CSC 374/407: Computer Systems II

Lecture 4

Joseph Phillips
De Paul University

2014 January 3

Reading

- Bryant & O'Hallaron "*Computer Systems, 2nd Ed.*"
  - Chapter 8: Exception Control Flow
- Hoover "*System Programming*"
  - System Calls 7.1-7.4

Topics

- Signals
- `setjmp()` and `longjmp()`

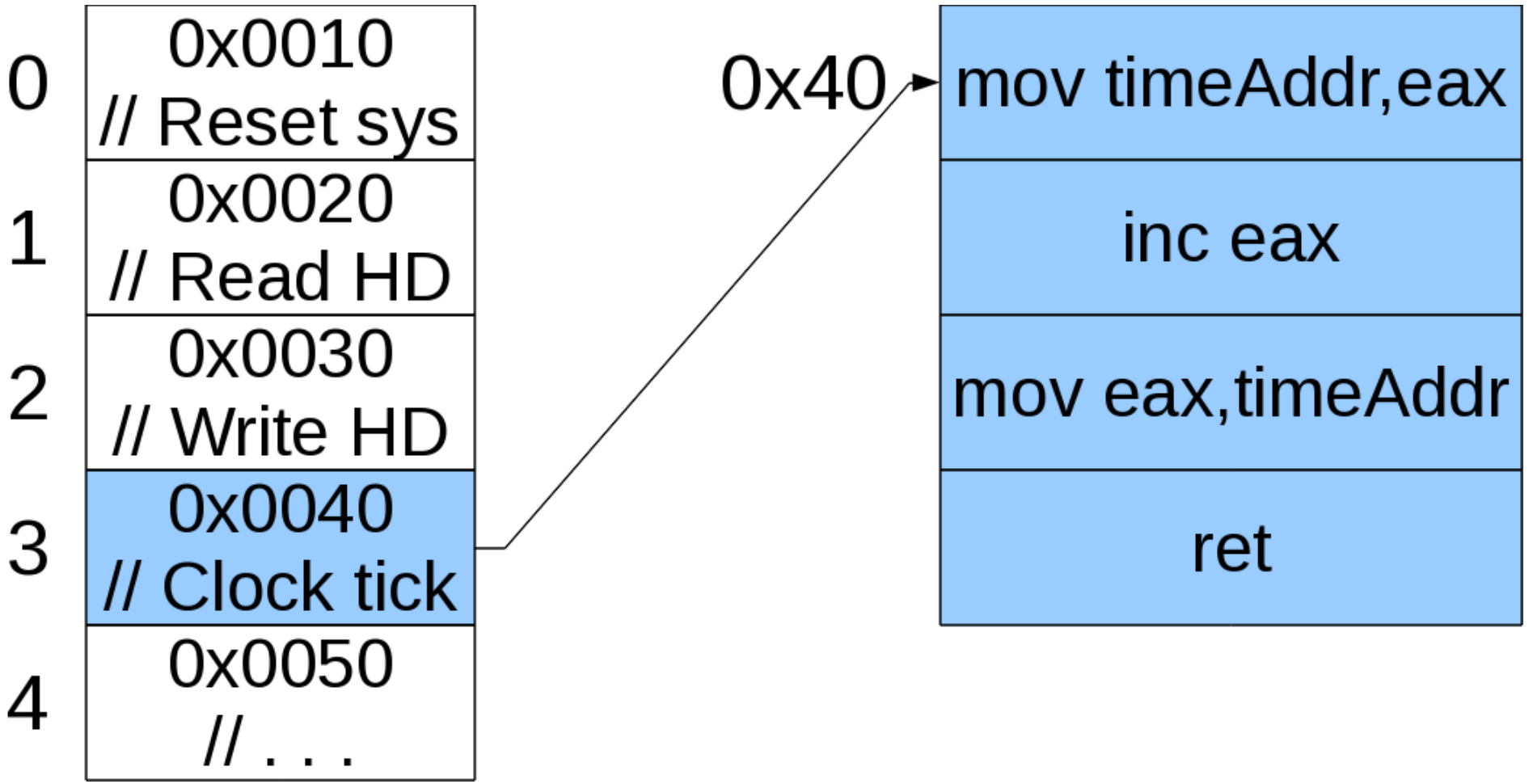
Remember the shell . . .

Let's re-write our simple shell program from last time.

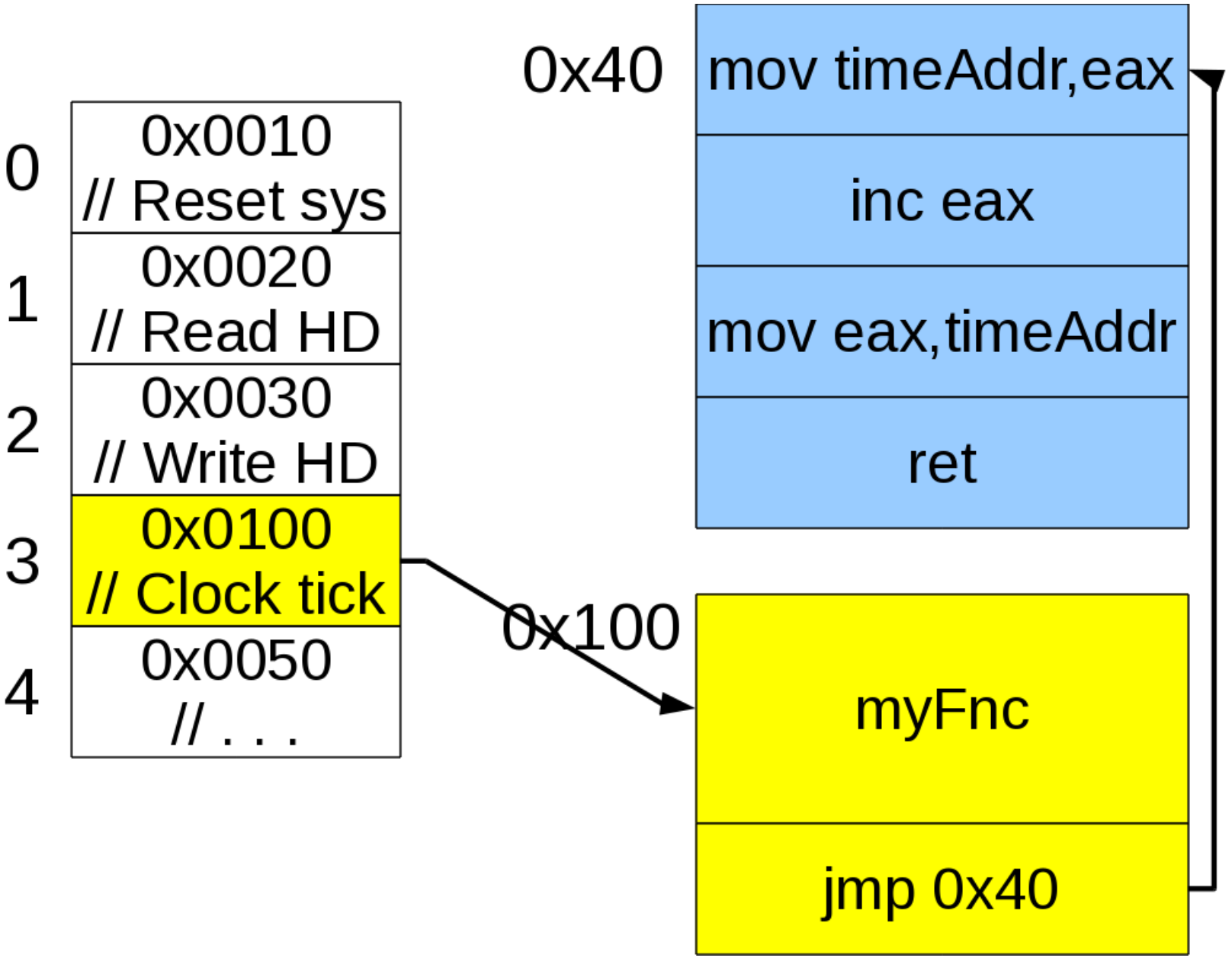
We may use:

- `fork()`
- `execl(char* path, char* arg0, char* arg1, . . .)`
- `wait(int* statusPtr)`
- `waitpid(pid_t pid, int* statusPtr, int flag)`

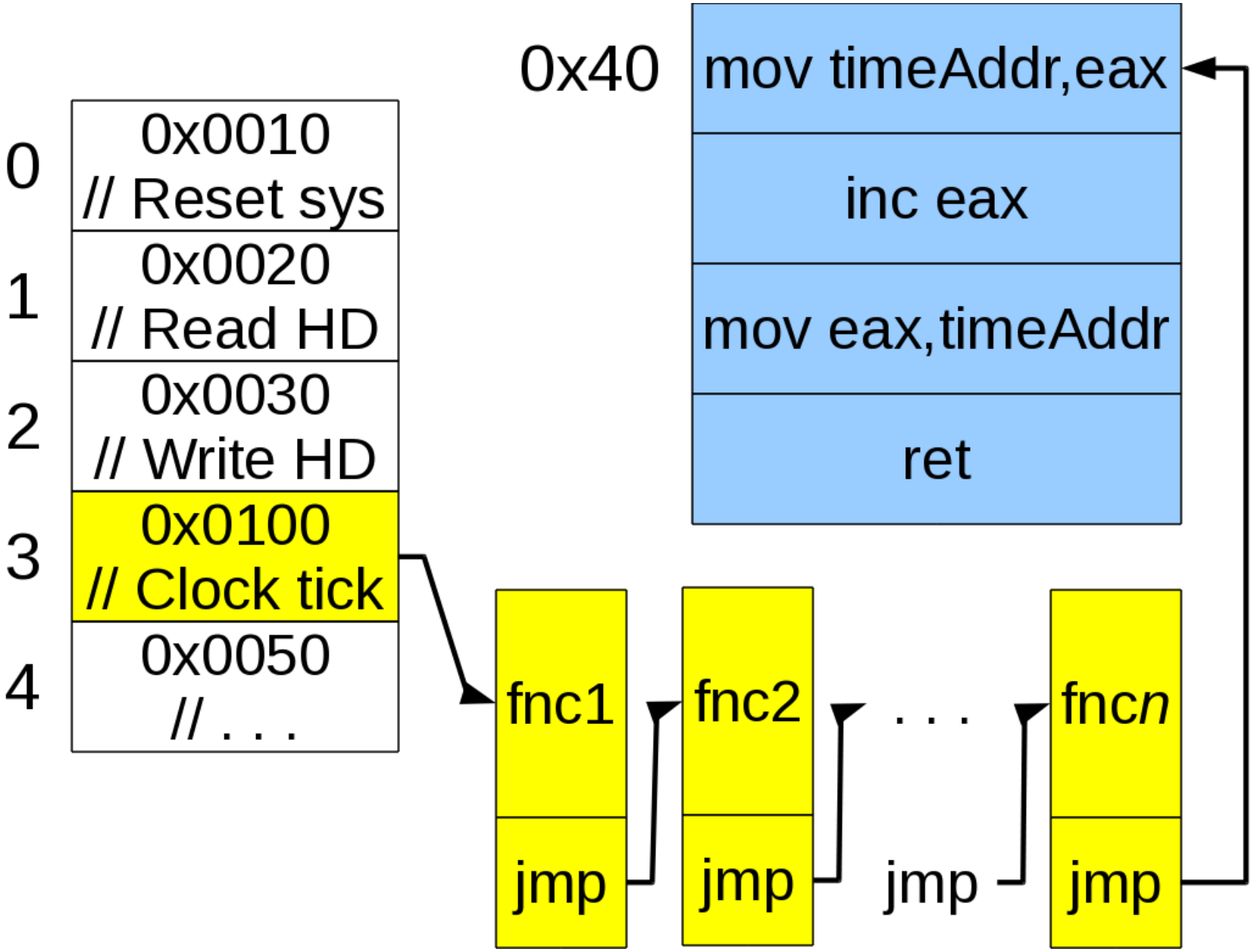
Remember: One interrupt vector table for whole computer



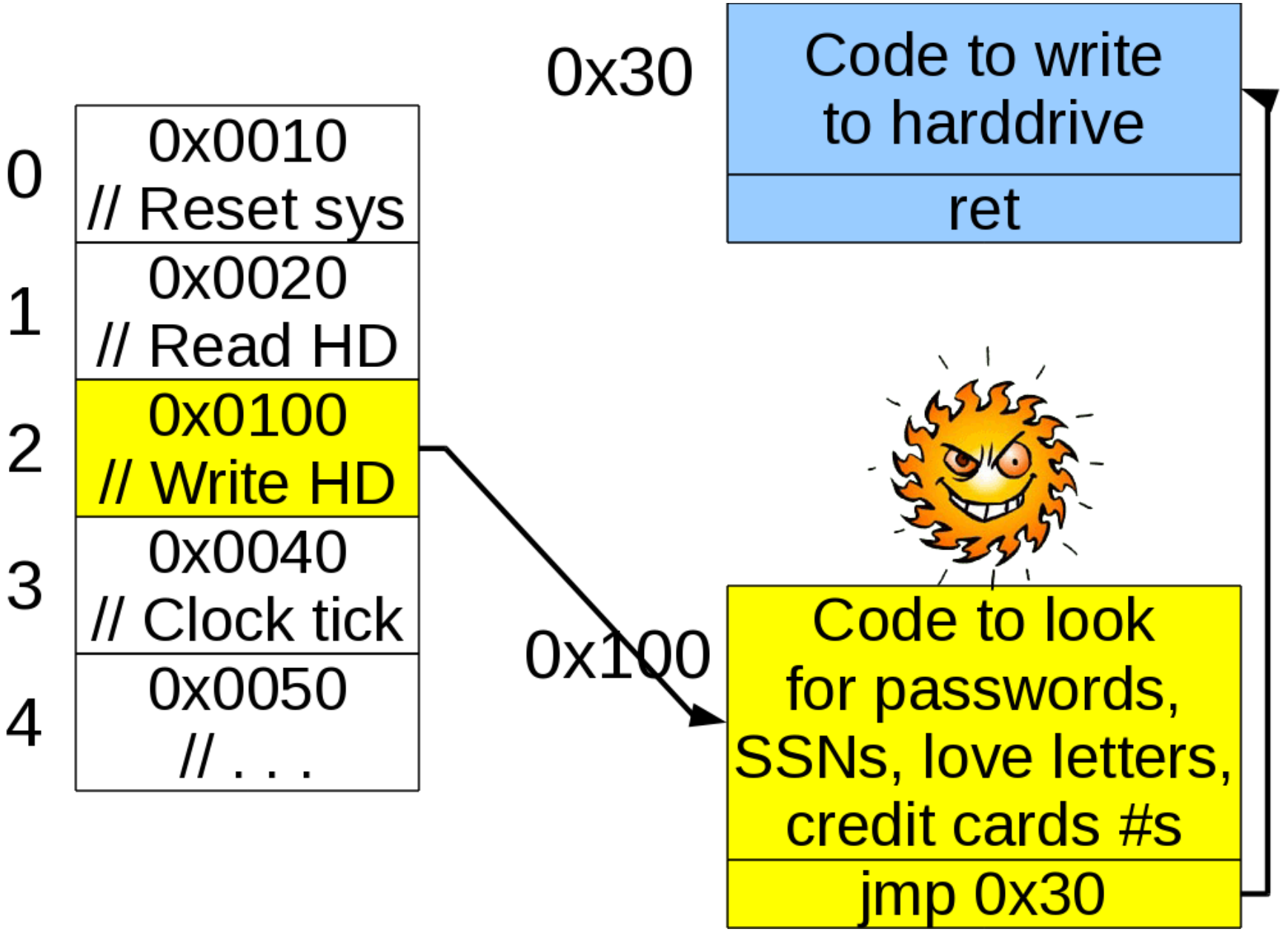
Remember how to do something periodically?



Is this technique efficient when there are many things to schedule?



Is this technique secure?



Signals: a better idea

The whole computer still has its interrupt table

- Nobody can mess with it except OS and ROM BIOS

Each process has its own "personal" interrupt table

It's "interrupts" include

- You set the alarm and went to `sleep()`, now it's ***Time to wake up!***
- ***Hey!*** One of your children just finished!
- Close things down and finish, *or less politely . . .*
- ***Die now!***

List of signals (Linux i386) (1)

The default actions are:

- `Term`: terminate the process
- `Core`: terminate the process and dump the core
- `Ign`: ignore the signal
- `Stop`: stop the process

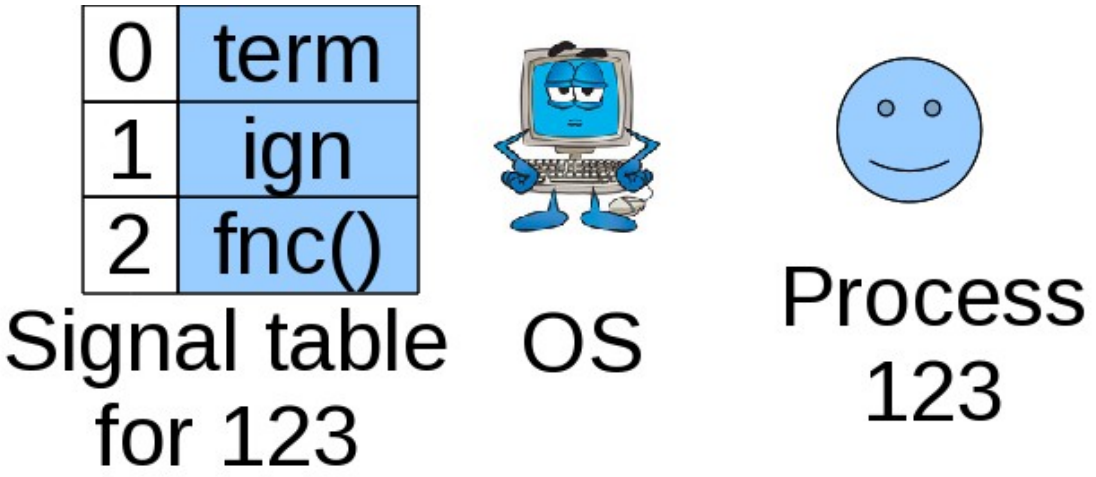
```
Signal  Value  Action   Comment
-----------------------------------
SIGHUP     1    Term    Hangup detected on controlling
                        terminal or death of controlling
                        process.
SIGINT     2    Term    Interrupt from keyboard (Ctrl-C)
SIGQUIT    3    Core    Quit from keyboard
SIGILL     4    Core    Illegal Instruction
SIGABRT    6    Core    Abort signal from abort(3)
SIGFPE     8    Core    Floating point exception
SIGKILL    9    Term    Kill signal
```

List of signals (Linux i386) (2)

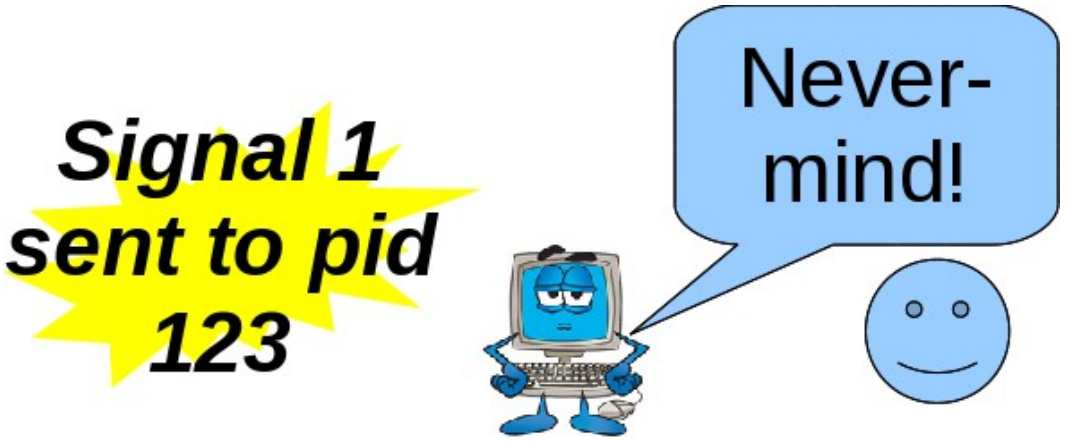
Signal	Value	Action	Comment
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	10	Term	User-defined signal 1
SIGUSR2	12	Term	User-defined signal 2
SIGCHLD	17	Ign	Child stopped or terminated
SIGCONT	18		Continue if stopped
SIGSTOP	19	Stop	Stop process
SIGTSTP	20	Stop	Stop typed at tty
SIGTTIN	21	Stop	tty input for background process
SIGTTOU	22	Stop	tty output for background process

How OS handles signals (1)

Each process has its own signal table:



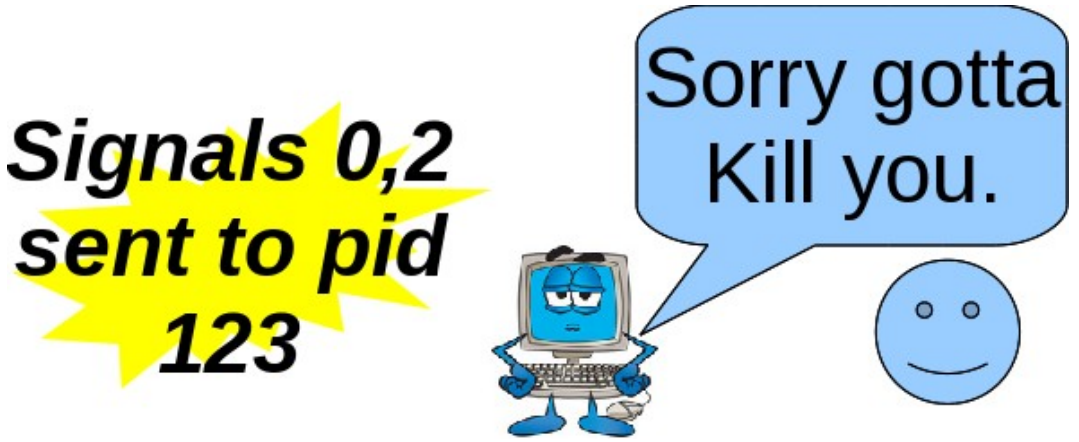
Some signals may be ignored or handled specially



How OS handles signals (2)

Question: What happens if multiple signals *of different number* arrive around same time?

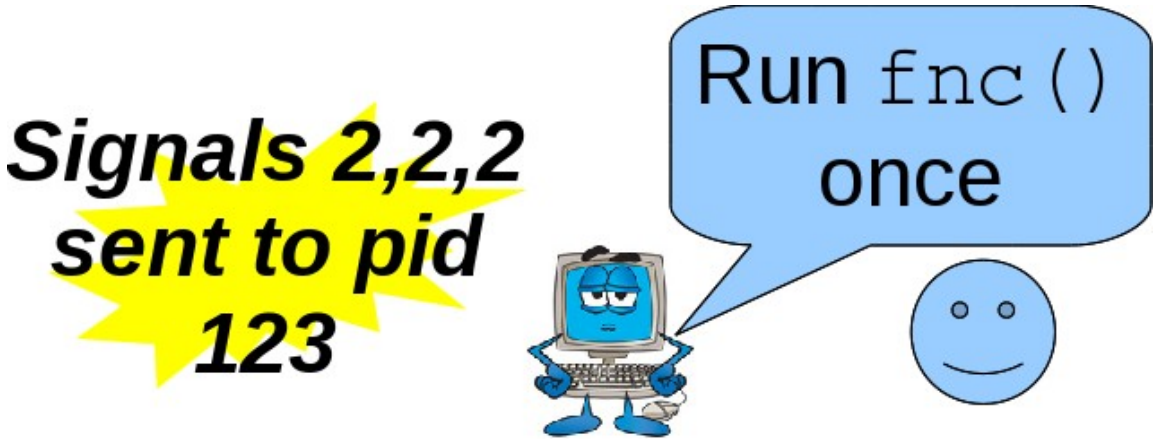
Answer: The lowest numbered signal takes precedence



How OS handles signals (3)

Question: What happens if multiple signals *of same number* are pending?

Answer: Because they are not buffered, any called routine will only be called once *and* should be clever enough to handle any number of waiting cases.



A look at those actions

Some signals are so serious that the process must ***die!***

- `SIGILL`: Illegal instruction (how could this happen?)
- `SIGSEGV`: Illegal memory reference (how could this happen?)
- Analogous to interrupt table's NMI (non-maskable interrupt) for very serious hardware problems like parity and checksum-like errors

Other signals are less serious

- `SIGCHLD`: A child has finished (Hey, wouldn't `wait()` or `waitpid()` ***always*** catch this for us?)

What does all this mean for me?

You the applications programmer get (some) control over your process' personal “interrupt table”

- ***Default action*** for this!
- ***Ignore*** that!
- Do it ***my way*** for the other!

Doing it!

```
#include <signal.h>

void    handler (int signalNum)
{
  //  Handling code here
}

int main ()
{
  signal(SIGINT,SIG_DFL); // Handle by default
  signal(SIGINT,SIG_IGN); // Ignore SIGINT
  signal(SIGINT,handler);
                 // Handle with handler()
  // All the cool coders now use sigaction()
  // (See appendix)
}
```

Your turn!

Write a program that

1. Prints "*`You can't stop me!  Ngyeah-ngyeah, ngyeah-ngyeah!`*"
2. Pauses for 2 seconds
3. Goes back to 1.

Indefinitely, and that ***cannot*** be stopped by `Ctrl-C`.

(***Question***: *Uh-oh, how can we stop it?*)

Sending signals to processes (1)

Oh no! The program from the previous slide is still running!

- `Ctrl-C` can't stop it!

If we could send `SIGKILL` (9) we could stop it, but how?

```
$ kill -9 <processId>
```

- ***Question***: How do we figure out the PID?

Forgive the *bloodthirsty* name, it should be called "**`sendSignal`**"

Sending signals to processes (2)

Process groups

- Each process belongs to a process group
- By default it's its parents group
- Can find out group `pid_t getpgrp()` in `unistd.h`.
- Can change group to one's own process id (or that of another process) `int setpgid (pid_t pid, pid_t pgid)`
- Can send signal to all processes in group by making process number negative

```
$ kill -9 -<processGroupId>
```

Back to the actions!

Some actions are un-ignorable and un-handle-able:

- ***QUESTION***: Name 2 or 3 which you think are.
- ***QUESTION***: Why?

Handlers should have form `void someName (int sigNum)`

- ***QUESTION***: Why is the return type `void`?
- ***QUESTION***: Why does it take the signal number as the sole parameter?

Your turn again!

Same as before, except this time each time the user does press `Ctrl-C` it randomly prints one of the following:

- "*`Ouch!`*"
- "*`Stop that!`*"
- "*`That hurts!`*"
- "*`Mercy!`*"

HINT: Use `switch (rand() %4) { .. }` to jump to cases 0 to 3 randomly

Sending signals to processes (3)

Application programs can signal each other too!

```
#include <sys/types.h>
#include <signal.h>

int kill (pid_t procToSignal, int sigToSend);
```

Again, forgive the *bloodthirsty* name, it should be called "**`sendSignal()`**"

Back to the shell

Our shell program always waited for the child process to finish.

Revise it so that it may have multiple children (*ie*. one of the children can run in the “background” along with the parent)

The background child process can finish at any time. (When it does `SIGCHLD` is sent to the parent, but by default this signal is ignored.)

Make sure you reap the child so it doesn't remain a zombie too long.

Back to the shell (2)

If multiple children finish around the same time will the different `SIGCHLD` signals be queued?

How can we revise our previous program to leave no zombies?

- ***HINT***: `wait()` quits if there is nothing for to wait.

SIGALRM

```
#include <unistd.h>

unsigned int alarm(unsigned int secs);
```

If **`secs>0`** then tells OS "*Send **`SIGALRM`** to me **`secs`** seconds in the future*"

If **`secs==0`** then tells OS "*Clear any alarms I may have set*"

Either case returns number seconds until next (now cleared) alarm, or 0 if there were none.

Your turn!

Write a program that:

(1) Let's the user type in how many seconds they want to wait until an alarm goes off

(2) Continually prints "***Tick-tock***" until the alarm goes off. Then it prints "***Ding-ding***" and stops.

(3) If the user press `Ctrl-C` then it prints how many seconds are left and waits for the user to press `Enter`. Then it goes back to (2).

`setjmp()` and `longjmp()`

Old school C way of doing error recovery

`int setjmp(jmp_buf j)`

- Memorize both position in code (`%eip`) and position on stack (`%esp`) inside buffer `j`.
- First time it's called returns 0 (so you know this is the installation case).

`void longjmp(jmp_buf j, int i)`

- ***Uh-oh!*** We have a hard-to-recover-from error!
- Set `%eax` to `i`, jump back to “safe” state described in `j`.

Use

```
jmp_buf j; // Put in global context

int main()
{
  if (setjmp(j) != 0)
  {
    // Handle error
  }
  attemptToDoSomethingErrorProne();
}

void attemptToDoSomethingErrorProne()
{
  if (haveMessedUp == 1)
    longjmp(j,1);
}
```

Abuse

```
/*  Question: Will this behave properly? */
jmp_buf j; // Put in global context

int main()
{
  foo();
  bar();
}

void foo ()
{
  if (setjmp(j) != 0)
  {
    // Handle error
  }
}
```

```
void bar ()
{
  if (haveMessedUp==1)
    longjmp(j,1);
}
```

Bottom line on `setjmp()` and `longjmp()`

If you use them be sure you longjmp() to a function that is still on the stack.

Question: What modern error-trapping technique is in C++, Java, etc. that makes this C construct not necessary?

Next time: *Threads!*

Appendix: `sigaction()`

`signal()` is *old school*. Nowadays folks prefer:

```
int sigaction(int signum,
 const struct sigaction *act,
       struct sigaction *oldact);
```

It lets you:

1. See what action is currently being taken for a signal
2. Selectively block other signals while you're in a signal handler
3. Other (esoteric) stuff, too

sigaction() (2)

#include <signal.h>

```
int sigaction(int signum, const struct sigaction
  *act,struct sigaction *oldact);
```

Where:

```
struct sigaction
{
  void (*sa_handler)(int); // Signal handler
  void (*sa_sigaction)(int, siginfo_t*, void*);
                              // Funkier sig handler
  sigset_t   sa_mask; // Sigs to allow in handler
  int        sa_flags;//
  void       (*sa_restorer)(void);
};
```

sigaction() (3)

Ex: Install SIGINT handler if not already ignoring:

```
int    main ()
{
  struct sigaction newAction, oldAction;

  //Define ctrlCHandler that wont block other sigs
  newAction.sa_handler = ctrlCHandler;
  sigemptyset (&newAction.sa_mask);
  newAction.sa_flags = 0;

  // See what is currently done for SIGINT
  sigaction (SIGINT, NULL, &oldAction);

  // Install new handler if not currently ignoring
  if (oldAction.sa_handler != SIG_IGN)
    sigaction (SIGINT, &newAction, NULL);
  . . .
```

`sigaction()` (4)

Lots more detail!

See ***`$ man sigaction`***