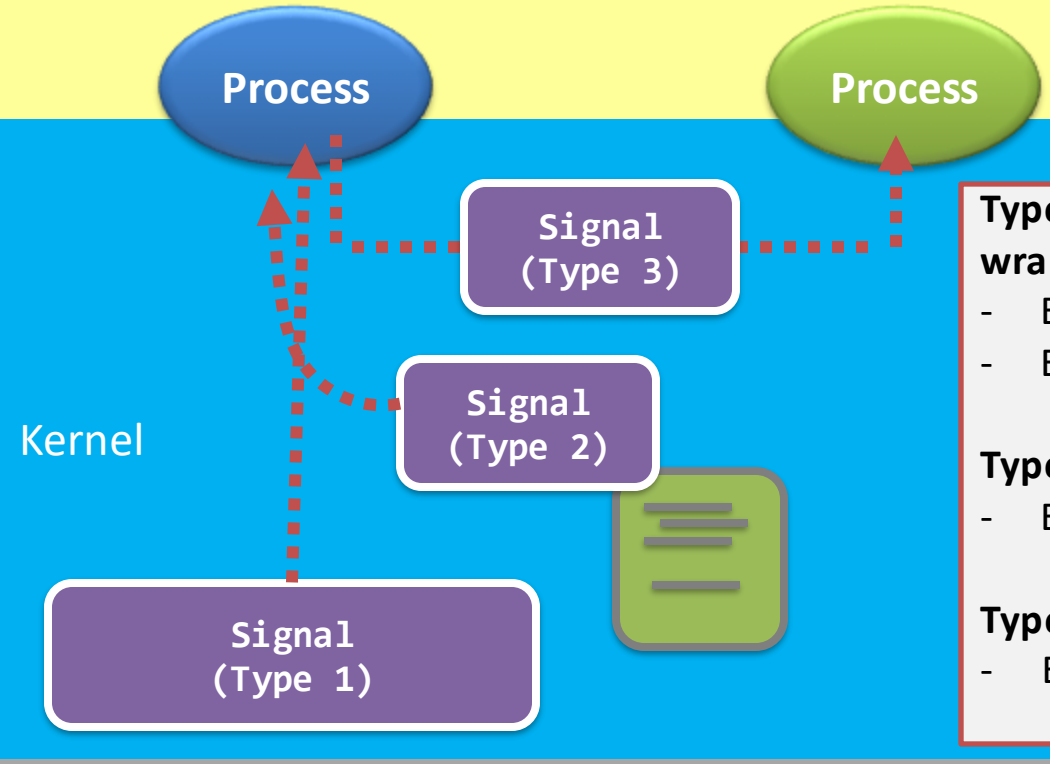
Operating Systems

Eric Lo

5 Signal

Signal – a limited form of communications



Type 1. From hardware interrupt / CPU exception, wrapped by the kernel as a POSIX signal

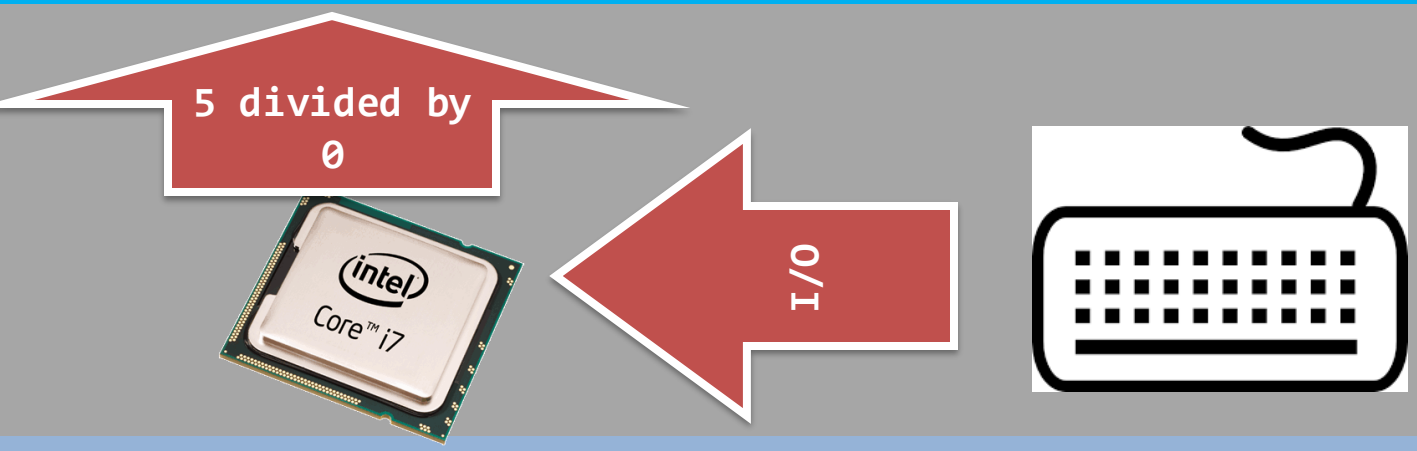
- E.g., Ctrl-C ➔ **`SIGINT`**
- E.g., segmentation fault ➔ **`SIGSEGV`**

Type 2. Generated directly from kernel.

- E.g., from **`exit()`** ➔ **`SIGCHLD`**

Type 3. Generated from one process to another.

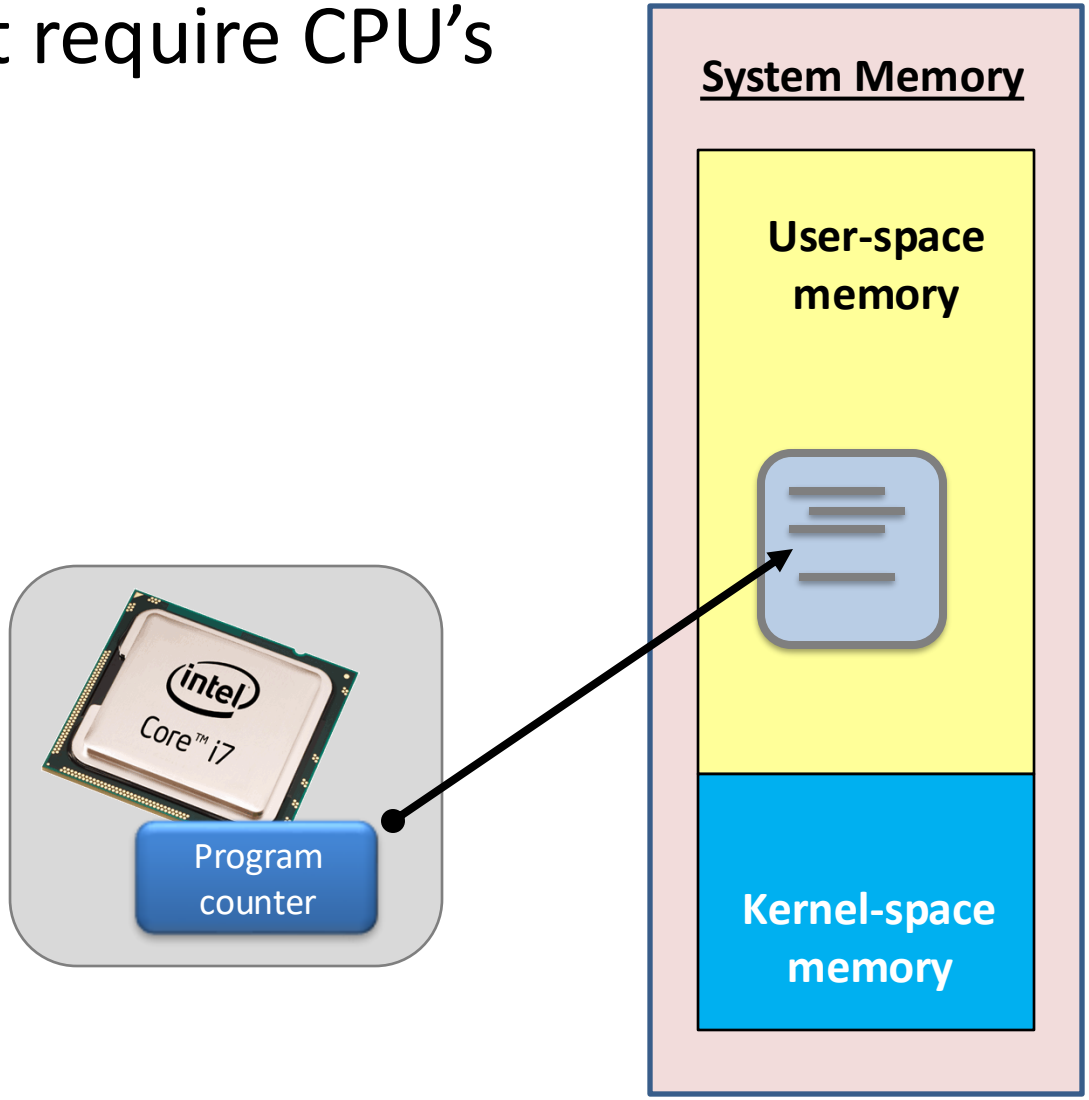
- E.g., "kill 1234" from another process 885



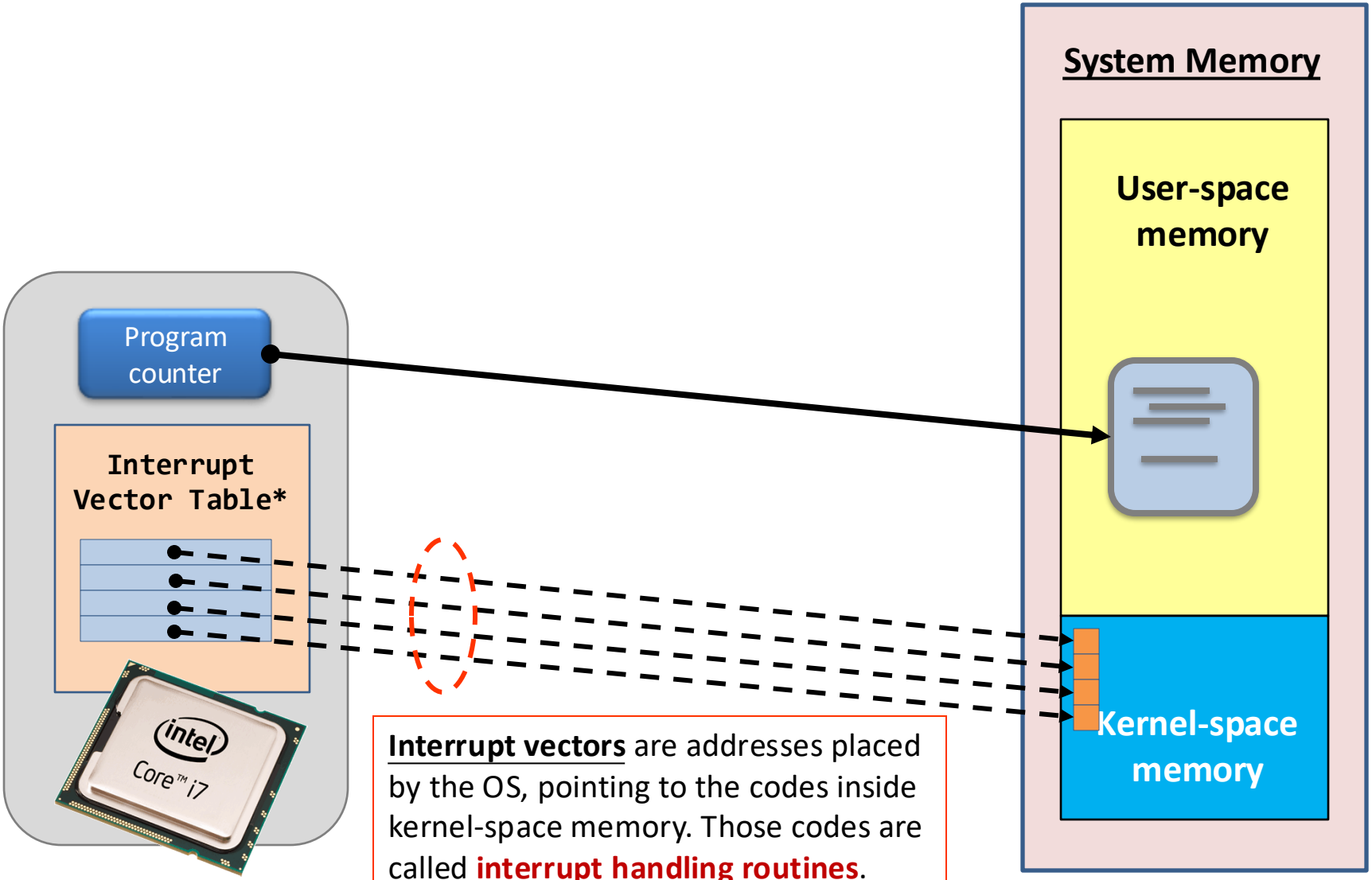
Hardware Interrupt

- External events that require CPU's attention

Originally, the CPU is working on a program code.



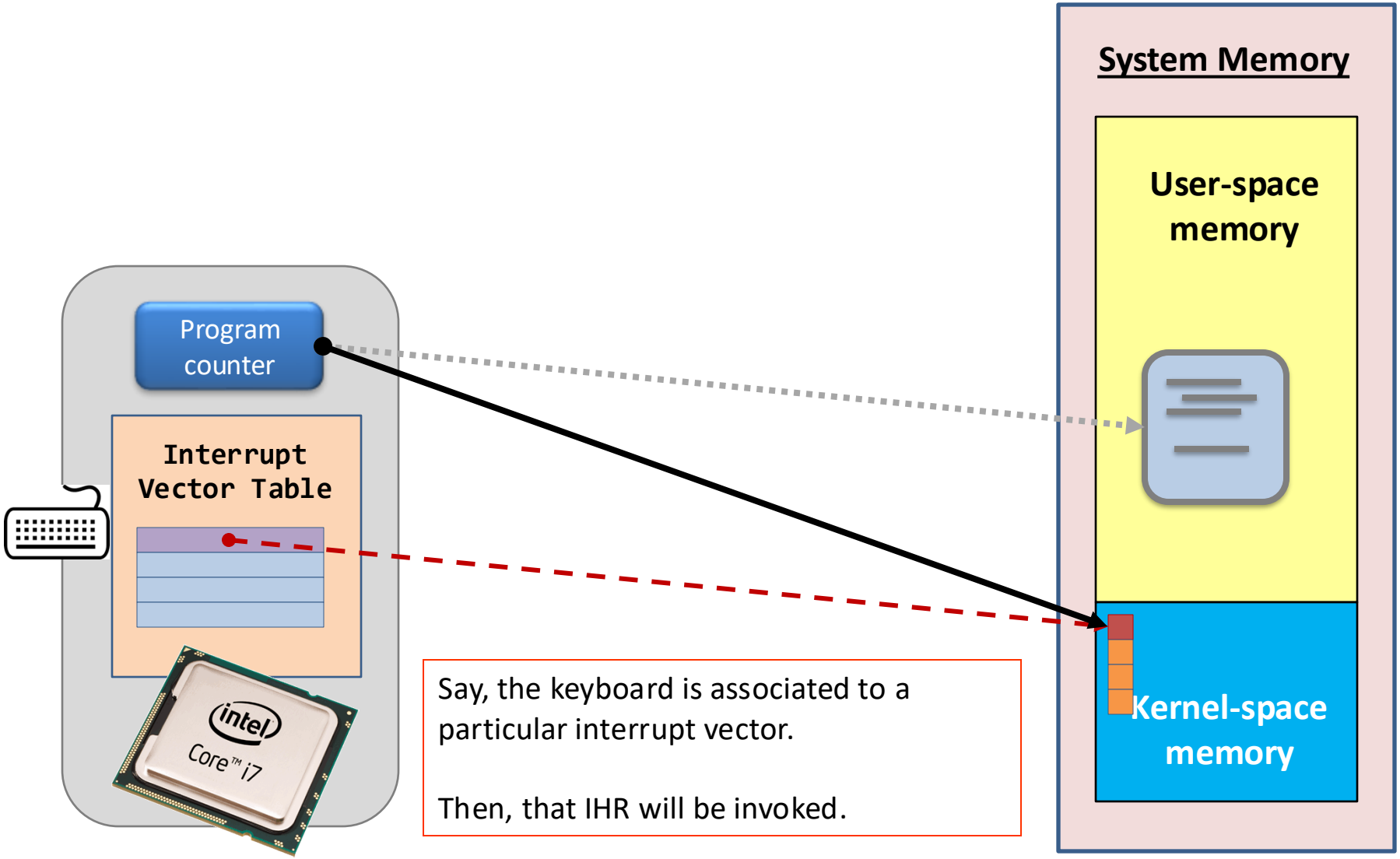
Interrupt Vector Table



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

*IVT's location is architecture-specific: maybe hardcoded in processor/designated space in the kernel-space

Hardware Interrupt

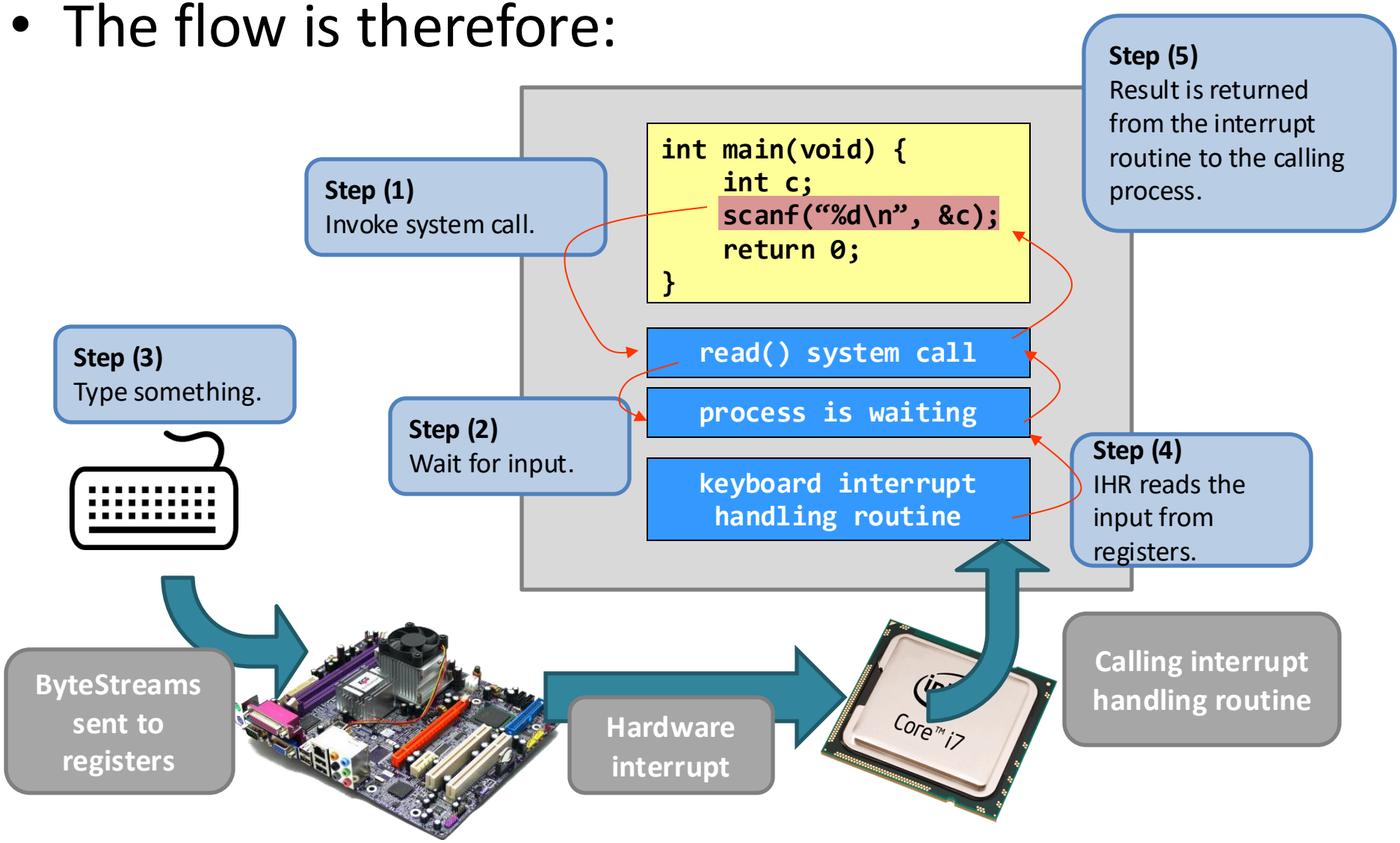


Say, the keyboard is associated to a particular interrupt vector.

Then, that IHR will be invoked.

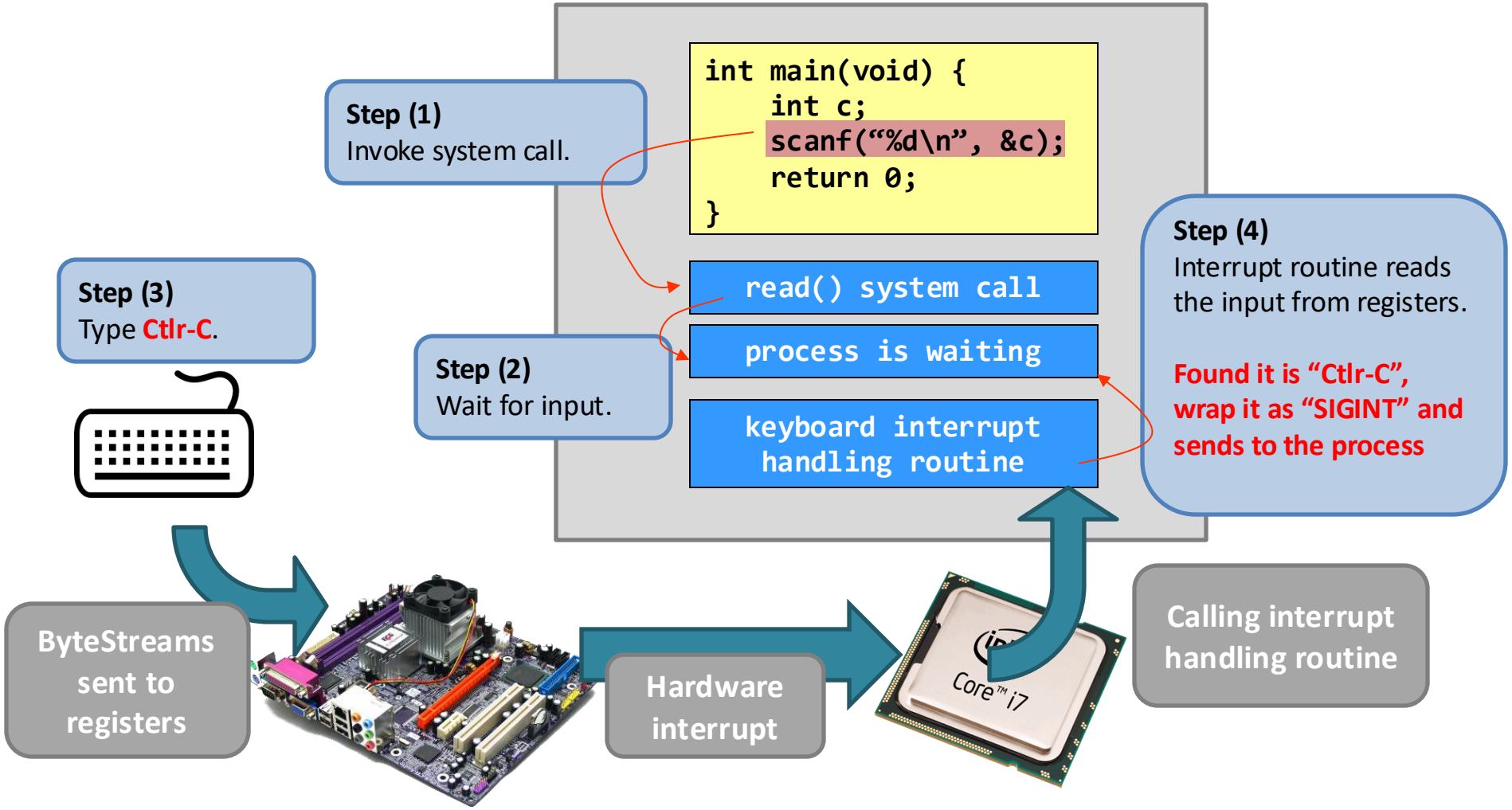
E.g., scanf

- The flow is therefore:



E.g., Ctrl-C to SIGINT

- The flow is therefore:



Process2Process: signal sending 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 ericlo
 PID   TTY
1234   ... ./infinite_loop
......
$ kill 1234
$ _
```

Terminal A

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

Terminal B

POSIX signals

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. (type 3)	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**.	Target process resumes execution if it is previously suspended.
SIGCHLD	(No special name). It is sent to the parent process from a terminated child via exit(). (type 2)	{ //do nothing by default }
SIGKILL	Its name is the **kill signal**. If sent, the process **MUST DIE**.	Target process termination (can't be replaced by other signal handlers)

Signal: Synchronous vs. Asynchronous

- Asynchronous signal:
  - The signal received is NOT generated by the process itself
    - So, its arrival time is **usually** not deterministic from the process point of view
  - E.g., External hardware interrupt, another process sends **`ctrl-c`**
- Synchronous signal:
  - The signal is caused by the process itself
    - So, its arrival time is **usually** deterministic from the process point of view
  - E.g., A certain line leads to **`SIGFPE`**
  - E.g., A certain line accesses an invalid memory region: **`SIGSEGV`**

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

```
$ ps -u ericlo
 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
```

Try press "enter" & then "ls"

The "terminal stop signal" suspends the process.

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

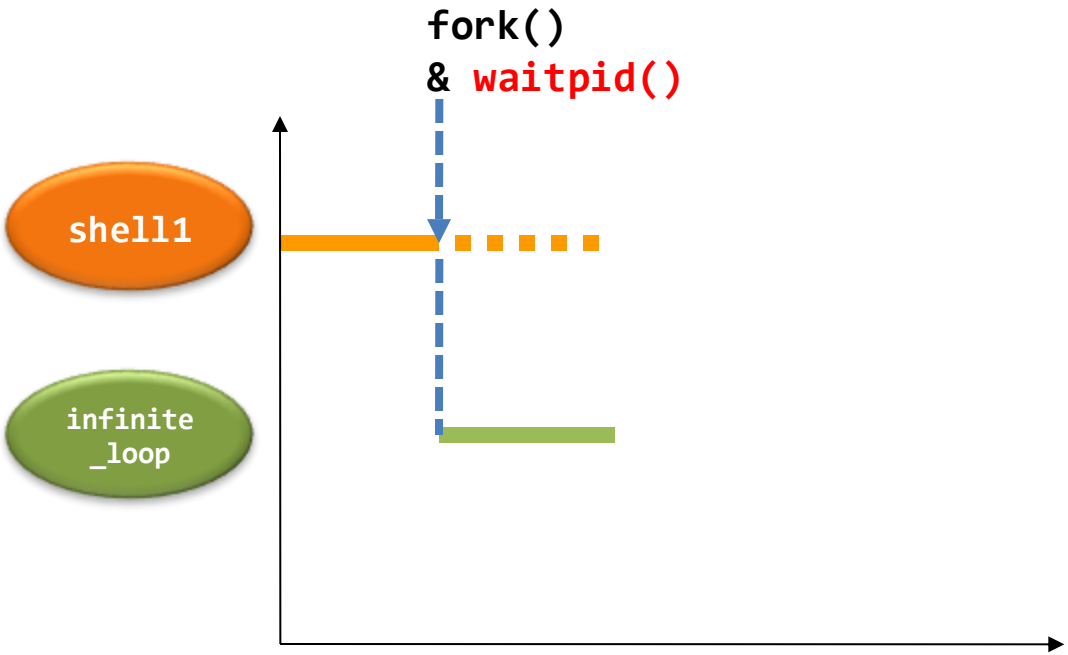
Shell 2

Shell 1

Foreground and background jobs

A characteristic of a foreground job in a shell is:
The shell is waiting for the job to change state.

Foreground job is a concept in a shell:
- read input from stdin
- write output to stdout




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

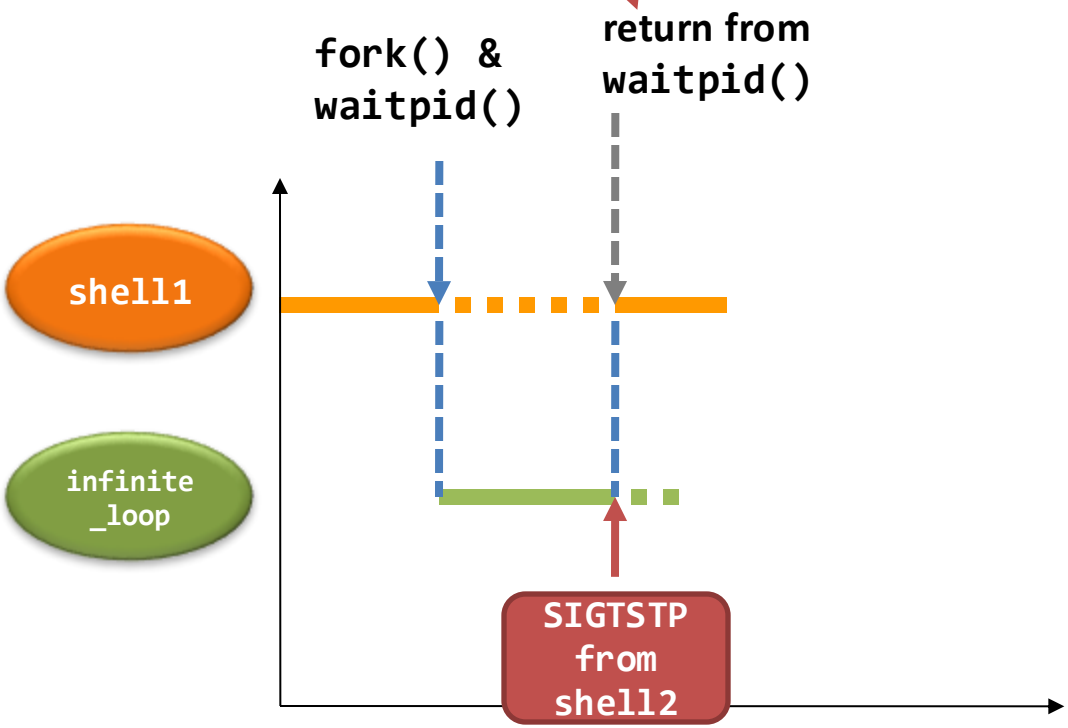
We name this a **foreground job**.

https://en.wikipedia.org/wiki/Job_control_(Unix)

Foreground and background jobs

This line is actually printed by the shell

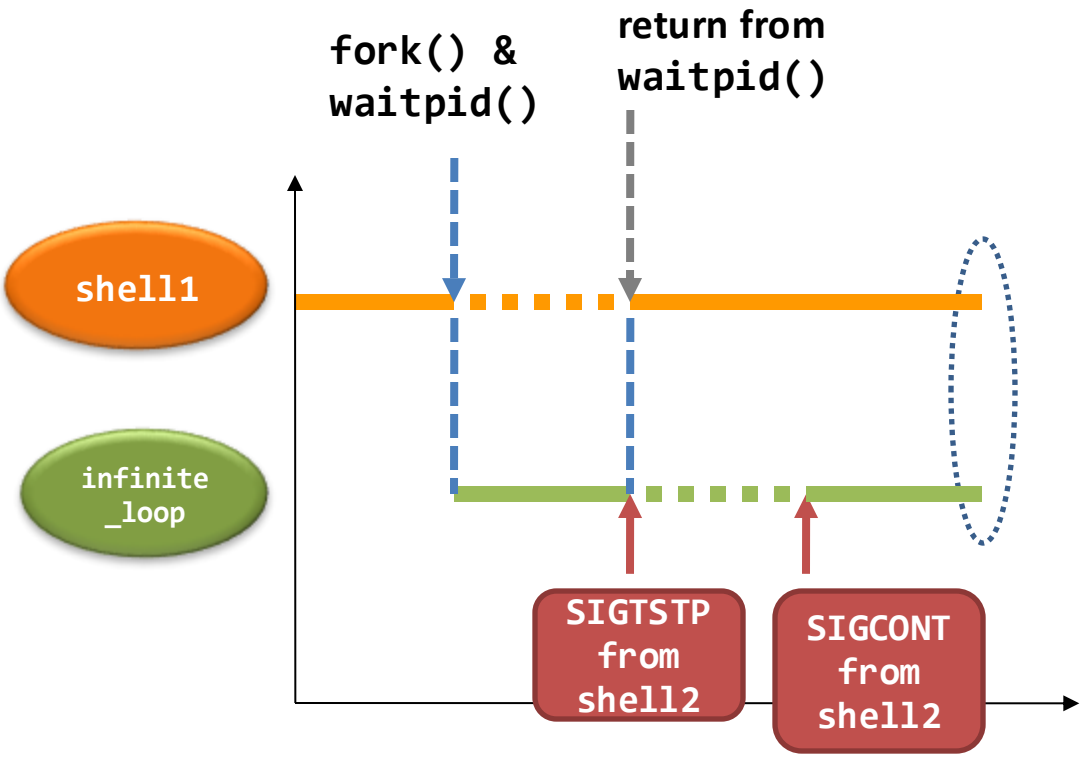
Now, the shell / parent wakes up.




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

Foreground and background jobs

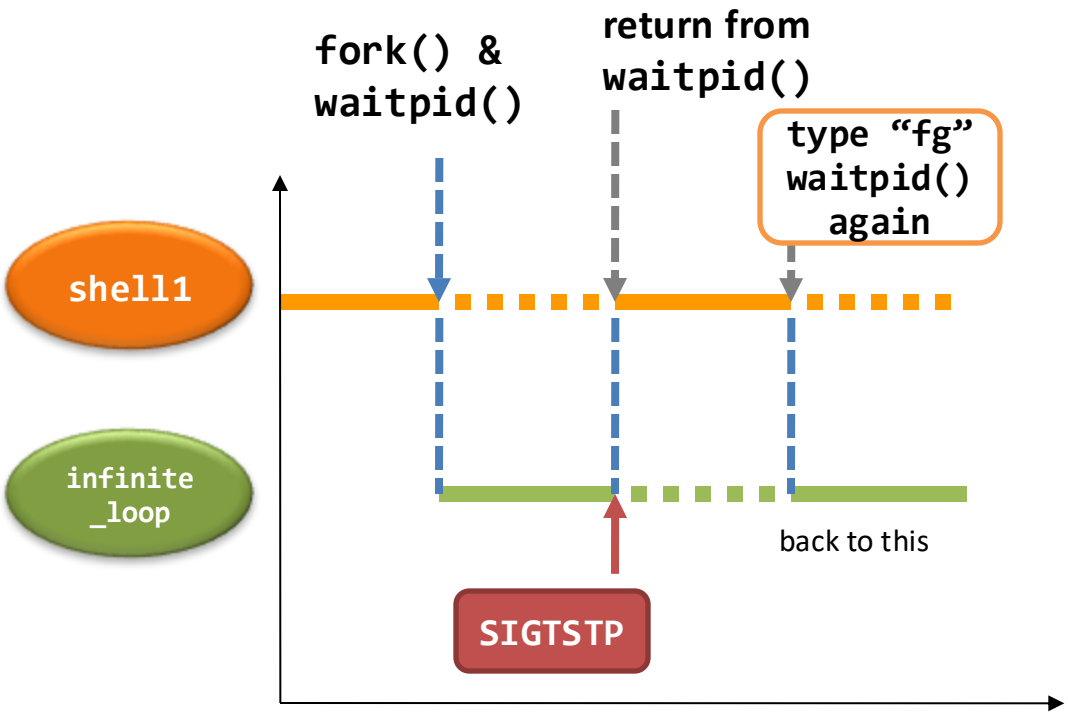
SIGCONT is fired from another shell process, say shell2 to infinite_loop
=> so shell1 doesn't know about that,
=> shell1 and infinite_loop just run in parallel.




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

infinite_loop inherits parent's (shell1) file so outputs to parent's stdout

Foreground and background jobs




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

Now, command **fg** turns its most recent suspended/background job (**infinite_loop**) back as a **foreground job**

That means shell1 is re-wait() again

Sending signals in a process

- Remember, **`kill()`** is intended to send signals.
- So, why not use the name **`signal()`**?
  - Answer: because **`signal()`** has already been used for registering a new signal handler

```
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
```

signal()^

- Register a new signal handler
  - The process **no longer executes the default handler**…
  - A signal is handled by a **user-defined/given function**

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

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

A function pointer to the new handler. This means your signal handler shall have:
- 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.

^deprecated: Recommend use: `sigaction` for multithreaded processes

Registering 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  }
```

Why put an 'if' here if we know it "always" handles SIGINT here?
Ans: yes, you are correct.
It isn't necessary here. But, check out handle_int.c to find the reason.

Line 7 registers the signal handler for **SIGINT**.

Lines 1-4 together define the customized signal handler.

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

Registering 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

Registering new signal handlers

- IMPORTANT:
  - **Apparently**, when a signal handler returns, **the process should go 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
```

Registering new signal handlers

- IMPORTANT:
  - **Apparently**, when a signal handler returns, **the process should go back to where it was executing**.
  - But…this only happens when **the involved system/library call can be resumed/restarted.**

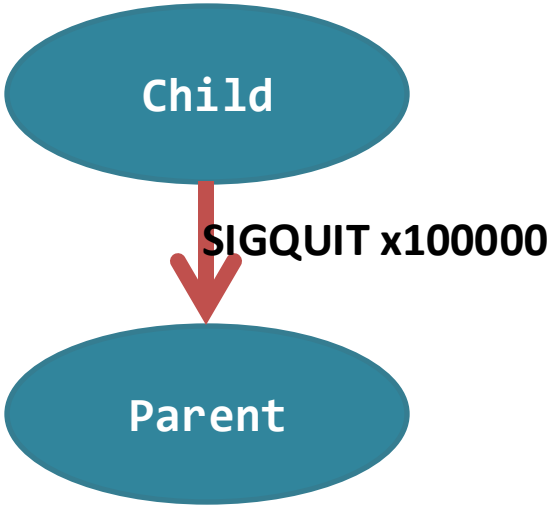
Can be resumed/restarted	Cannot be resumed/restarted

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

How to implement “Signal” indeed?

```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 < 100000; i++)
            kill(getppid(), SIGQUIT);
    }
    else {
        signal(SIGQUIT, handler);
        sleep(1000);
        wait(NULL);//wait until Child terminates
    }
  return 0;
}
```




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

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

How to implement “Signal” indeed?

- Standard signals are not implemented as a queue
- A bit 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 set to 1 once **`SIGQUIT`** is received.
  - The mask will be set to 0 once the signal is handled.

How does a process come to know that it has received a signal?
http://stackoverflow.com/questions/1860175/how-does-a-process-come-to-know-that-it-has-received-a-signal

Signal Implementations

- Check your knowledge, read this:
  - http://www.linuxjournal.com/article/3985?page=0,0
  - In the bottom, you'll read 2 kernel data structures:
    - current-signal – "*contains a bitmask of pending signals*"
    - sigqueue – "*double-linked list of pending signals*"
- Tip:
  - Read man 7 signal
    - http://man7.org/linux/man-pages/man7/signal.7.html

Misc. Topics

- Waiting for signals;
- Cleanup when facing Ctrl-C;
- Timers and periodic signals;

(1) - Waiting for a signal

- The **`pause()`** system call suspends the calling process until a signal is received.
- Vs. **`wait()`** :
  - **`wait()`** doesn't mean "I wait"
  - **`wait()`** means "I wait **for** …"
  - It suspends only when it has a child process

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

```
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) – Cleanup when facing **Ctrl+C**

```
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;
}
```

"**Ctrl+C**" terminates a program too ugly.

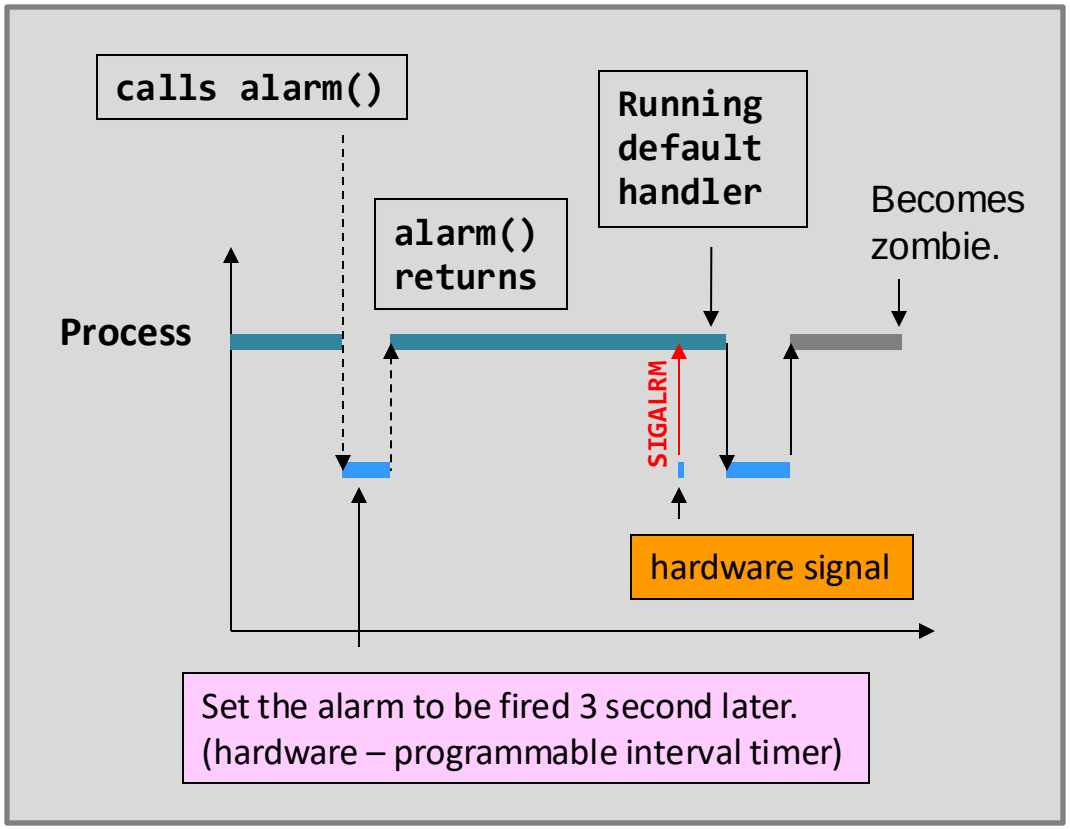
It allows the program to **exit gracefully when facing** "**Ctrl+C**", e.g., you can:

- close network connections,
- commit database changes,
- etc.

```
[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(3);
    while(1);
    return 0;
}
```

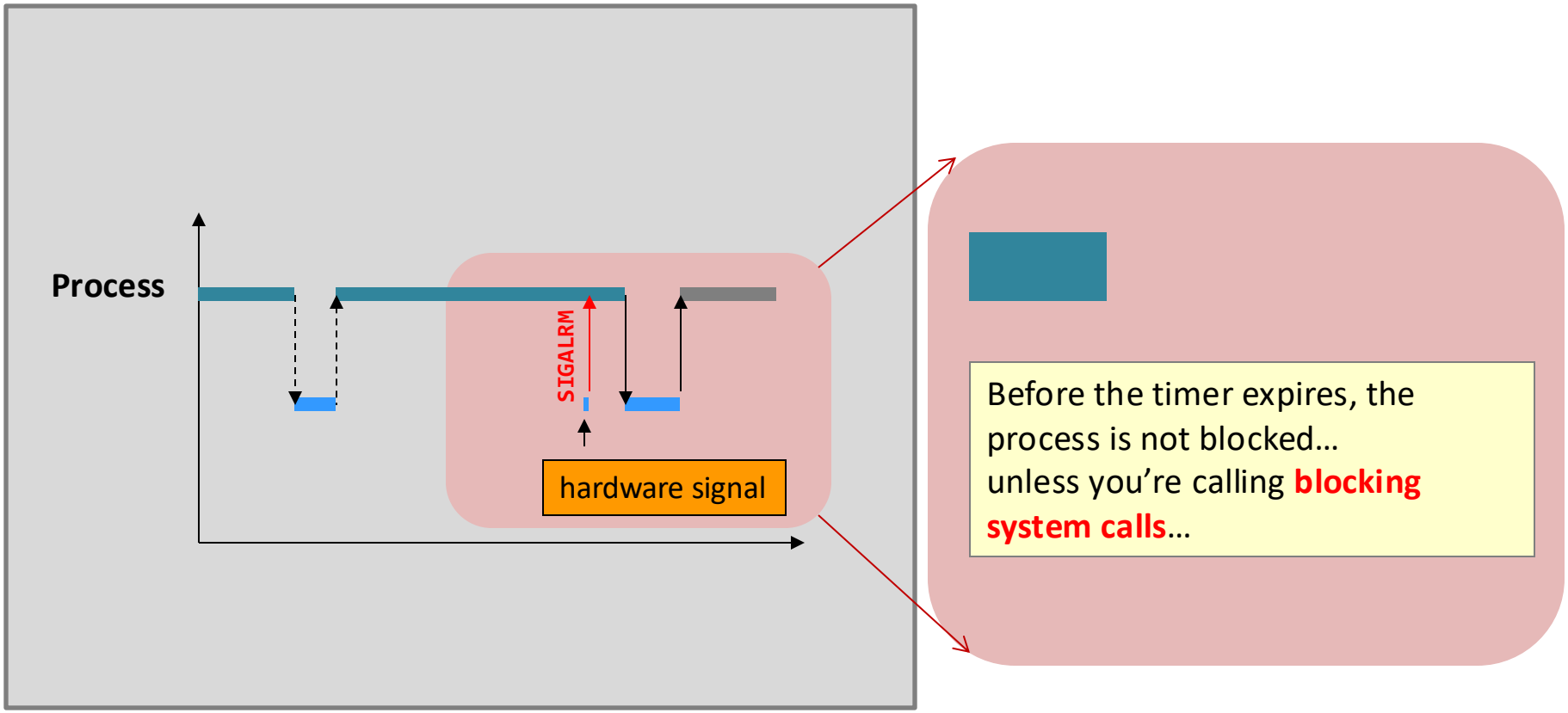
```
$ ./alarm
Alarm clock
$ _
```

3 sec later

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

(3) – Timer and alarm()

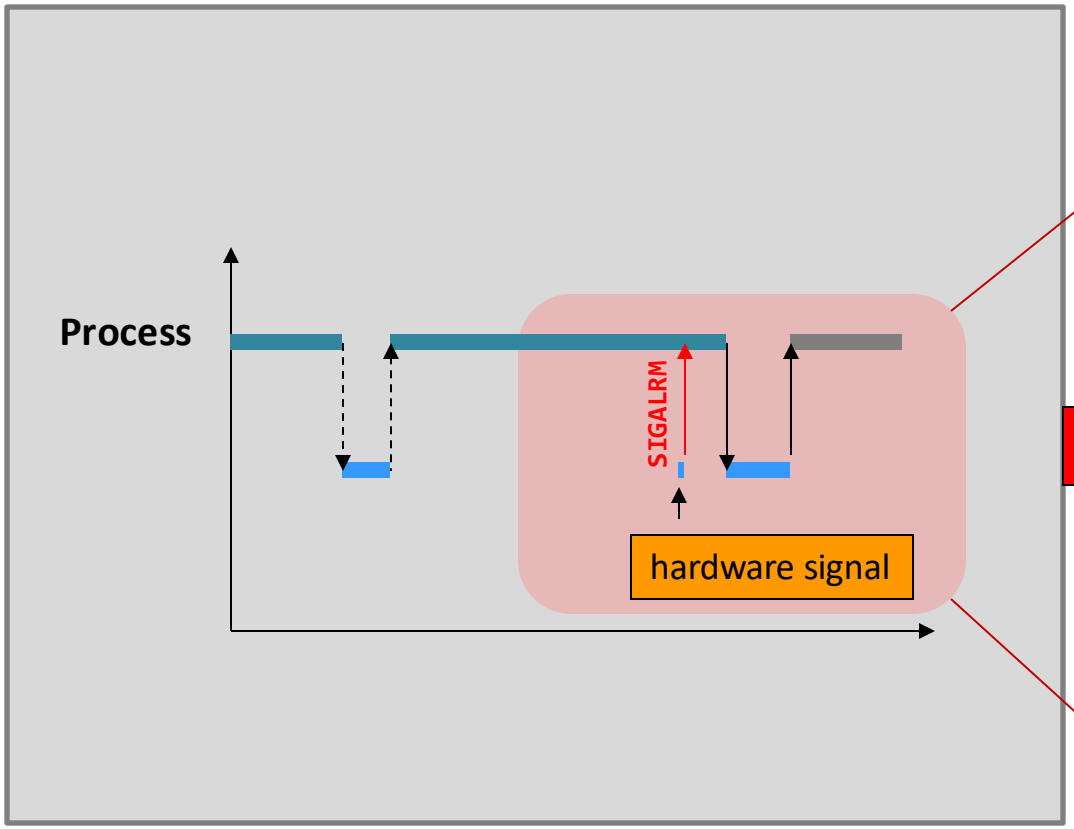
- **`alarm()`** is a system call that allows **asynchronous timing** for a process.




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

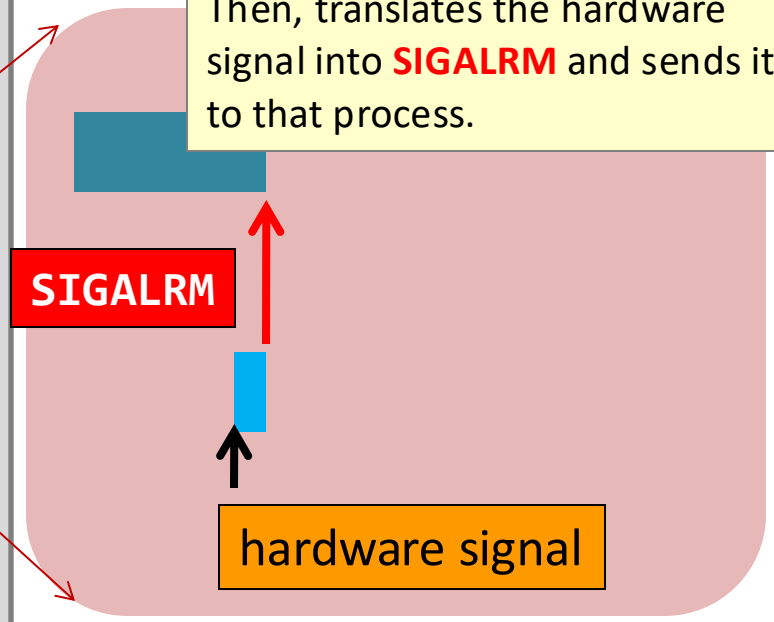
(3) – Timer and alarm()

- **`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.

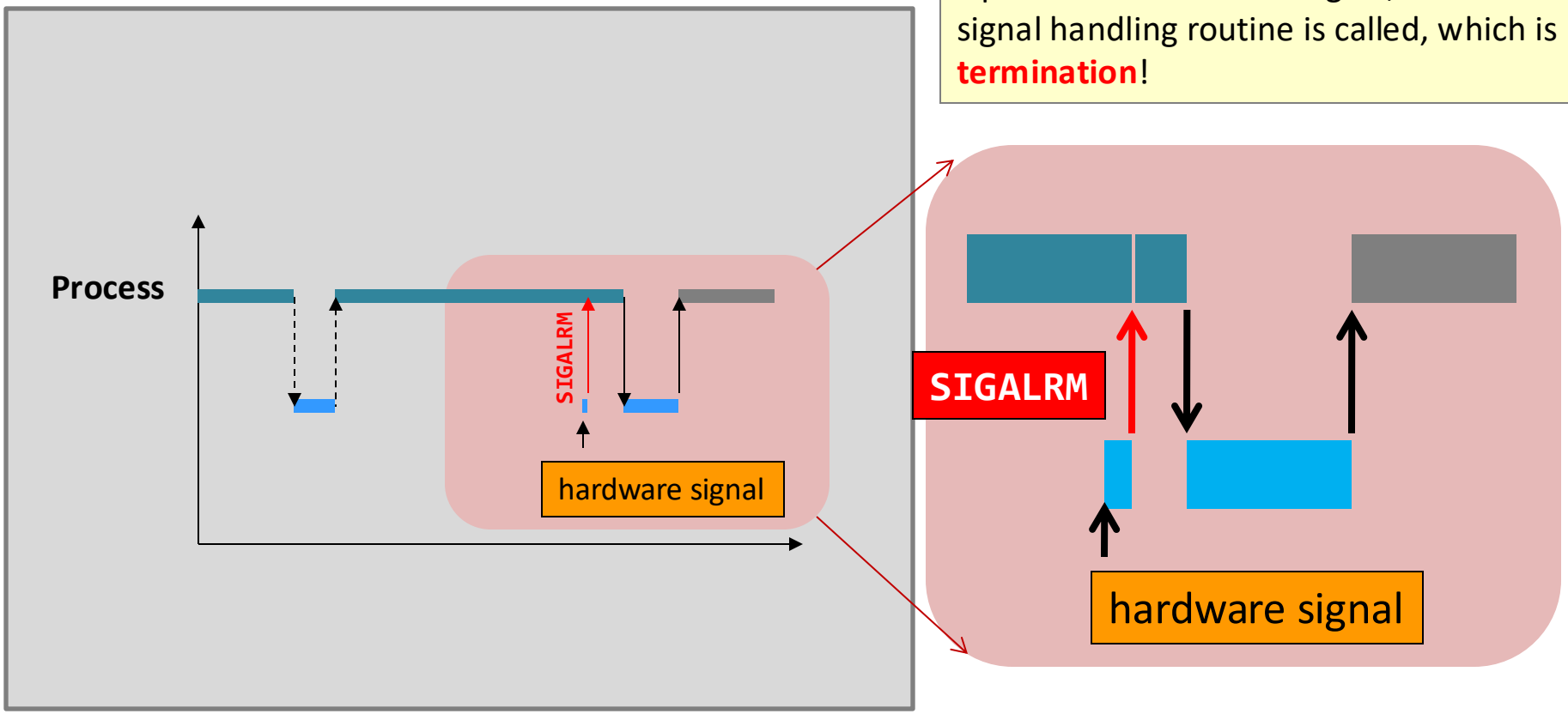



```
[examples@3150] cat alarm.c
```

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

Further reading

- *Advanced Programming Environment in UNIX*;
- "**`man signal`**" is a vast resource.

When will a process check for an arrival of signal?

A. Decode phase of every instruction
B. When context switch back to the process
C. A signal arrival will update the program counter to the signal handler
D. None of the above

Join at:
ahaslides.com/
0LQ2X