CSC 374/407: Computer Systems II

Lecture 4

Joseph Phillips
De Paul University

2014 February 2

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()`

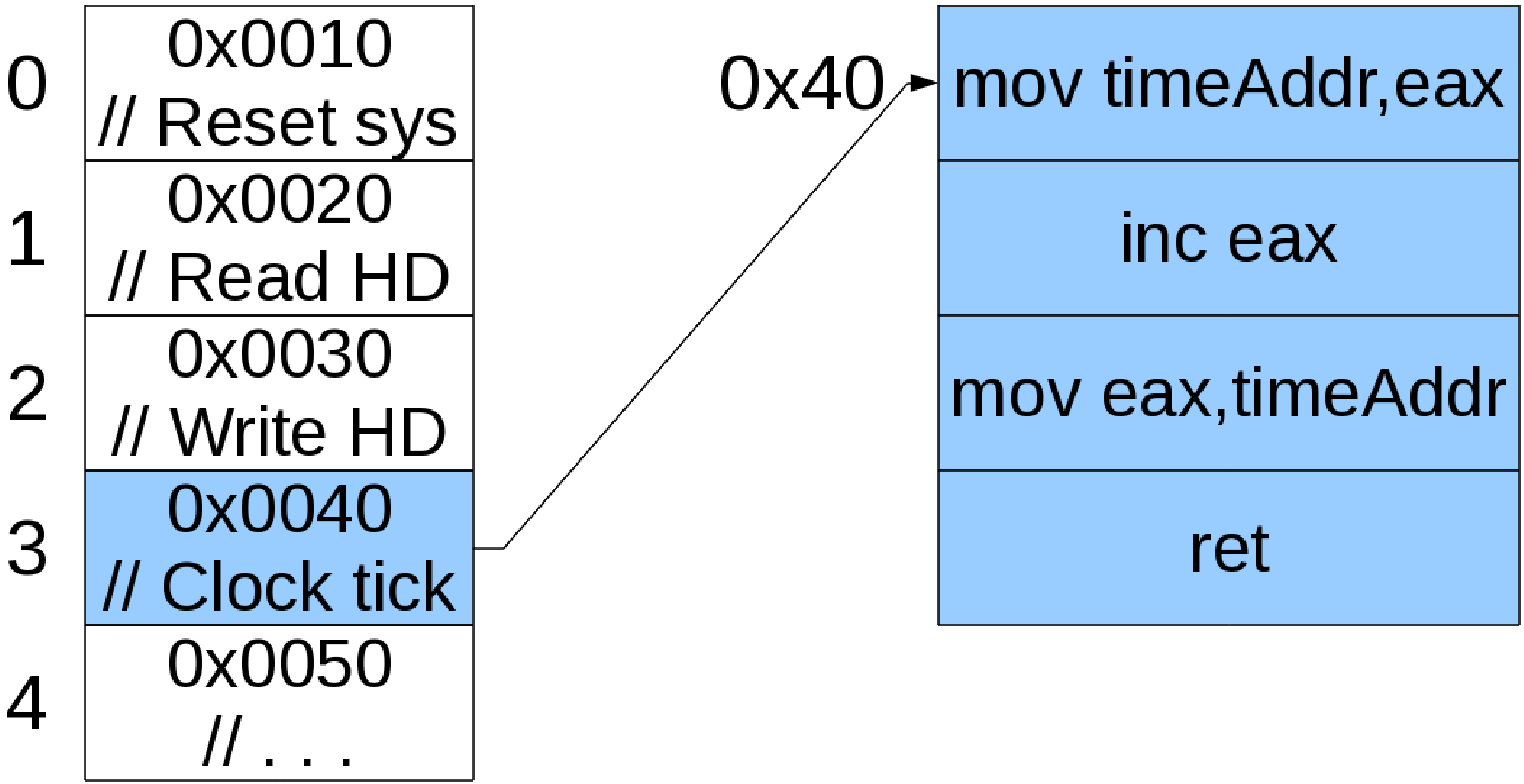
Remember the shell . . .

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

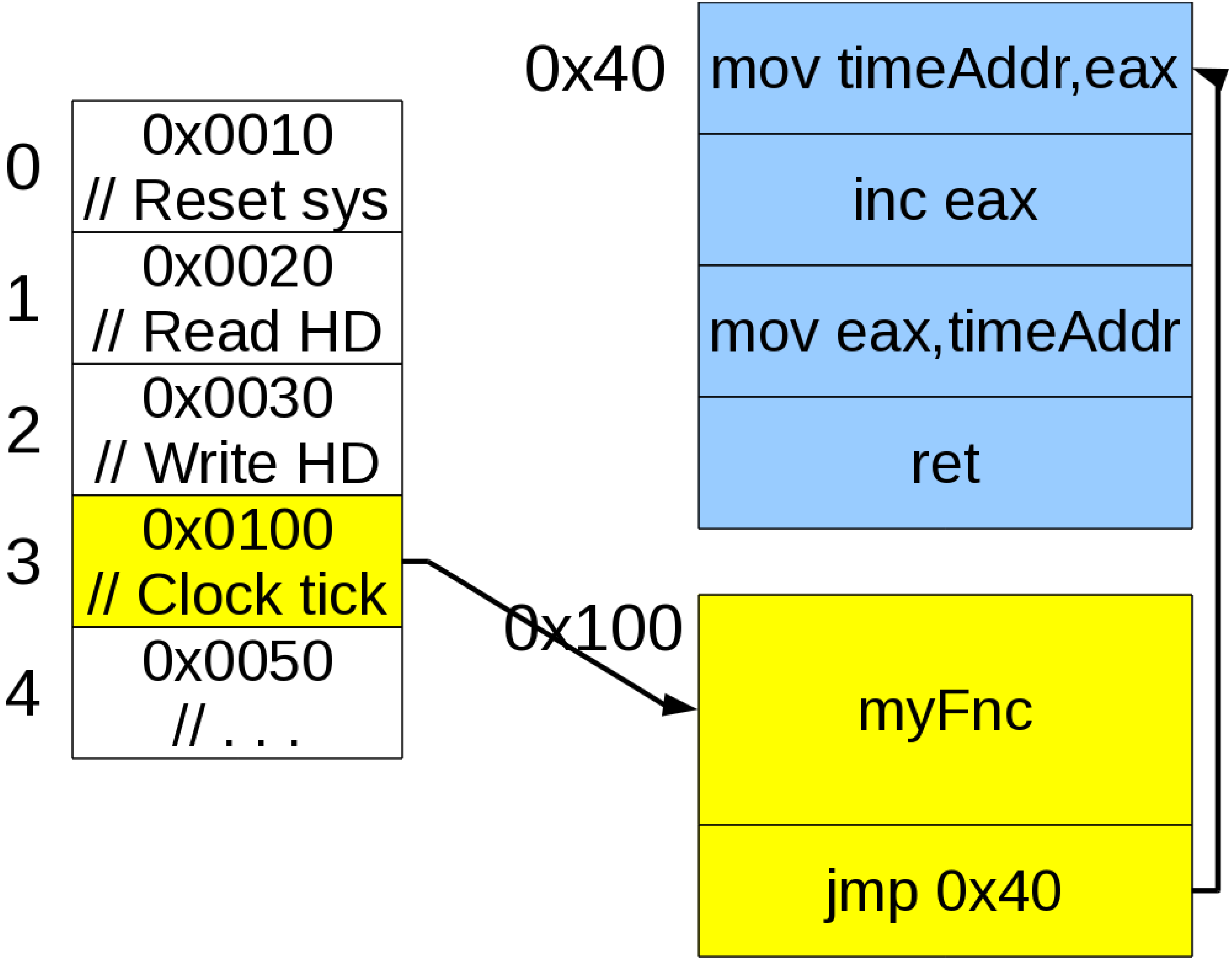


0	0x0010 // Reset sys
1	0x0020 // Read HD
2	0x0030 // Write HD
3	0x0040 // Clock tick
4	0x0050 // . . .

0x40

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

Remember how to do something periodically?

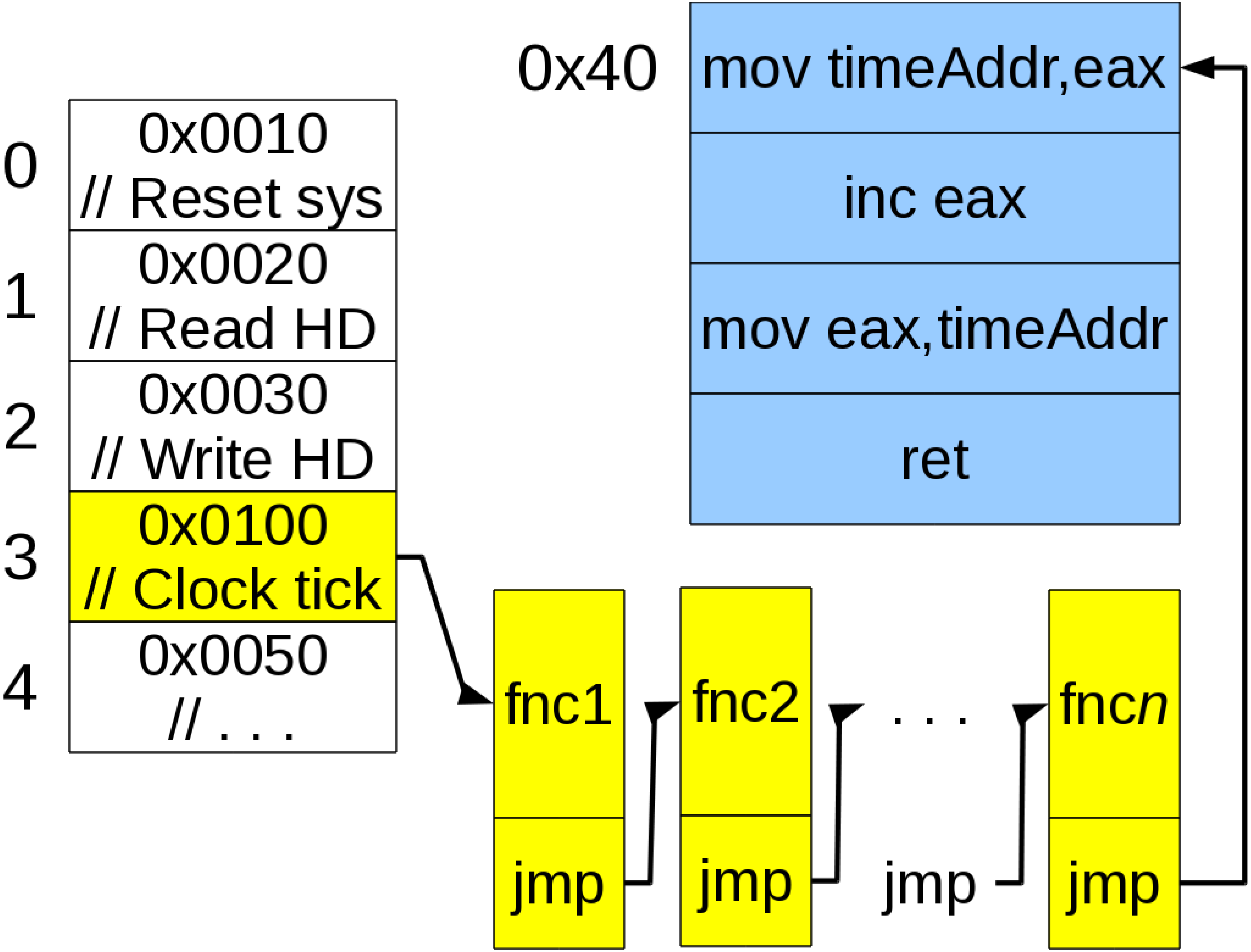


0	0x0010 // Reset sys
1	0x0020 // Read HD
2	0x0030 // Write HD
3	0x0100 // Clock tick
4	0x0050 // . . .

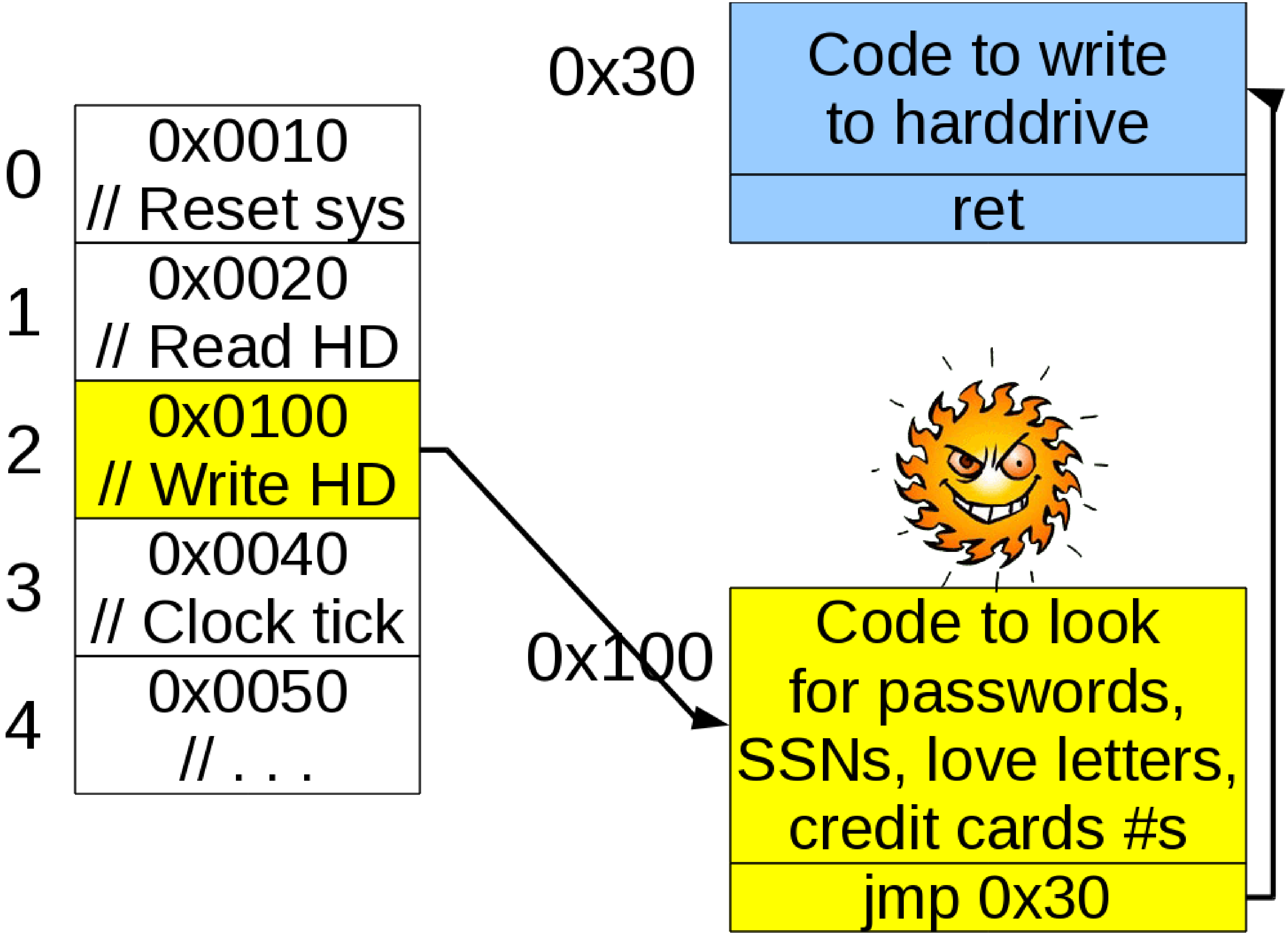
```
0x40   mov timeAddr,eax
       inc eax
       mov eax,timeAddr
       ret

0x100  myFnc
       jmp 0x40
```

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



Is this technique secure?



0	0x0010 // Reset sys
1	0x0020 // Read HD
2	0x0100 // Write HD
3	0x0040 // Clock tick
4	0x0050 // . . .

0x30

Code to write to harddrive

ret

0x100

Code to look for passwords, SSNs, love letters, credit cards #s

jmp 0x30

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)

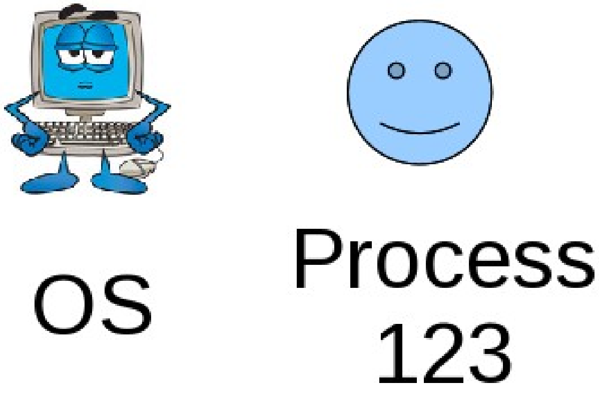
```
Signal  Value  Action   Comment
-------------------------------------
SIGSEGV   11    Core    Invalid memory reference
SIGPIPE   13    Term    Broken pipe: write to pipe with no
                        readers
SIGALRM   14    Term    Timer signal from alarm(2)
SIGTERM   15    Term    Termination 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)

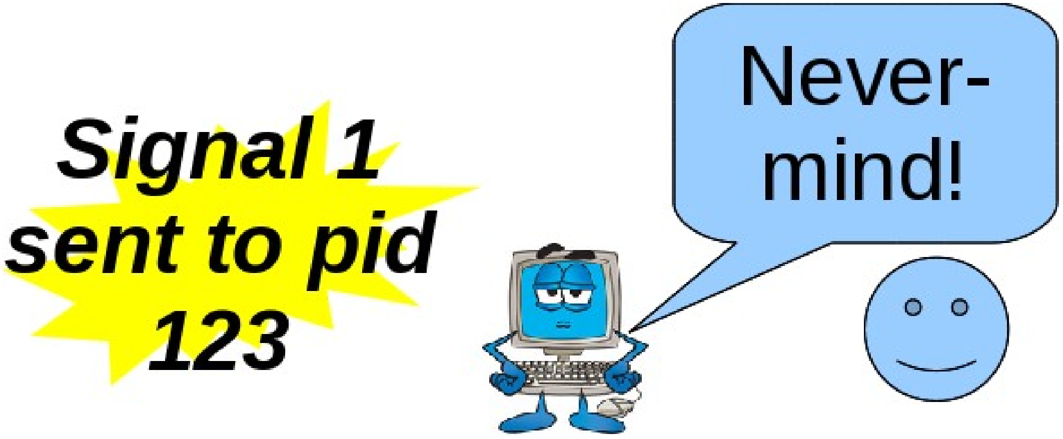
Each process has its own signal table:

0	term
1	ign
2	fnc()

Signal table for 123



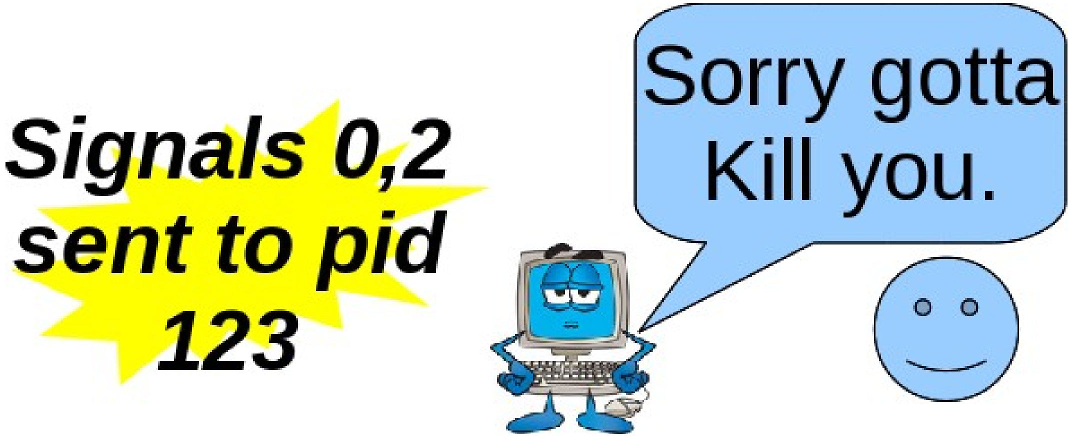
Some signals may be ignored or handled specially



How OS handles signals (2)

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

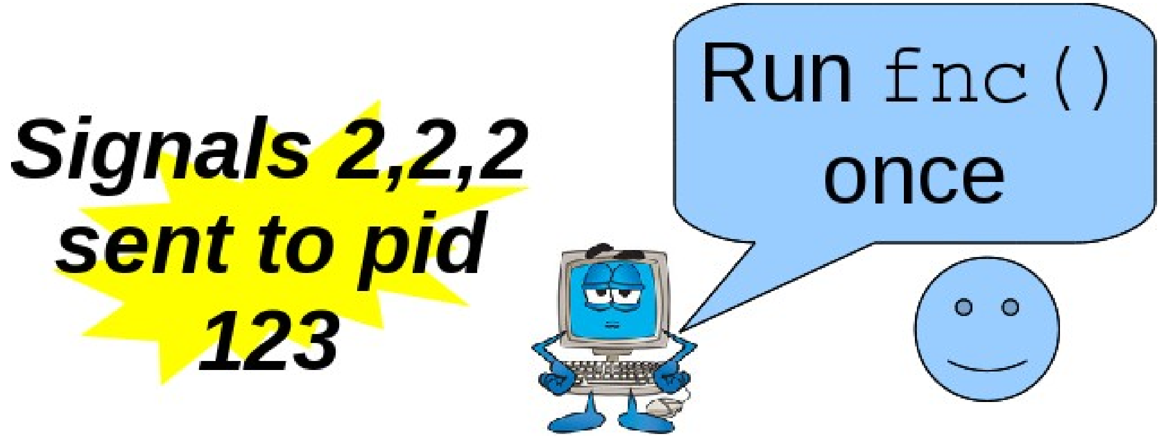
Answer: The lowest numbered signal takes precedence



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.



Signals 2,2,2 sent to pid 123

Run `fnc()` once

A look at those actions

Signals so serious that can't be trapped, ignored:

- `SIGSTOP`: Stop that! (Restartable with `SIGCONT`)
- `SIGKILL`: **Die now! *Punk!***

Other serious signals include:

- `SIGILL`: Illegal instruction (how could this happen?)
- `SIGSEGV`: Illegal memory reference (how could this happen?)

Other signals are less serious

- `SIGCHLD`: A child has finished (Hey, wouldn't `wait()` or `waitpid()` ***always*** catch this for us?)
- `SIGINT`: Ctrl-C
- `SIGHUP`: Controlling terminal hung-up (disconnected)

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!

sigaction()
(1) Defining a simple action

#include <signal.h>

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

Where:

```
struct sigaction
{
  void (*sa_handler)(int); // Simple 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);
};
```

Doing it! (1)

```
#include <signal.h>

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

int main ()
{
  // Set up struct to specify the new action.
  struct sigaction act;

  memset(&act,'\0',sizeof(struct sigaction));
  sigemptyset(&act.sa_mask);
  act.sa_flags = 0;
  . . .
```

Doing it! (2)

```
  act.sa_handler = SIG_DFL;//Handle by default
  sigaction(SIGINT,&act,NULL);

  act.sa_handler = SIG_IGN;// Ignore SIGINT
  sigaction(SIGINT,&act,NULL);

  act.sa_handler = simpleHandler;
               // Handle with simpleHandler()
  sigaction(SIGINT,&act,NULL);


  . . .
}
```

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 waited for child process to finish.

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

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.)

Reap the child so it doesn't remain a *zombie* too long.

HINT: use:

```
action.sa_flags =
   SA_NOCLDSTOP// No SIGCHLD on stop(pause) or resume
  |SA_RESTART; // If interrupted in sys call then
               //  restart sys call after sig handler
```

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.

sigaction()
(2) Defining more complex actions

#include <signal.h>

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

Where:

```
struct sigaction
{
  void (*sa_handler)(int); // Simple 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);
};
```

Look at all the goodies in `siginfo_t` (1)

```
siginfo_t
{
  int        si_signo;    // Signal number
  int        si_errno;    // An errno value
  int        si_code;     // Signal code
  int        si_trapno;   // Trap number that caused
                          // hardware-generated signal
                          // (unused on most archs.)
  pid_t      si_pid;      // Sending process ID
  uid_t      si_uid;      // User ID of sending proc
  int        si_status;   // Exit value or signal
  clock_t    si_utime;    // User time consumed
  clock_t    si_stime;    // System time consumed
  sigval_t si_value;      // Signal value
  int        si_int;      // POSIX.1b signal
  void      *si_ptr;      // POSIX.1b signal
  . . .
```

Look at all the goodies in siginfo_t (2)

```
  . . .
  int        si_overrun;   // Timer overrun count;
                           // POSIX.1b timers
  int        si_timerid;   // Timer ID;
                           // POSIX.1b timers
  void      *si_addr;      // Mem loc that caused fault
  long       si_band;      // Band event (was int in
                           // glibc 2.3.2 and earlier)
  int        si_fd;        // File descriptor
  short      si_addr_lsb;  // Least sign. bit of addr.
                           // (since kernel 2.6.32)
}
```

sigaction() (Example 2a)

```
/*  More advanced sig handlers take   */
/* tell which process sent the signal */

#include <signal.h>
#include <string.h>
#include <stdio.h>

void    signal_handler
(int sig, siginfo_t* infoPtr, void* dataPtr)
{
  printf("signal:[%d], pid:[%d], uid:[%d]\n",
         sig,
         infoPtr->si_pid,
         infoPtr->si_uid );
  //  dataPtr is not used so much
}
```

sigaction() (Example 2b)

```
int main (int argc, char *argv[])
{
  struct sigaction sa;
  memset(&sa,'\0',sizeof(struct sigaction));
  sigemptyset(&sa.sa_mask );

  sa.sa_flags= SA_SIGINFO //Install sa_sigaction
                          // (as opposed to
                          //  sa_handler)
             | SA_RESTART;//If interrupted in
                          // sys call then
                          // restart sys call
                          // after signal handler

  sa.sa_sigaction = signal_handler;
  sigaction(SIGINT, &sa, NULL);
```

sigaction() (Example 2c)

```
  int i;
  for  (i = 0;  i < 60;  i++)
  {
    printf("%2d of 60\n",i);
    sleep (1);
  }
  return(EXIT_SUCCESS);
}
```

Our turn!

Write a program that forks a process.
The child process should print its PID.
The child should also install a SIGINT handler.
When the child receives SIGINT it is being tickled.

The parent enters a loop:
(1) wait for user to press enter
(2) send SIGINT to child

Child enters sleep(1) loop.
When the parent tickles the child says
“***Hee hee, that tickles!***”
When any other process tickles the child says
“***Keep your hands to yourself, MISTER!***”

sigaction()
(3) Seeing what already is being done

#include <signal.h>

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

Where:

```
struct sigaction
{
  void (*sa_handler)(int), // Simple 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() Continued!

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()`
But wait! *There's more!*

Way more detail!

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

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!*