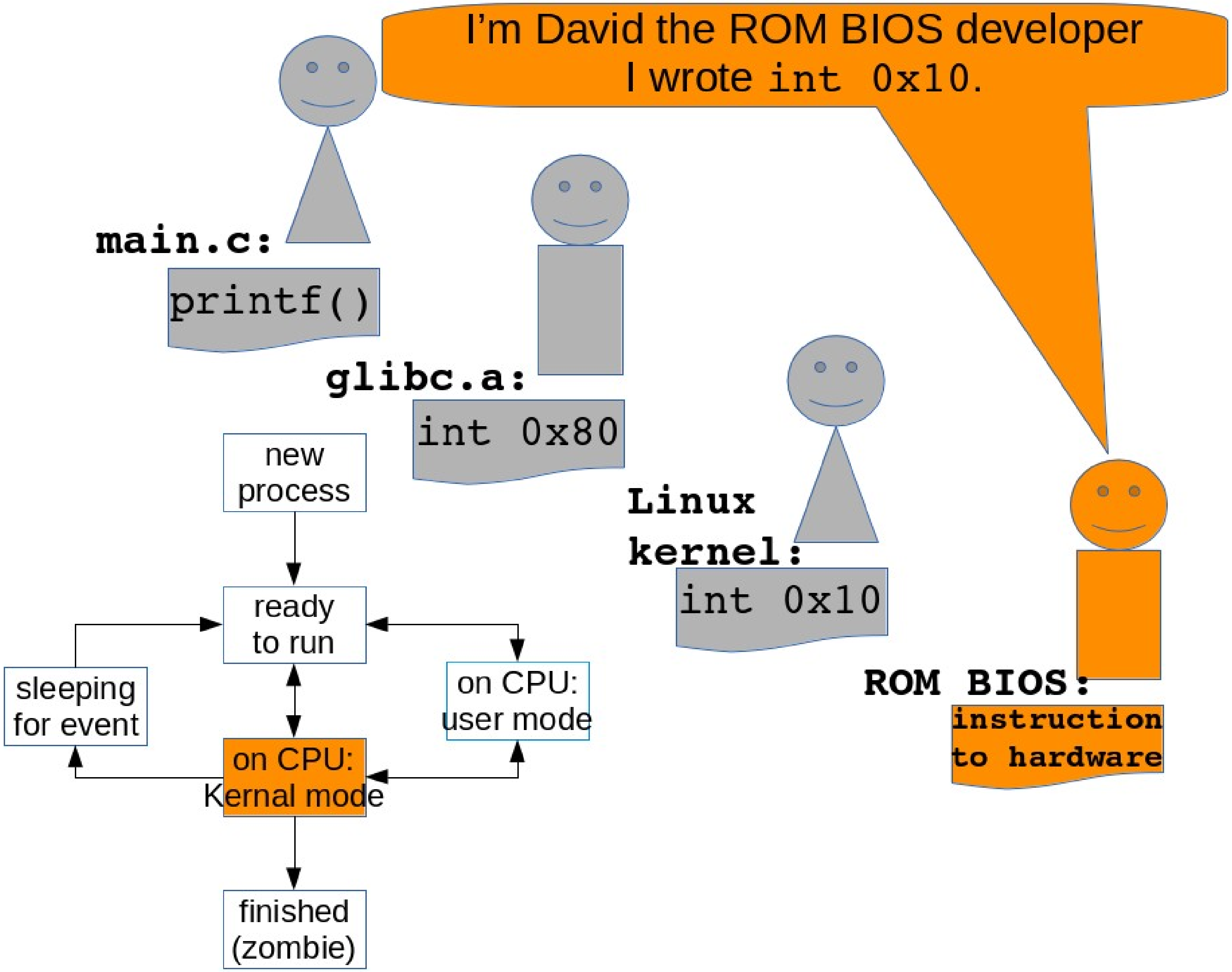
sleeping for event

on CPU: user mode

on CPU: Kernal mode

finished (zombie)

Doing a trap (system call): The same process in kernel mode



YOUR TURN!

This is such a good idea that Microsoft does it too!
There is an application level
There is a OS level (*e.g*. Windows 8)
There is, of course, the ROM BIOS.

What is their equivalent of glibc.a/libc.c?

Fault

- Waiting on some system-allocated resource:
  - Example: memory (in a virtual memory environment):

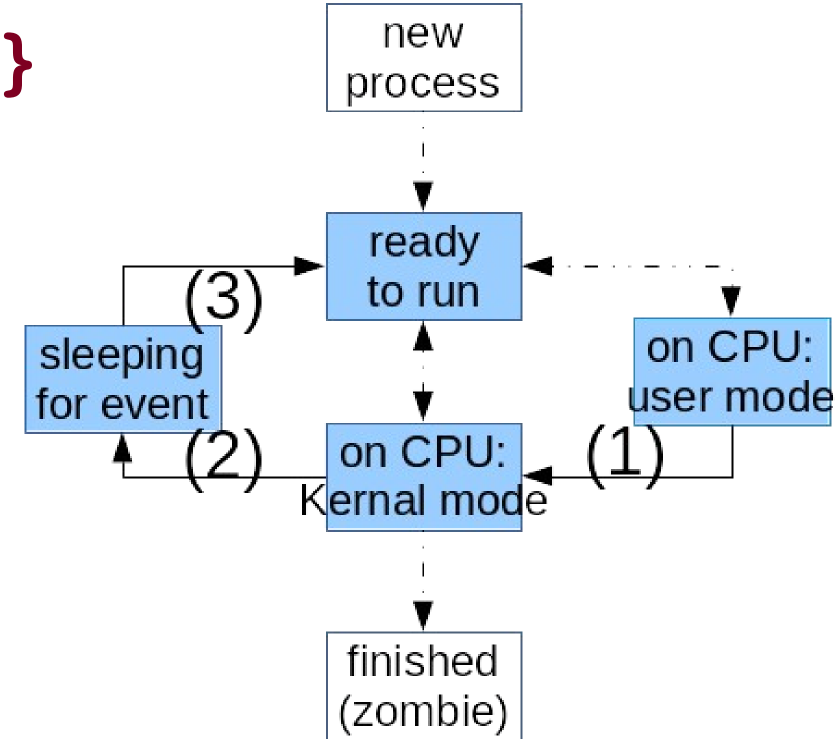
```
int main()
{
  int bigArray[5000];

  bigArray[4999] = 4998;
  // Page fault when not in memory
}
```

Fault (cont'd)

```
int main()
{
   int bigArray[5000];

   bigArray[4999] = 4998;
   // Page fault when not in memory
}
```



(1) "Oops, that page is not in memory, Let's get the OS to investigate"

(2) OS: "That page does belong to you. Let me put you to sleep while the harddrive loads it"

(3) OS "The harddrive just told me the page is loaded. I'll run you when I can."

Abort

- OS “You can't do ***that!***”
  - Writing to where you can't write to
  - Reading from where you can't read from

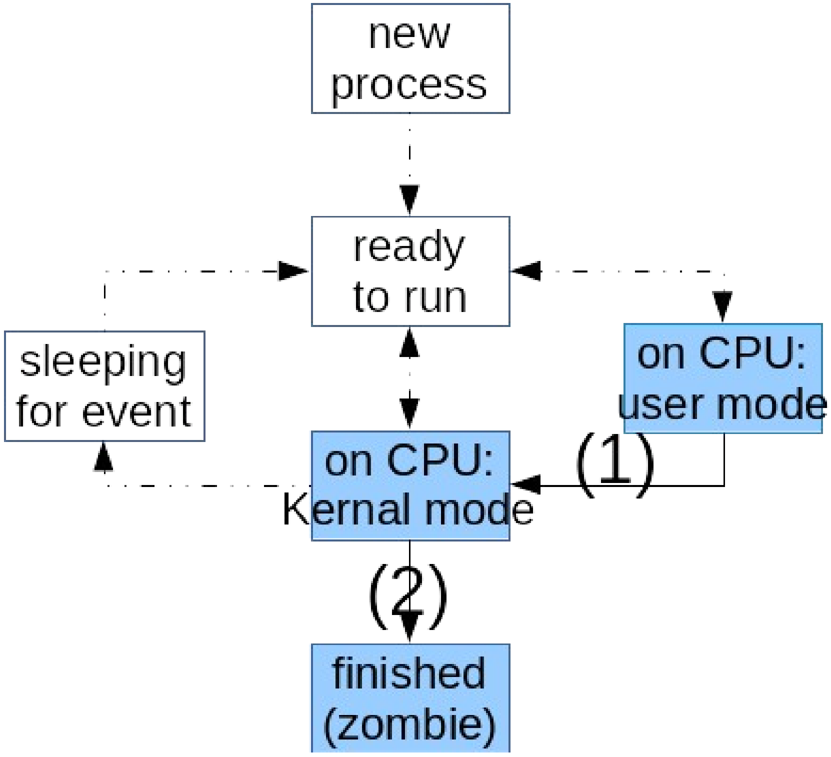
```
int main()
{
  int bigArray[5000];

  bigArray[-400000] = 4998;
  // Abort when addr not in page table
}
```

Abort (cont'd)

```
int main()
{
   int bigArray[5000];

   bigArray[-400000] = 4998;
   // Abort when addr not in pg table
}
```



(1) "Oops, that page is not in memory, Let's get the OS to investigate"

(2) OS: "Hey! That's not yours! I'm going to finish you off with a **SEGFAULT**, Mister!"

Your turn!

Can't the compiler or linker tell beforehand whether an instruction will be a trap or abort and do something about it at compile or link time?

Yeah yeah, other than `printf()` and `fgets()` what other system calls can I do?

- Among the most powerful things you can do are:
  - Make ***Brand New*** processes! (This lecture)
  - Tell your processes how to behave when they receive ***signals***! (Next lecture)

`fork()`, `getpid()` and `getppid()`

- **`fork()`** makes a baby process that is ***an exact copy*** of a mama process
- **`fork()`** returns type **`pid_t`** (usually an **`int`**):
  - -1: “***Too many processes, fool!***”
  - 0: “***I must be the baby***”
  - (positive number): “***I'm the mama and I got my baby's number***”
- **Question**: How does a baby know its number?
  - **Answer**: **`getpid()`**
- **Question**: How does a baby know its mama?
  - **Answer**: **`getppid()`**

fork() example 1

```
#include <stdlib.h>
#include <stdio.h>
int main ()
{
  int x = 1;
  pid_t pid = fork();

  if  (pid == 0)
    printf("Child has x = %d\n",++x);
  else
    printf("Parent has x = %d\n",--x);

  printf("Bye from process %d with x = %d\n",
         getpid(),x
        );
  return(EXIT_SUCCESS);
}
```

fork() example 2

```
// How many processes result from this?

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

int main ()
{
  puts("L0");
  fork();
  puts("L1");
  fork();
  printf("Bye from process %d\n",getpid());
  return(EXIT_SUCCESS);
}
```

Your turn!

- OS's are cheap ***and*** lazy. If they can get away with letting the mama and baby process share a segment, they will.
- Which segments can mama and baby share?
  - **`.text`**
  - **`.rodata`**
  - **`.data`** and **`.bss`**
  - Heap
  - Dynamic libraries
  - Stack

Your turn, again!

- Write programs to prove that the mama and baby have distinct:
  - **`.data`** segments
  - **`.bss`** segments
  - Stacks
  - Heaps

`exit()`, a way to end the process without returning to `main()`

- **`exit(EXIT_SUCCESS)`**
  - Everything went fine
  - Integer value 0
- **`exit(EXIT_FAILURE)`**
  - *Uhoh!* An error occurred, but at least I did not crash
  - Integer value 1
- Also doable as **`return(EXIT_SUCCESS)`** or **`return(EXIT_FAILURE)`** within **`main()`**

`execl()`: same process, different program

- Same process running different program:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
int     main    (int argc, char* argv[])
{
  char  line[100];
  printf("What program do you want to run? ");
  fgets(line,100,stdin);
  char* cPtr = strchr(line,'\n');
  if  (cPtr != NULL)  *cPtr = '\0';
  execl(line,line,NULL);
  printf("Will you see this?\n");
  return(EXIT_SUCCESS);
}
```

Calling **`execl()`**

-
  ```
  execl
  ("./myProgram",
   "myProgram",
   "first arg",
   "2nd arg",
   "3rd arg",
   NULL
  )
  ```

- **`"./myProgram"`**
  - So OS can find prog to run
- **`"myProgram"`**
  - So process knows name of program it runs
- Give as many arguments as you want
  ```
  "first arg",
  "2nd arg",
  "3rd arg"
  ```
  - All are ***strings!***
- ***Must*** end with **`NULL`**
  - Means “No more arguments”

. . . and being called by `execl()`

- **`argc`** tells how many command line arguments there all
  - It is almost always at least 1, why?
- **`argv[]`** is an array that tells the argument strings
- By convention, almost always called **`argc`** and **`argv[]`**

```
#include  <stdlib.h>
#include  <stdio.h>
int  main (int argc, char*  argv[])
{
  printf("%d parameter(s) which are:\n",argc);
  int i;
  for  (i = 0;  i < argc;  i++)
    printf("argv[%d] = %s\n",i,argv[i]);
  return(EXIT_SUCCESS);
}
```

Your turn!

`fork()` and **`execl()`** together:
Write a program where the mama process asks for a program to run, and the baby process runs it.

“*But wait!* There's more . . .”

- There are several versions of `exec*()`.
- All do the same thing with different parameters:
  - *E.g*. **`execv()`** lets you pass a NULL-terminated array instead of listing all arguments in command:
  - Do ***`$ man execl`*** for more details

```
#include <unistd.h>

extern char **environ;

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ...,
           char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],
            char *const envp[]);
```

Almost a shell!

- We have to tell the mama process to *wait* for the baby process
- **`pid_t wait(int* statusIntPtr)`**
  - How fortuitous!
  - Make mama wait until some baby finishes
  - Return the process id of the baby that finishes
  - Sets pointed-to address equal to an integer that encodes **`exit()`** integer *and* if it exited properly
    - **`WIFEXITED(statusInt)`**: non-0 means “okay”, 0 means “error”
    - **`WEXITSTATUS(statusInt)`**: Lowest byte of integer returned by **`exit()`**.

WIFEXITED() & WEXITSTATUS() 1

- To see **WIFEXITED()** and **WEXITSTATUS()** in action, let’s first make 3 child programs:

```
//  iSucceed.c
#include <stdlib.h>
int main () { return(EXIT_SUCCESS); }

//  iFail.c
#include <stdlib.h>
int main () { return(EXIT_FAILURE); }

//  iCrash.c
#include <stdlib.h>
int main () { char* cPtr = NULL;  (*cPtr)++; }
```

WIFEXITED() & WEXITSTATUS() 2

```
//  Now let us make a parent that runs all three

#include        <stdlib.h>
#include        <stdio.h>
#include        <unistd.h>
#include        <wait.h>

void    runChild(const char* progName);

int     main    ()
{
  runChild("./iSucceed");
  runChild("./iFail");
  runChild("./iCrash");
  return(EXIT_SUCCESS);
}
```

WIFEXITED() & WEXITSTATUS() 3

```
void runChild(const char* progName)
{
  int   status;
  pid_t childId;

  printf("Running %s:\n",progName);

  if  (fork() == 0)
  {
    //  The child case:
    execl(progName,progName,NULL);
    fprintf(stderr,"Could not find %s\n",progName);
    exit(EXIT_FAILURE);
  }
```

WIFEXITED() & WEXITSTATUS() 4

```
  //  The parent case:
  childId = wait(&status);

  if  ( WIFEXITED(status) )
  {
    if  (WEXITSTATUS(status) == EXIT_SUCCESS)
      printf("%d succeeded.\n\n",childId);
    else
      printf("%d failed but didn’t crash.\n\n",
             ChildId
            );
  }
  else
    printf("%d crashed.\n\n",childId);

}
```

Your turn again!

Revise our almost-shell program so that the mama **`wait()`**s until for baby to finish.

Waiting for the *right* baby

- **`wait()`** waits until some baby (the first baby) finishes
- Want to wait for a ***specific*** baby?

  ```
  pid_t waitpid(pid_t childId, int*
  statusIntPtr, 0)
  ```

- Want ***not*** to wait ***if no baby has finished***?

  ```
  pid_t waitpid(pid_t childId, int*
  statusIntPtr, WNOHANG)
  ```

- Want to wait for ***any old baby*** just like ***`wait()`***?

  ```
  pid_t waitpid(-1, int* statusIntPtr, 0)
  ```

`waitpid()` example, pg 1

```
/* Setup */
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

const int N = 16;

int main ()
{
  pid_t pid[N];
  int   i;
  int   status;
```

waitpid() example, pg 2

```
/* Making babies
   How many second does the FIRST baby wait?
   How many second does the LAST  baby wait?
 */
for  (i = 0;  i < N;  i++)
  if  ((pid[i] = fork()) == 0)
  {
    /* Child case */
    sleep(N-i);
    return(100+i);
  }
```

waitpid() example, pg 3

```
  /* waiting for babies (babysitting?) */
  for  (i = 0;  i < N;  i++)
  {
    pid_t childPid = waitpid(pid[i],&status,0);

    if  (WIFEXITED(child_status))
      printf("Child %d ended with status %d\n",
             childPid,WEXITSTATUS(status)
            );
    else
      printf("Child %d ended abnormally\n",
             childPid);
  }

  return(EXIT_SUCCESS);
}
```

Child neglect

1. Mama **`fork()`**s a baby
2. The baby does its job and finishes
3. Baby takes no CPU time (it's finished) but there is still an entry in process table (so mama can get return status, *etc*.)
4. But mama ***ignores*** finished baby

The Baby becomes a ZOMBIE!

Zombie survival guide

- A finished process with an entry in process table is “not-quite” dead. It's a zombie!
- The not-so-bad:
  - Zombies take no CPU time (they're finished)
- The pretty-bad:
  - Zombies do take System memory (in process table)
  - **One zombie?** *No big deal*.
  - **An ARMY of zombies?** *System starts running vvvveeerrrryyyy sssslllloooowwwwllllyyyy*
  - Might have to reboot system (*Is this a problem with any OS you know?* ;)

zombie_maker.c, page 1

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main ()

{
  int child_status;

  if (fork() == 0)
  {
    printf("HC: hello from child %d\n",getpid());
    return(50);
  }
```

zombie_maker.c, page 2

```
  else
  {
    printf("HP: hello from parent\n"
           "In another shell say 'ps aux'\n"
           "Look for '<defunct>'.\n"
          );
    sleep(20);
    wait(&child_status);
    printf("CT: child has terminated and has given
us %d\n",
           WEXITSTATUS(child_status)
          );
  }
  printf("Bye\n");
  return(EXIT_SUCCESS);
}
```

We don’t have time now, but also check out

- **`nice`**: A way to make your program use less resources
- **`nohup`**: A way to make your still run after your terminal has ended
- **&**: A way to make a child program run “in the background”, and have the shell immediately come back.
- **`atexit()`**: A way to register functions to run after **`main()`** has finished.
- **`execlp()`**, **`execle()`**, **`execvp()`**, **`execvp()`**: Similar to **`execl()`**.

Next time: *Signals!*

atexit(): A way to do things after main() ends

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

void cleanup ()
{
  printf("Cleaning up after %d\n",getpid());
}

int main ()
{
  atexit(cleanup);
  puts("Forking");
  fork();
  printf("Process %d's main() finished.\n",getpid());
  return(EXIT_SUCCESS);
}
```