3150 - Operating Systems

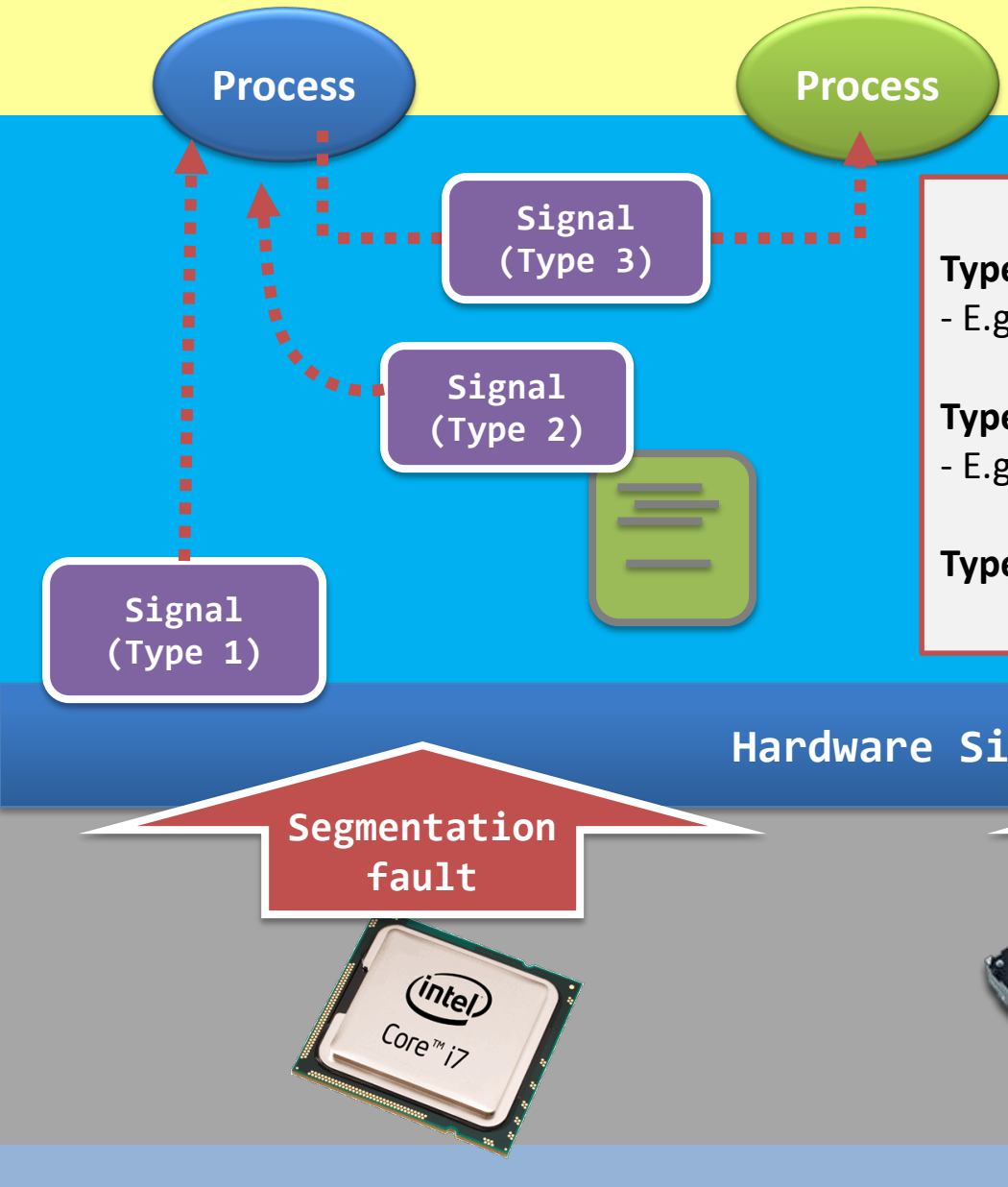
Dr. WONG Tsz Yeung

Chapter 2, part 3 – POSIX Signals

- Let's learn how to interrupt a process (and have fun)

Outline

POSIX – **P**ortable **O**perating **S**ystem **I**nterface for uni**X**



Type 1. From hardware signal to POSIX signal.
- E.g., segmentation fault, floating-point exception.

Type 2. Generated directly from kernel.
- E.g., from **`exit()`** to the creation of **`SIGCHLD`**.

Type 3. Generated from one process to another.

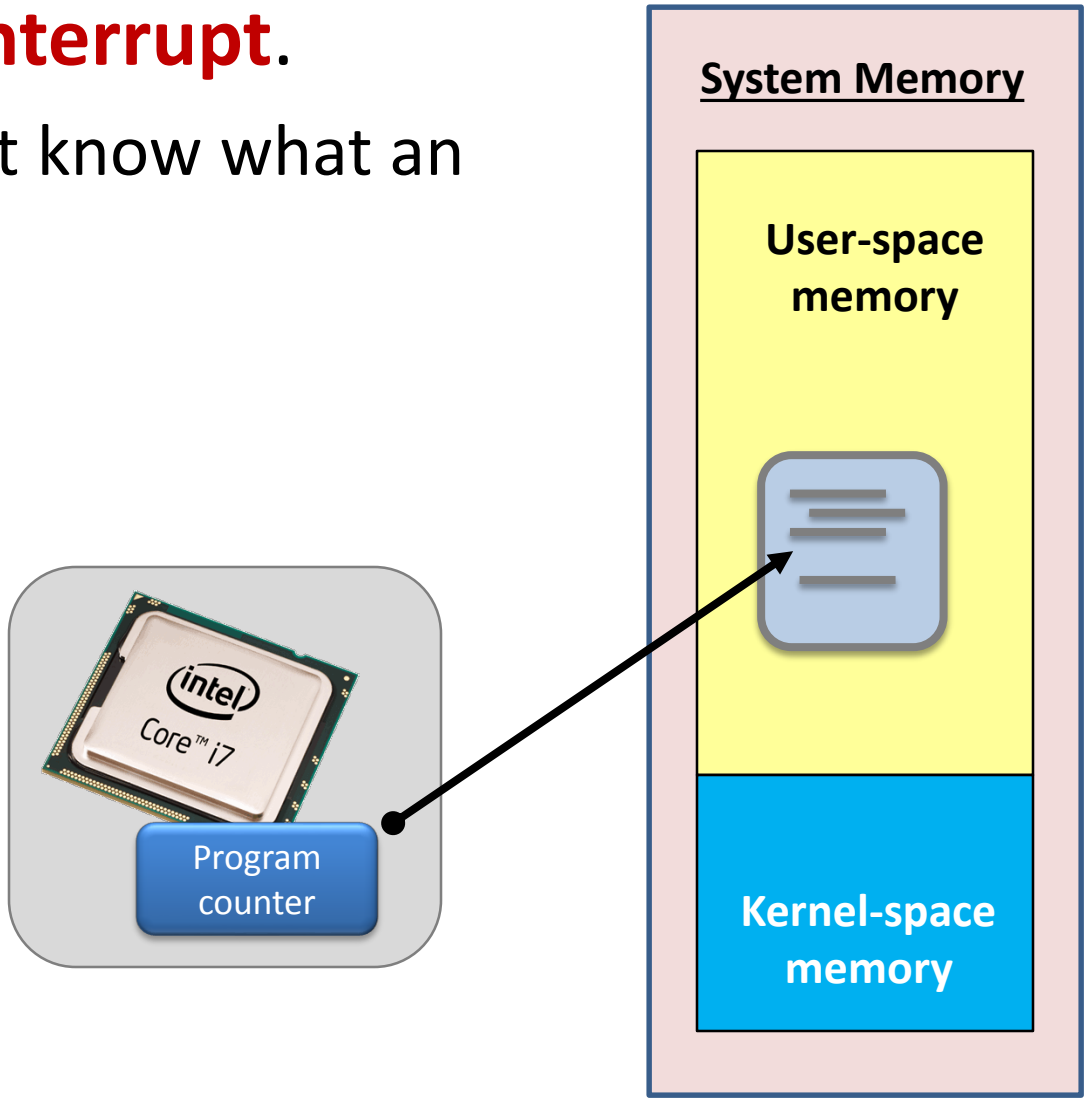
Signals

-preface: hardware signals;

Hardware signal

- It is the **hardware interrupt**.
  - CS students may not know what an interrupt is…

Originally, the CPU is working on a program code.



Hardware signal

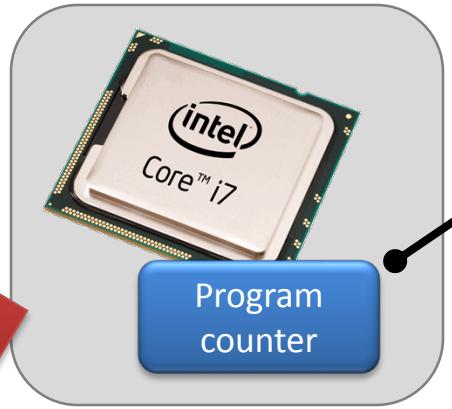
- It is the **hardware interrupt**.
  - CS students may not know what an interrupt is…

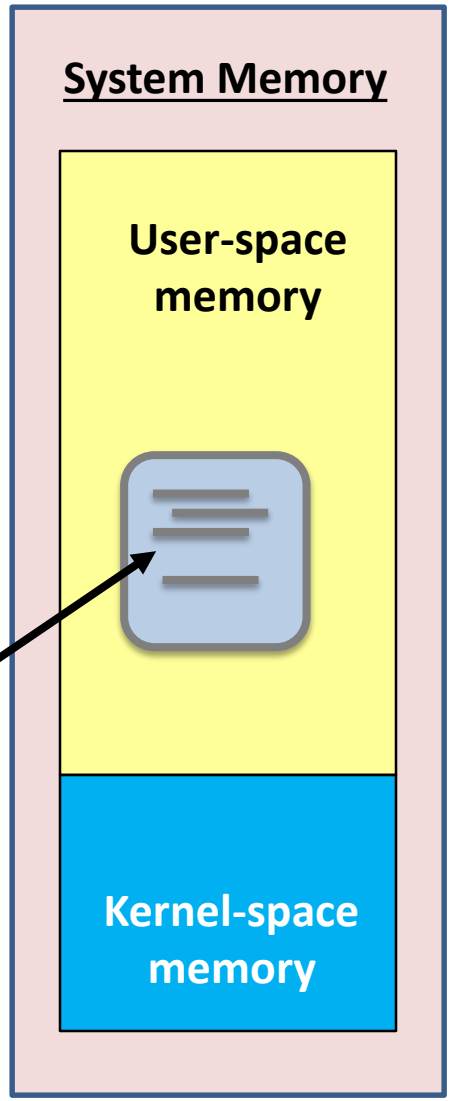
Suddenly, someone types things on to the keyboard.

This generates a hardware-interrupt that *interrupts* the execution of the CPU.

noun

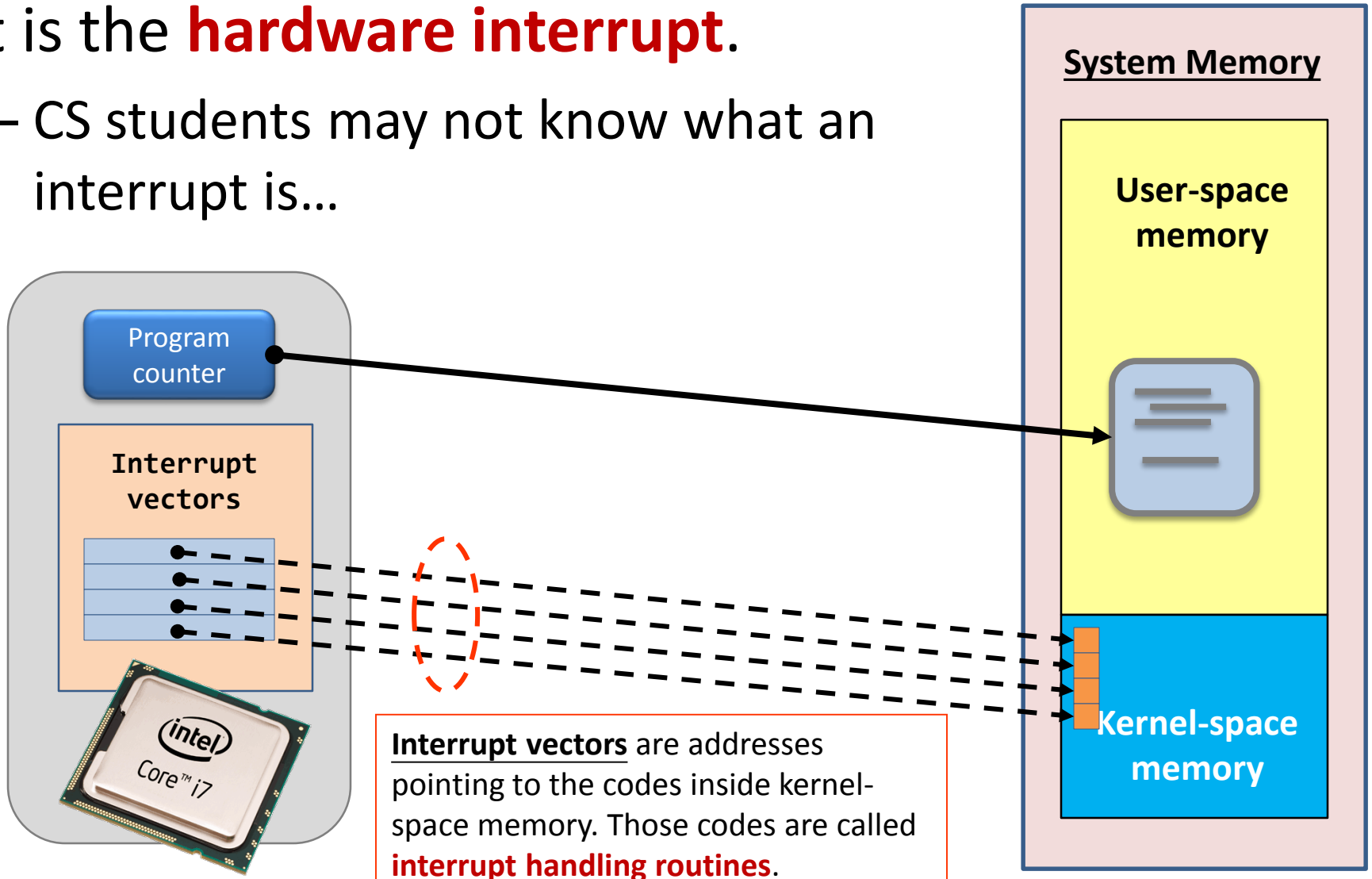
verb





Hardware signal

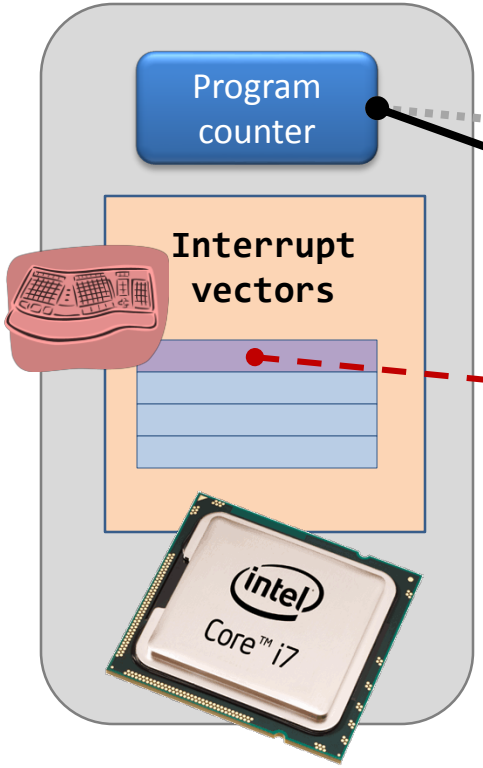
- It is the **hardware interrupt**.
  - CS students may not know what an interrupt is…

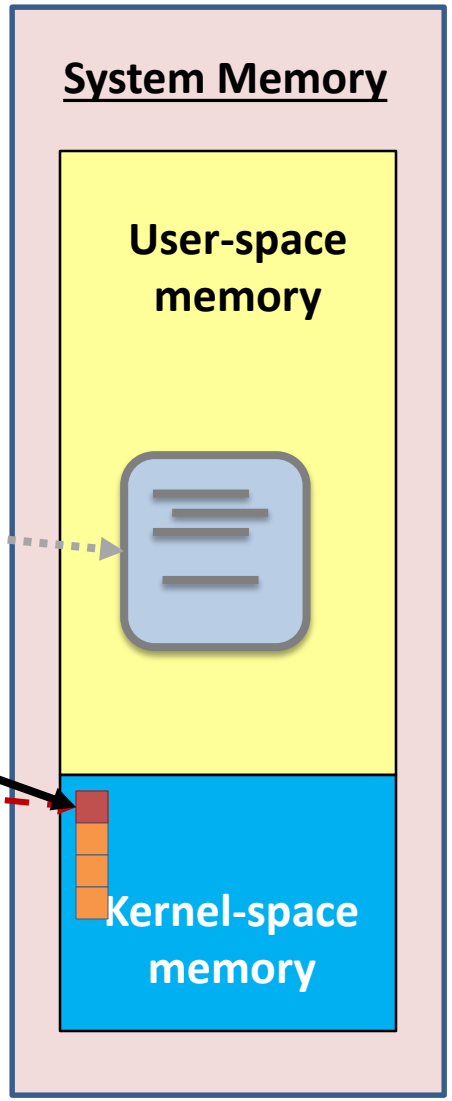


Interrupt vectors are addresses pointing to the codes inside kernel-space memory. Those codes are called **interrupt handling routines**.

Hardware signal

- It is the **hardware interrupt**.
  - CS students may not know what an interrupt is…



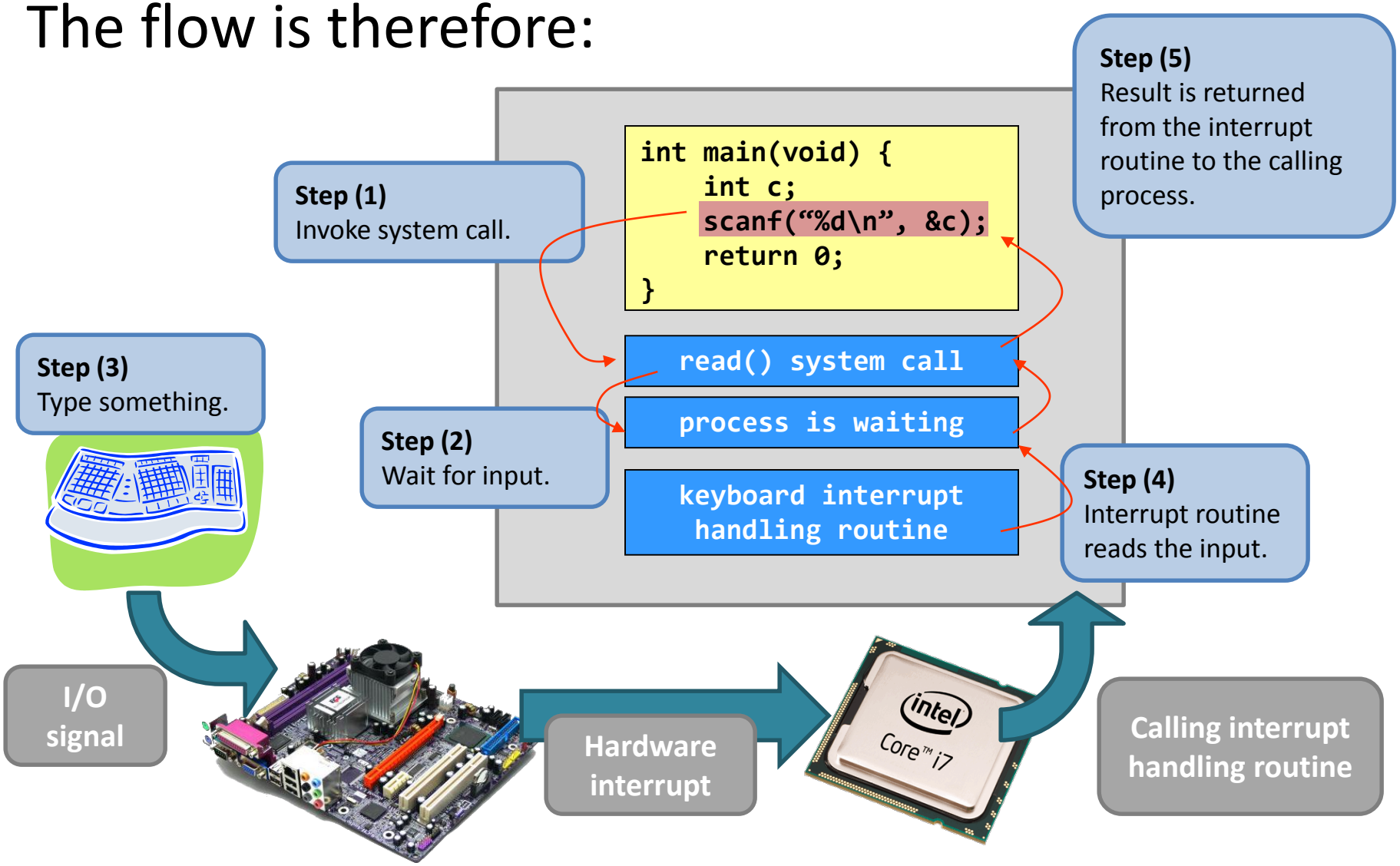


Say, the keyboard is associated to a particular interrupt vector.

Then, that interrupt handling routine will be invoked.

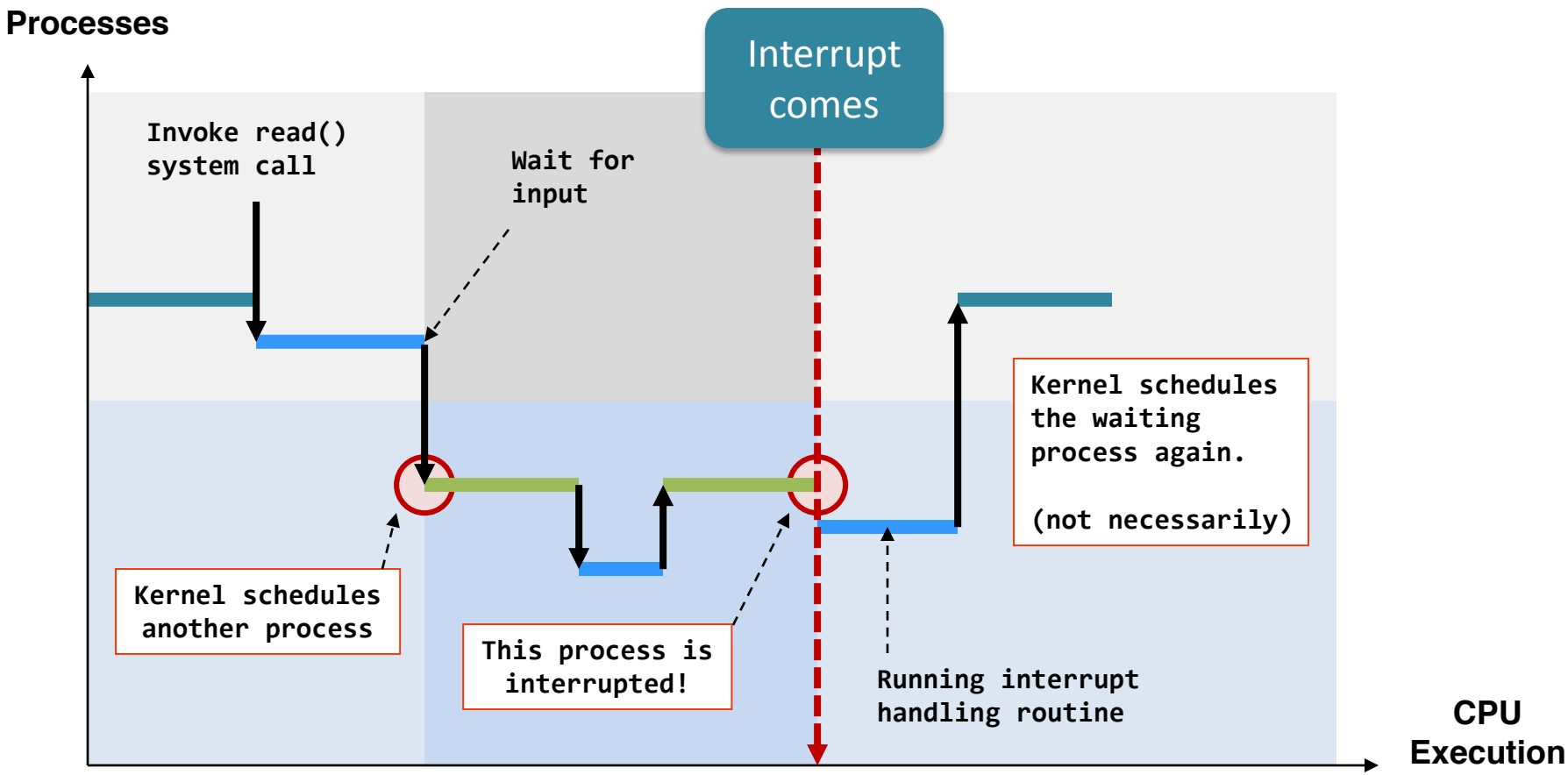
Hardware signal

- The flow is therefore:



Hardware signal

- Don't think that the kernel will be waiting for the keyboard input **forever**!



Signals

-preface: hardware signals;

-POSIX signals basics;

What is a POSIX signal?

Generated from one process to anther	Generated from CPU to kernel, or CPU to processes
From terminals: E.g., "**`Ctrl + C`**", "**`Ctrl + Z`**" and "**`Ctrl + \`**"	**<u>Example 1: Segmentation fault.</u>** The signal is labeled **`SIGSEGV`**, which comes from CPU to kernel then to processes.
Using programs: E.g., "**`kill`**", "**`top`**", etc.	**<u>Example 2: Floating point exception.</u>** The signal is labeled **`SIGFPE`**, which is coming from CPU when a user-level code makes things wrong: division by zero.
Using the "**`kill()`**" system call.	**<u>Example 3: Child process termination.</u>** The signal is labeled SIGCHLD, which is coming from the kernel; CPU is not involved.

Send signals using kill program

- The kill program is to **send signals** to target processes.

The **default signal** sent by the kill program is SIGTERM.

+

For each process, the **default signal handler** for SIGTERM is termination.

=

Eventually, the target process *looks like being* terminated by kill.

```
$ ps -u tywong
 PID   TTY
1234   ... ./infinite_loop
......
$ kill 1234
$ _
```

Terminal A

```
$ ./infinite_loop
1 sec
2 sec
3 sec
Terminated
$ _
```

Terminal B

Some default things…

Signal	Description	Default signal handler
SIGINT	Its name is the **interrupt signal**. Can be generated by "**`Ctrl + C`**".	Target process termination.
SIGTERM	Its name is the **termination signal**. The default signal sent by the "**`kill`**" program.	Target process termination.
SIGTSTP	Its name is the **terminal stop signal**. Can be generated by "**`Ctrl + Z`**".	Target process suspension.
SIGCONT	Its name is the **continuation signal**. Will discuss this later.	Target process resumes execution if it is previously suspended.
SIGCHLD	(No special name). It is sent to the parent process to a terminated child.	Ignore by default. ***unless use wait()***
SIGKILL	Its name is the **kill signal**. If sent, the process **MUST DIE**.	Target process termination. (Plus, no one could stop the termination.)

What are **SIGTSTP** & **SIGCONT**?

```
$ ps -u tywong
 PID   TTY
1234   ... ./infinite_loop
......
$ kill -TSTP 1234
$ kill -CONT 1234
```

SIGTSTP

SIGCONT

```
$ ./infinite_loop
1 sec
2 sec
3 sec
[1] Stopped   ./infinite_loop
$ 4 sec
5 sec
6 sec
```

What happened to the shell?!

The "terminal stop signal" stops the process.

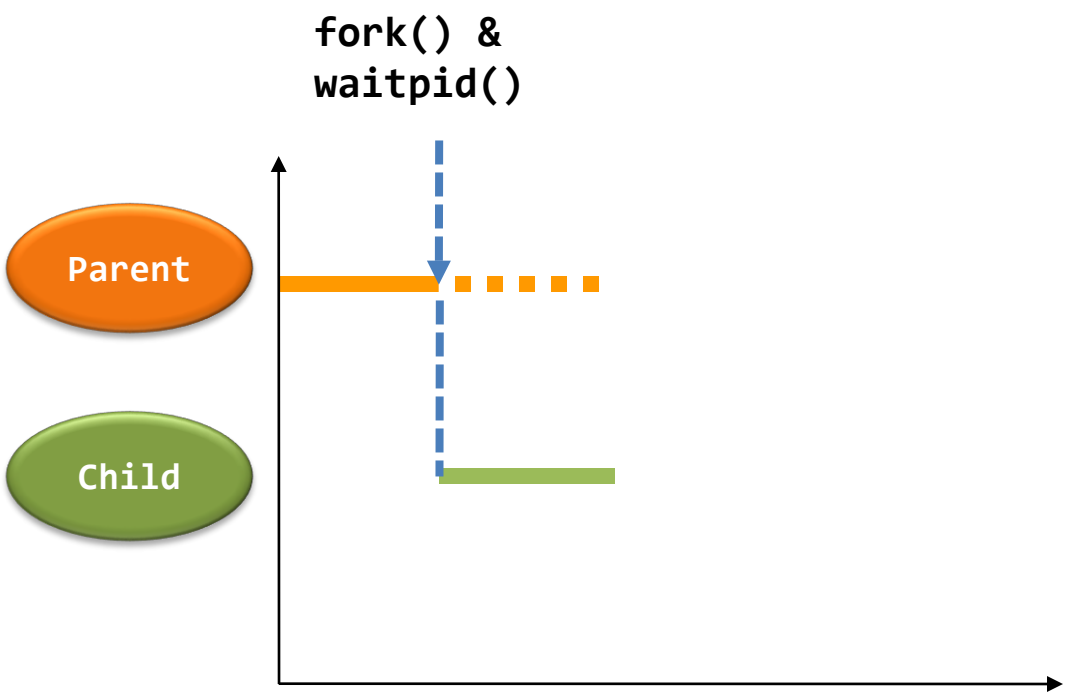
The "continuation signal" resumes the process. But, wait….

Terminal A

Terminal B

Foreground and background jobs

A characteristic of a foreground job is shell is:
<u>The shell is waiting for the job to change state.</u>

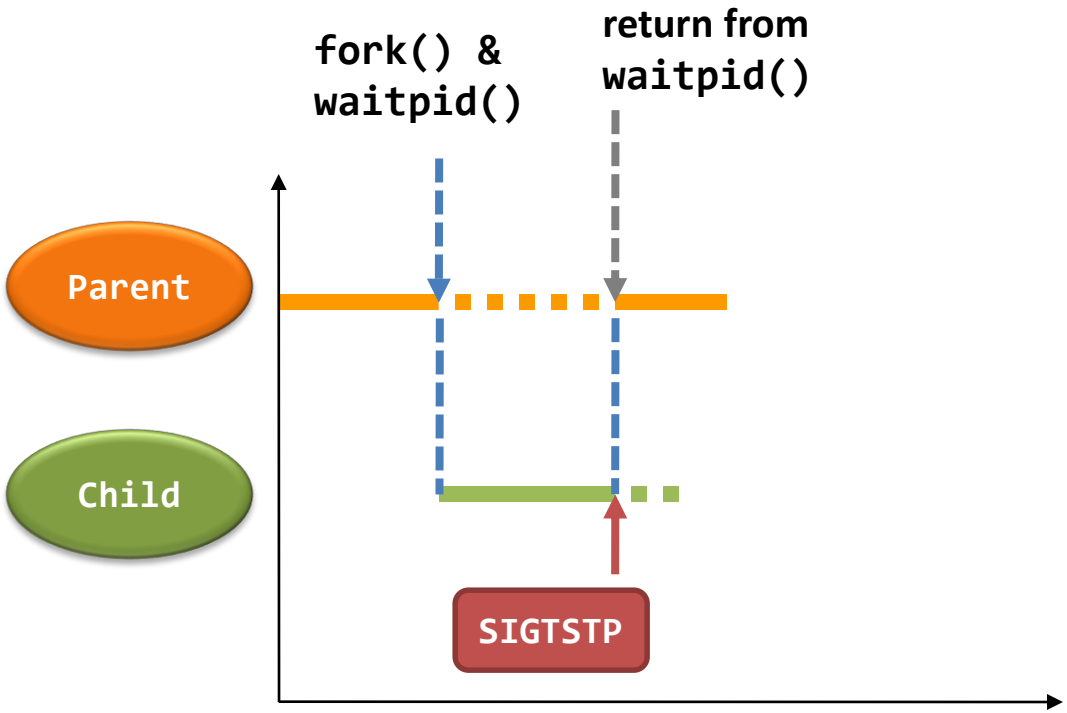



```
$ ./infinite_loop
1 sec
2 sec
3 sec
```

We name this a **<u>foreground job</u>**.

Foreground and background jobs

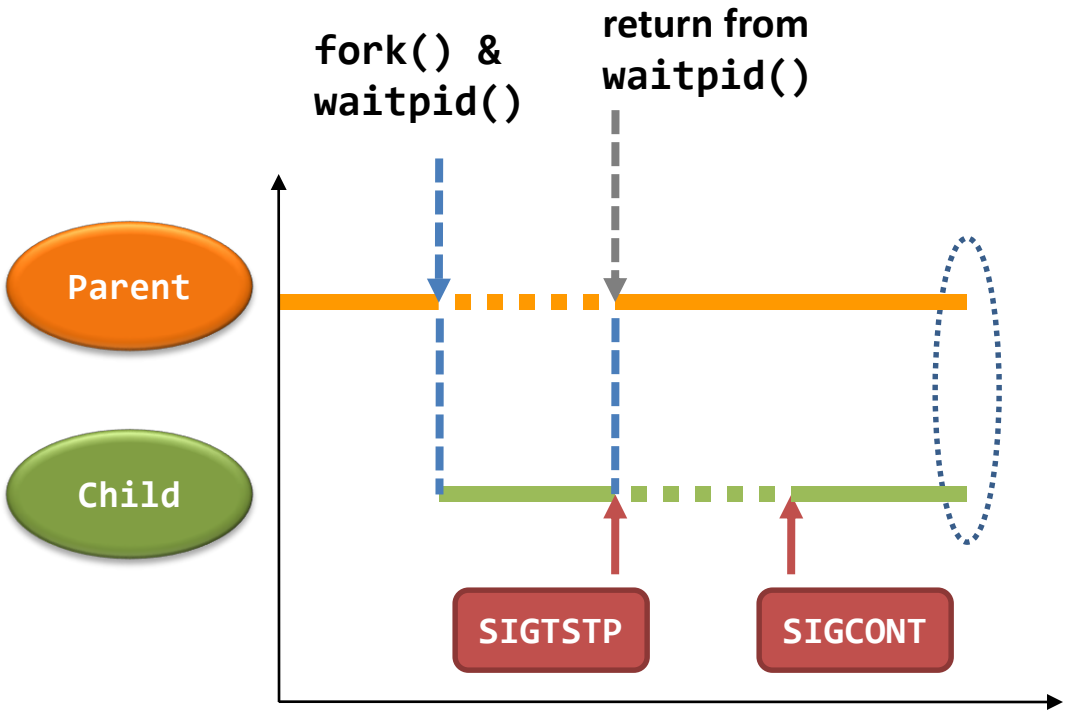
Now, the shell wakes up. By the way, "*[1] Stopped* …" is actually printed out by the shell!




```
$ ./infinite_loop
1 sec
2 sec
3 sec
[1] Stopped ...
$ _
```

Foreground and background jobs

SIGCONT is fired from another shell. Since the parent shell doesn't know about this fact, they both just run in parallel.



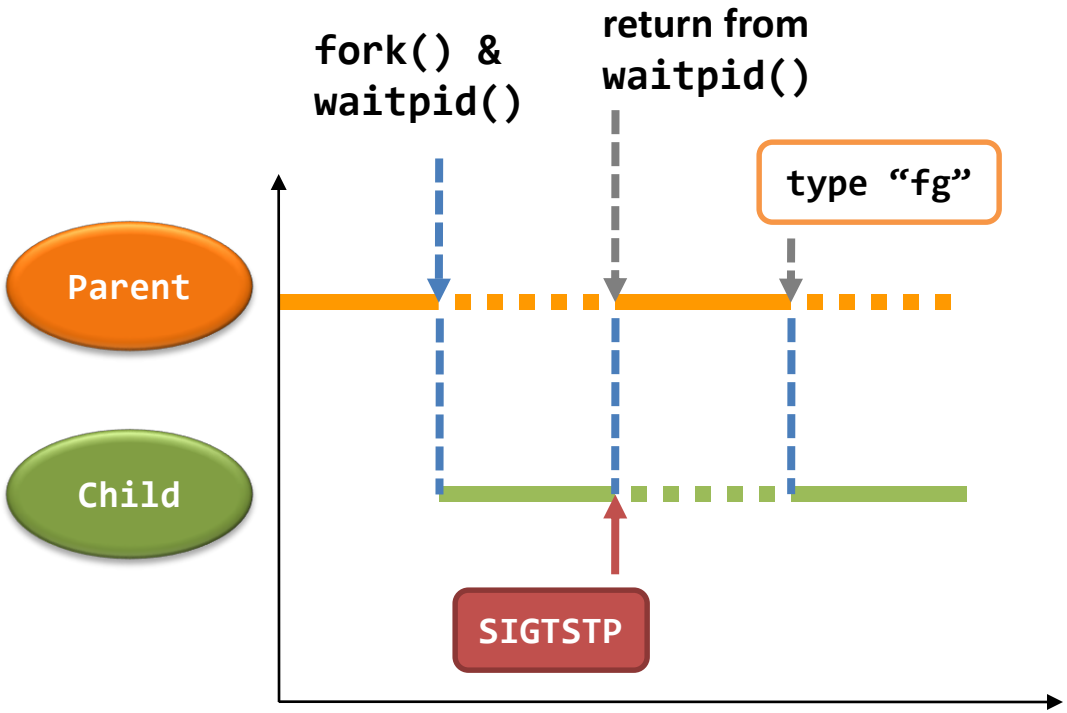

```
$ ./infinite_loop
1 sec
2 sec
3 sec
[1] Stopped ...
$ 4 sec
5 sec
6 sec
```

Now, the job becomes a **background job**.

Foreground and background jobs

DISCUSSION

The question is simple: **how to implement the command "fg"?**




```
$ ./infinite_loop
1 sec
2 sec
3 sec
[1] Stopped ...
$ fg
4 sec
5 sec
6 sec
```

Now, the job becomes a **foreground job**.

Sending signals in a process

- Remember, **`kill()`** is <u>not intended to kill anybody</u>, but to send signals.

```
int kill(pid_t pid, int sig )
```

PID of the target process

The desired signal. For the complete list, please read the manpages:

On Linux:
man 7 signal
On Unix:
man -s 3HEAD signal

```
int main(void) {
    int i, sum;
    srand(time(NULL));
    while(1) {
        sum = 0;
        for(i = 0; i < 3; i++)
            sum += (rand() % 6) + 1;
        if(sum == 18)
            kill(getpid(), SIGTERM);
    }
    return 0;
}
```

An alterative: **`raise(SIGTERM);`**

```
[examples@3150] cat suicide.c
```

Signals

-Preface: hardware signals;

-POSIX signals basics;

-Handling POSIX signals;

Setting new signal handlers

- What is the meaning?
  - The process **<u>no longer executes the default handler</u>**…
  - The signal handlers can be changed to an **<u>user-level function</u>**!

```
signal( int signum, void (* function)(int) )
```

The signal number, e.g., **SIGINT** or **SIGTERM**.

<u>A function pointer to the new handler.</u> This means you should provide the name of a function, with:
- return type of "**void**" and
- one argument with type "**int**"

Two special values:
- **SIG_DFL**: set to the default handler.
- **SIG_IGN**: ignore this signal completely.

Setting new signal handlers

```
1   void sig_handler(int sig) {
2       if(sig == SIGINT)
3           printf("\nCtrl + C\n");
4   }
5
6   int main(void) {
7       signal(SIGINT, sig_handler);
8       printf("Press enter\n");
9       getchar();
10      printf("End of program\n");
11  }
```

Line 7 registers the signal handler when **SIGINT** is received.

Lines 1-4 together define the signal handler.

```
[examples@3150] cat handle_int.c
```

Setting new signal handlers

```
void sig_handler(int sig) {
    if(sig == SIGINT)
        printf("\nCtrl + C\n");
}

int main(void) {
    signal(SIGINT, sig_handler);
    printf("Press enter\n");
    getchar();
    printf("End of program\n");
}
```

```
$ ./handle_int
Press enter
^C
Ctrl + C
```

[examples@3150] cat handle_int.c

Setting new signal handlers

EXTRA & IMPORTANT

- An important point to note:
  - **Apparently**, when a signal handler returns, **the process goes back to where it was executing**.
  - But…

```
void sig_handler(int sig) {
    printf("\nSignal received.\n");
}

int main(void) {
    signal(SIGINT, sig_handler);
    printf("Sleep for 24 hours\n");
    sleep(24 * 60 * 60);
    printf("Wake up and die.\n");
}
```

```
$ ./break_sleep
Sleep for 24 hours
^C
Signal received.
Wake up and die
$_
```

```
[examples@3150] cat break_sleep.c
```

Setting new signal handlers

EXTRA & IMPORTANT

- An important point to note:
  - **Apparently**, when a signal handler returns, **the process goes back to where it was executing**.
  - But…this only happens when **the involved system/library call can be restarted automatically**.

Can be restarted	Cannot be restarted

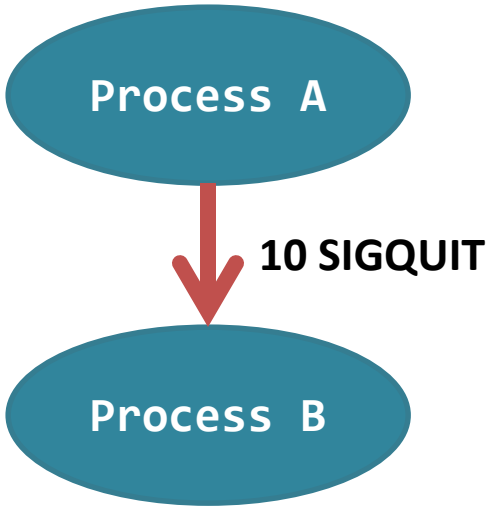
[file related]
`open(), read(), write();`
 	
[process related]
`wait(), waitpid();`
	`sleep(); pause();`
	
	With dozens of calls that you may not meet before…

Counting Signal Received?

EXTRA & IMPORTANT

```c
void handler(int sig) {
    static int count = 0;
    printf("count = %d\n", ++count);
}

int main(void) {
    int i;
    if( fork() == 0 ) {
        printf("Press Enter...\n");
        while(getchar() != '\n');
        for(i = 0; i < 10; i++)
            kill(getppid(), SIGQUIT);
    }
    else {
        signal(SIGQUIT, handler);
        sleep(1000);
        wait(NULL);
    }
  return 0;
}
```




```
$ ./many_signal
Press Enter...
[ENTER]
count = 1
$_
```

```
[examples@3150] cat many_signal.c
```

Counting Signal Received?

EXTRA & IMPORTANT

- Another important note:
  - Signals are not queued!
  - An array is used for indicating a signal has received or not.

recv

SIG	HUP	INT	QUIT	...	KILL	SEGV	...
BIT	0	0	1	...	0	0	

  - In the previous example, the bit (or mask) is always 1 no matter how many signals are sent.
  - The mask will be set to 0 when the signal is handled.
  - **Guess** what will happen with **1 million** SIGQUIT sent?

Handling exception signals…

CHALLENGE

- Write programs that try to handle the signals:
  - SIGSEGV; SEGFPE;
    - Question: are the errors really go away?
  - SIGCHLD;
    - Question: Is there any zombie left in the system?

Misc. Topics

- -Waiting for signals;
- -Breaking out of loops;
- -Timers and periodic signals;

(1) - Waiting for a signal

- The **`pause()`** system call suspends the calling process until…
  - a signal which is **handled** by the process is received, or
  - a signal which **terminates** the process is received.

It suspends the execution of the program until a signal is caught…

Of course, **`pause()`** is designed not to be restarting after a signal handler.

```
1  void sig_handler(int sig) {
2  }
3
4  int main(void) {
5      signal(SIGINT, sig_handler);
6      pause();
7      printf("Ctrl+C received. Bye!\n");
8      return 0;
9  }
```

```
[examples@3150] cat pause.c
```

(2) – Breaking out of loop

```
int stop = 0;

void sig_handler(int sig) {
    stop = 1;
}

int main(void) {
    unsigned int i = 0;
    signal(SIGINT, sig_handler);

    while( !stop ) {
        sleep(1);
        printf("%d sec\n", ++i);
        fflush(stdout);
    }

    printf("Exit peacefully\n");
    return 0;
}
```

The while loop is an infinite one until the user presses "**Ctrl+C**". We usually call that feature: the **graceful termination**.

It allows the program to **exit normally** so that it has a chance to do things such as:

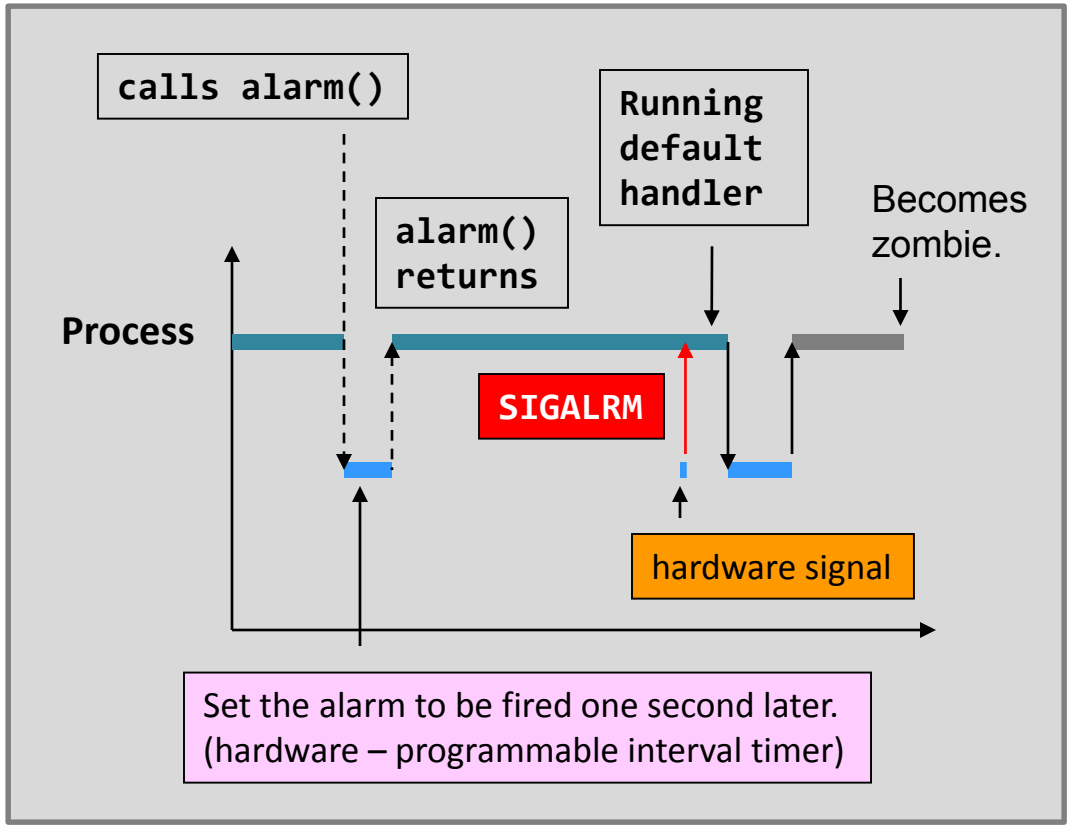
- closing network connections,
- committing database changes,
- etc.

rather than terminating the process forcefully.

```
[examples@3150] cat infinite_loop.c
```

(3) – Timer and alarm()

- **`alarm()`** is a system call that allows **asynchronous timing** for a process.




```
int main(void) {
    alarm(1);
    while(1);
    return 0;
}
```

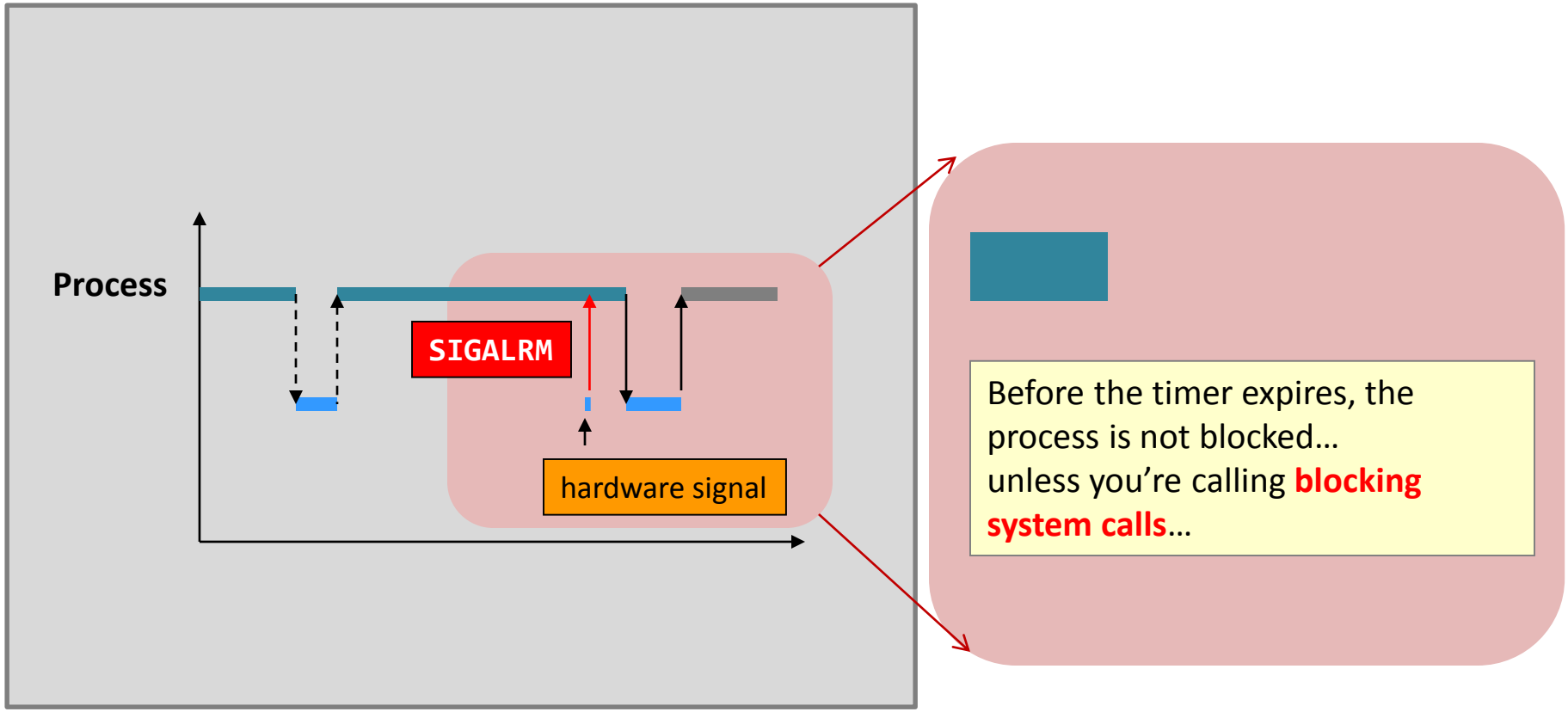
```
$ ./alarm
Alarm clock
$ _
```

1 sec later

```
[examples@3150] cat alarm.c
```

(3) – Timer and alarm()

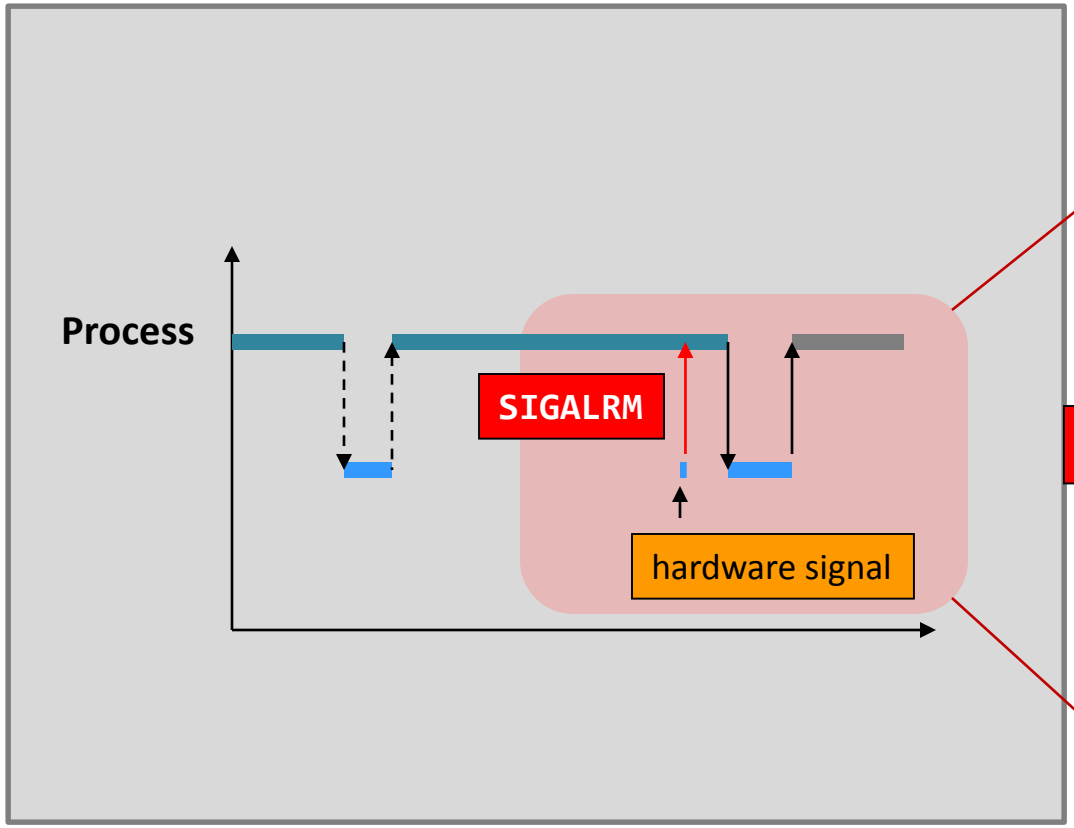
- **`alarm()`** is a system call that allows **asynchronous timing** for a process.




```
[examples@3150] cat alarm.c
```

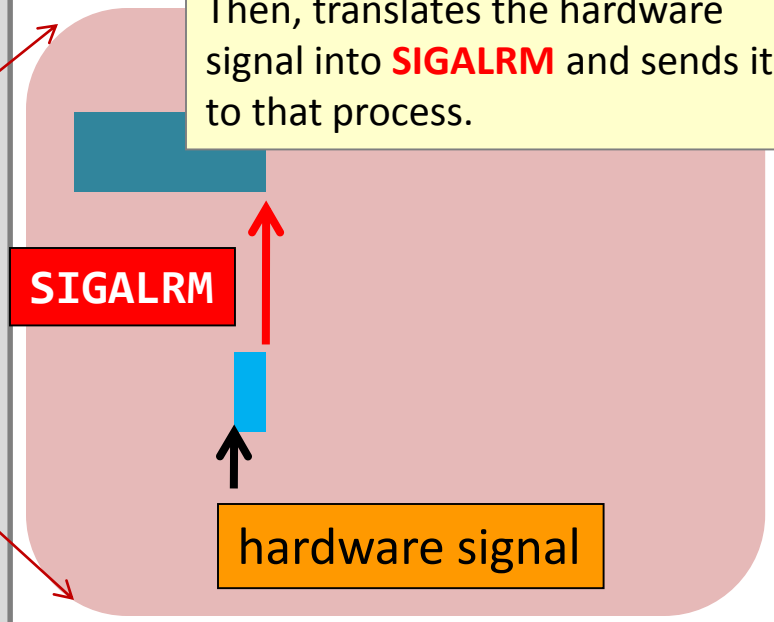
(3) – Timer and alarm()

- **`alarm()`** is a system call that allows **asynchronous timing** for a process.



Kernel looks for the process which has requested for the timer service.

Then, translates the hardware signal into **SIGALRM** and sends it to that process.

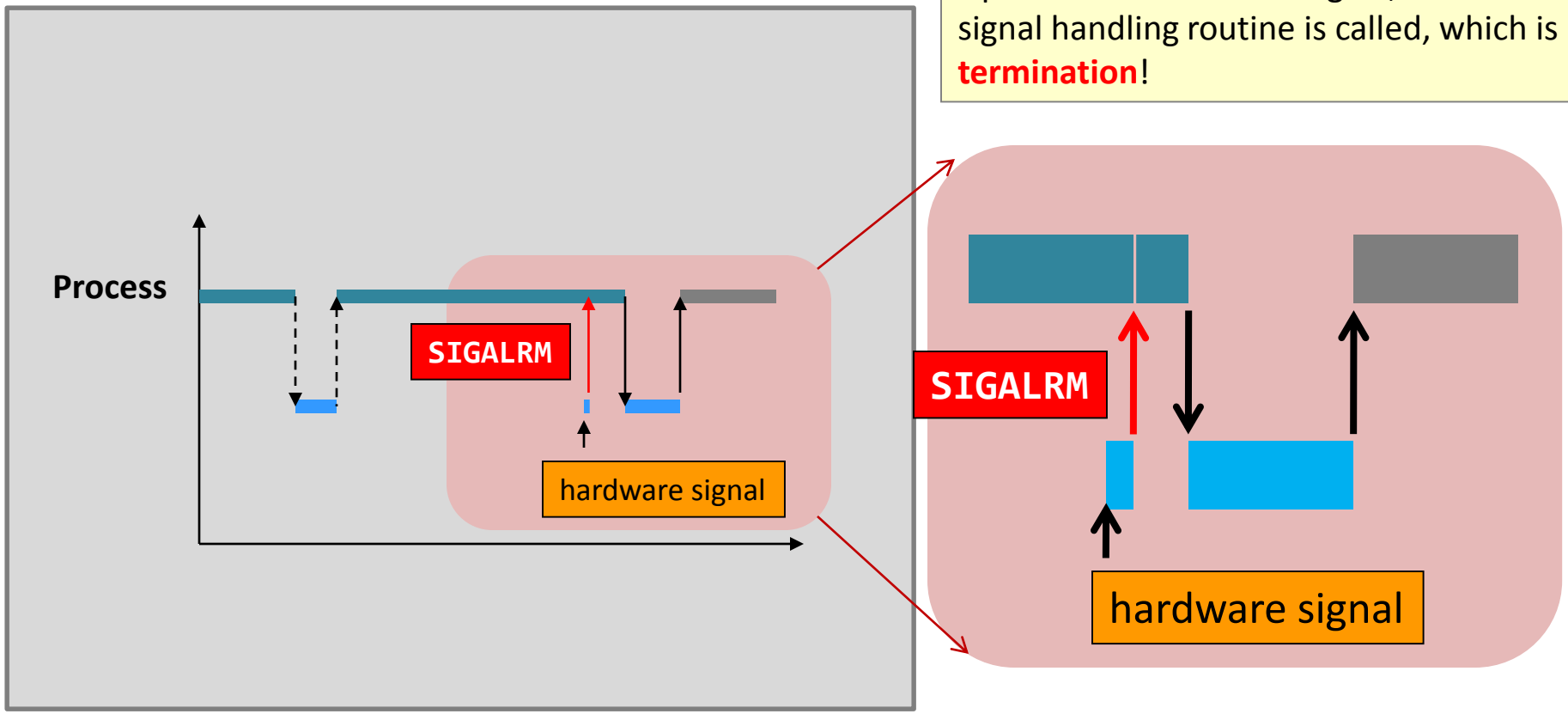



```
[examples@3150] cat alarm.c
```

(3) – Timer and alarm()

- **`alarm()`** is a system call that allows **asynchronous timing** for a process.

Upon the arrival of the signal, the default signal handling routine is called, which is **termination**!




```
[examples@3150] cat alarm.c
```

(3) – Timer and alarm()

- Of course, you can implement something more meaningful.

Guess: what will happen with this **`exit()`** call.

Listen! You've only 5 seconds to finish your typing!

This **cancels** the scheduled clock interrupt!

```
void sig_handler(int sig) {
    printf("\nTimeout! Goodbye!\n");
    exit(0);
}

int main(void) {
    char buf[1024];
    signal(SIGALRM, sig_handler);
    alarm(5);
    if(fgets(buf, 1024, stdin) == NULL) {
        printf("No input. Goodbye!\n");
        exit(0);
    }
    alarm(0);
    printf("Your input: %s", buf);
}
```

```
[examples@3150] cat alarm_fgets.c
```

(3) – Timer and alarm()

- Remember, "**`alarm()`**" only fires once!
  - What if I want **periodic signals**?

- How about calling **`alarm()`** again in a signal handler? Or…

- "**`setitimer()`**" (set interval timer) can help you.
  - Her sibling is "**`getitimer()`**".
  - Read the manpage by yourself.

Summary

- Signal is a kind of interrupts…
  - This is quite hard to master…
    - and the course instructor still needs to refer to man pages before explaining special behaviors of some signals.
  - It is the source of the many fancy (or evil) scenarios.

- You may need a reference book if you want to go deeper:
  - *Advanced Programming Environment in UNIX*;
  - "**`man 7 signal`**" is a vast resource.