Interrupts

Hardware and **Software**

Real Events	ID	Action Function
Building on fire	1	Get out immediately!
Telephone rings	2	Pick up phone to chat with the caller.
Knock on door	3	Yell come in (or pretend not there).
Cut own finger	4	Apply band-aid.
. . . . .	.	. . . . .

From hardware : building on fire, alarm clock goes off, etc.

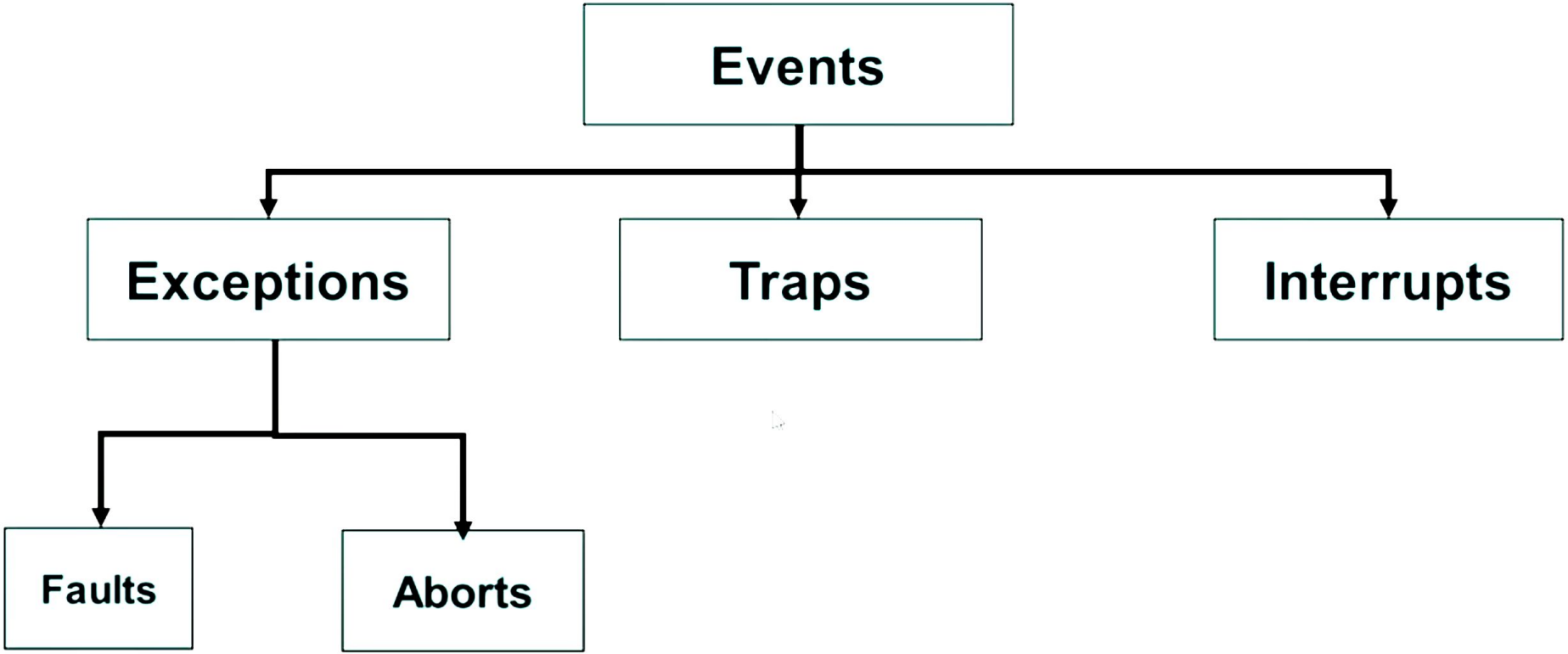
From other person: phone call, knocking on door, etc.

Self-inflicted : cut own finger, eat too much, etc.

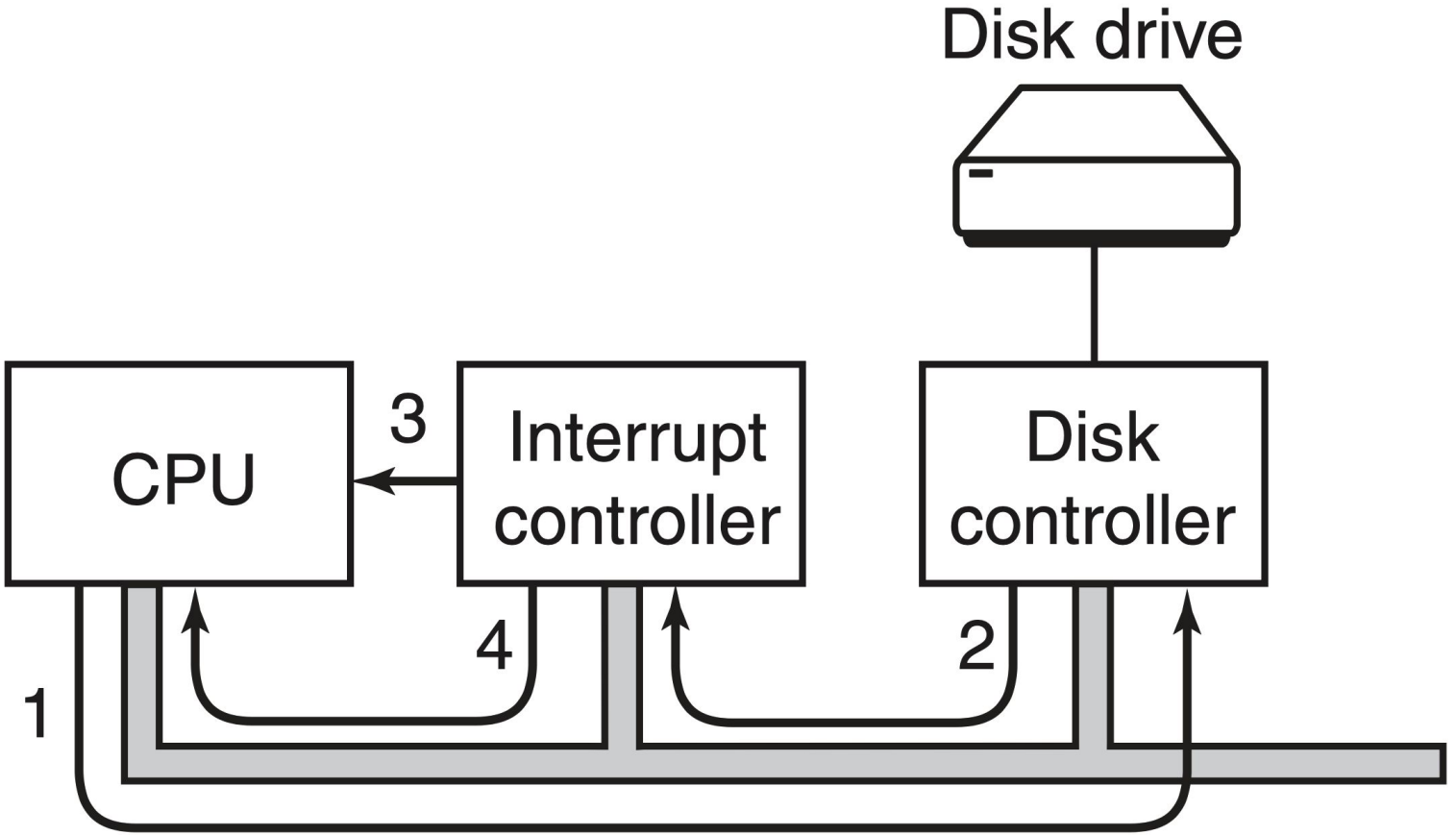
Real Events	ID	Action Function
Building on fire	1	Get out immediately!
Telephone rings	2	Pick up phone to chat with the caller.
Knock on door	3	Yell come in (or pretend not there).
Cut own finger	4	Apply band-aid.
. . . . .	.	. . . . .

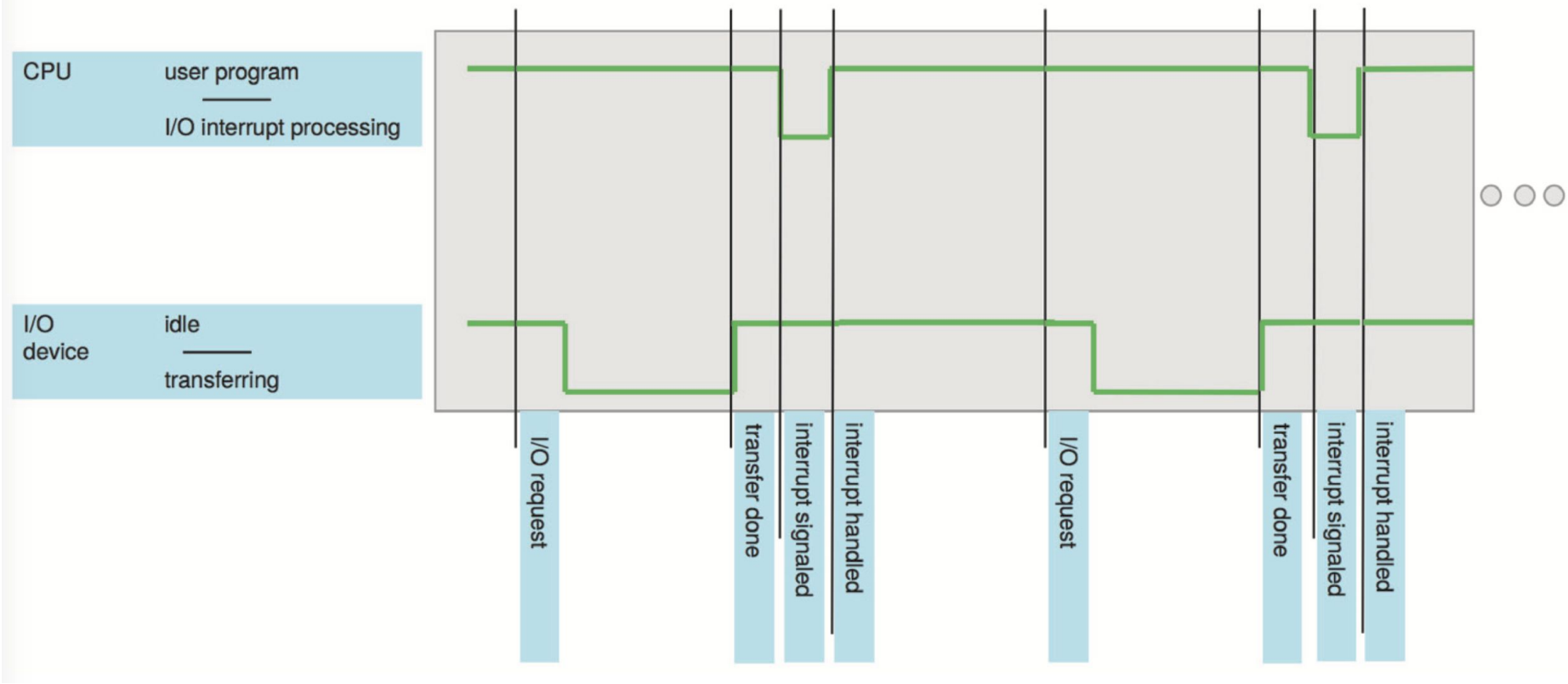
From hardware : building on fire, alarm clock goes off, etc.
From other person: phone call, knocking on door, etc.
Self-inflicted : cut own finger, eat too much, etc.

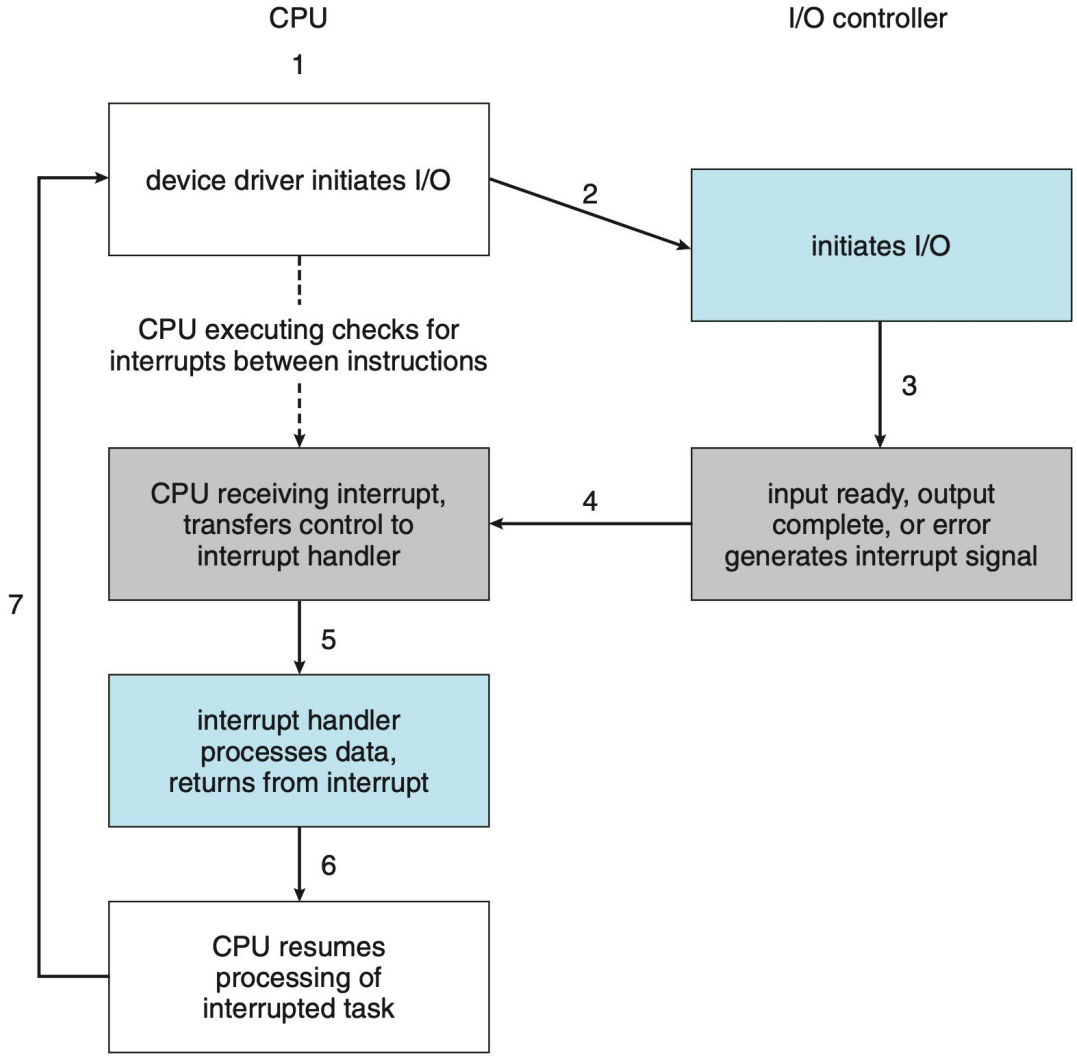
From hardware : Control_C key from terminal, interval timer, etc.
From other process: kill(pid, SIG#), death_of_child, etc.
Self-inflicted : divide by zero, invalid address, etc.


Events
Exceptions
Traps
Interrupts
Faults
Aborts

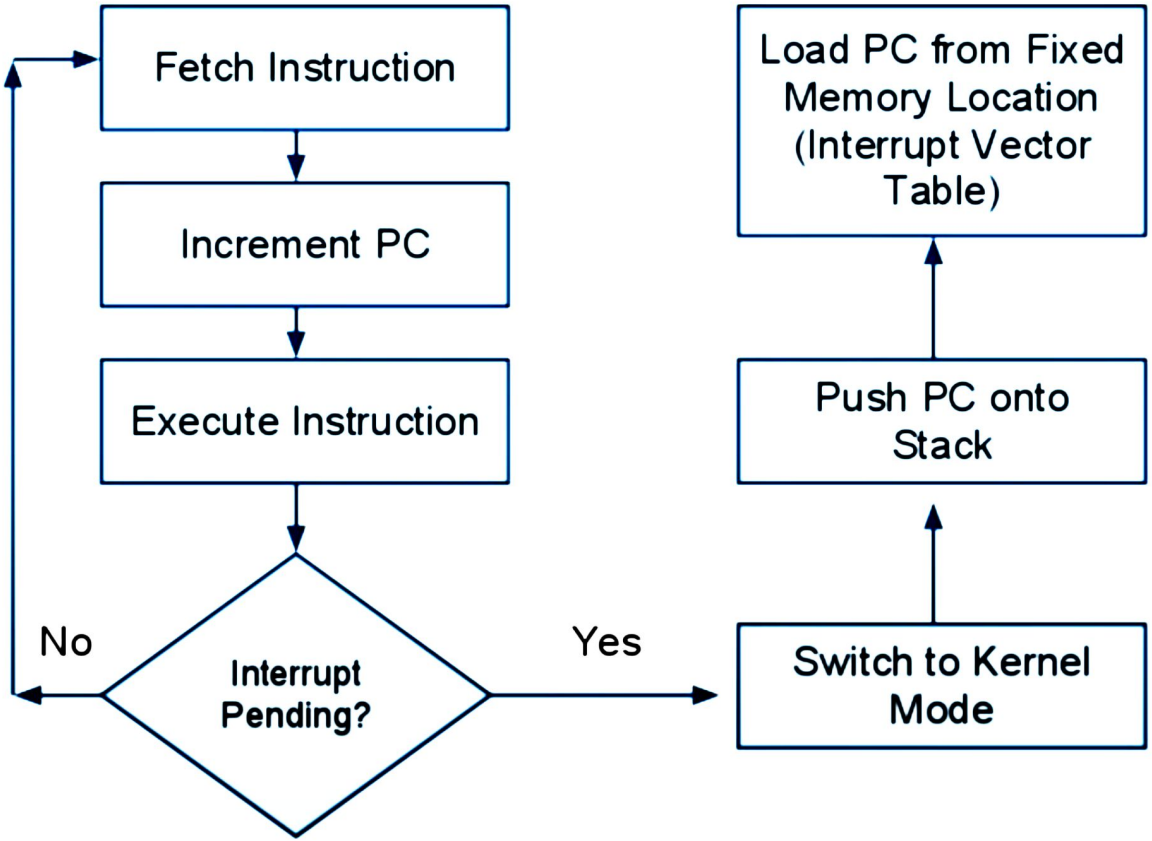
Interrupts (hardware)




CPU
user program
I/O interrupt processing
I/O device
idle
transferring
I/O request
transfer done
interrupt signaled
interrupt handled
I/O request
transfer done
interrupt signaled
interrupt handled


CPU
I/O controller
1
device driver initiates I/O
2
initiates I/O
CPU executing checks for interrupts between instructions
3
CPU receiving interrupt, transfers control to interrupt handler
4
input ready, output complete, or error generates interrupt signal
7
5
interrupt handler processes data, returns from interrupt
6
CPU resumes processing of interrupted task

Fetch-Execute Cycle



Common Functions of Interrupts

- Interrupt **transfers control** to the interrupt service routine generally, through the **interrupt vector**, which contains the addresses of all the service routines
- Interrupt architecture must save the address of the interrupted instruction
- A **trap** or **exception** is a software-generated interrupt caused either by an error or a user request
- An operating system is **interrupt driven**

Signals

- Interrupts sent to a process
- Three different sources
  - From **hardware**: Control+C key from terminal, interval timer, etc.
  - From **other process**: kill (pid, SIG#), death_of_child, etc.
  - **Self-inflicted**: divide by zero, invalid address, etc.

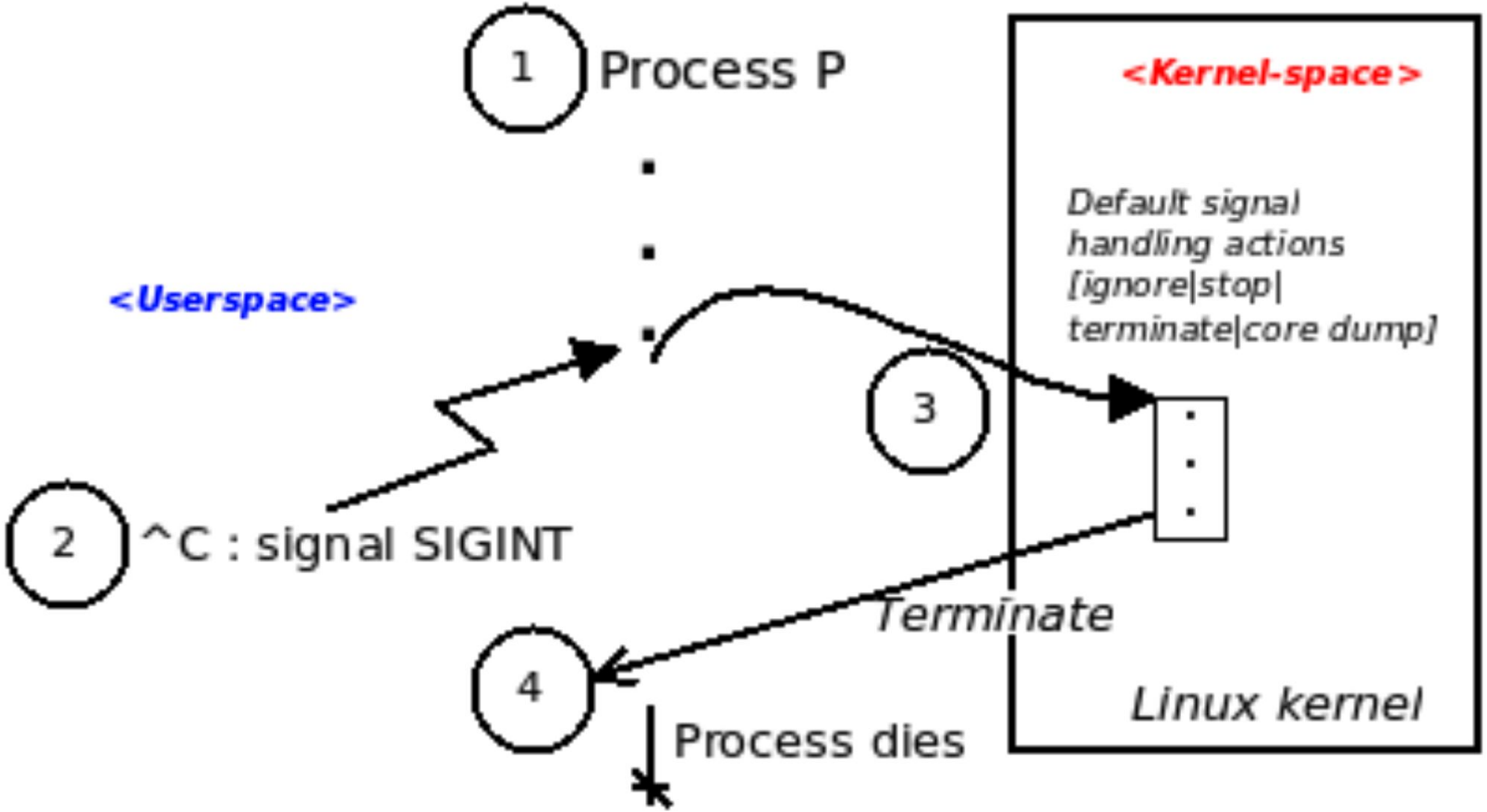
Signals

- asynchronous events delivered to a target process.
- every signal is associated with a function to run when it is delivered
  - this function is called the **signal handler**
- a signal is purely a **software** mechanism
- at the code level, a signal is merely an integer value (a bit in a bitmask)

```
int main(void)
{
     unsigned long int i=1;
     while(1) {
          printf("Looping, iteration #%02ld ...\n", i++);
          (void)sleep(1);
     }
     exit (EXIT_SUCCESS);
}
```

```
$ ./sig1
Looping, iteration #01 ...
Looping, iteration #02 ...
Looping, iteration #03 ...
^C
$
```

- user presses Ctrl + C (^C)
- kernel's **tty** layer code processes the input
- delivers a signal to the foreground process on the shell
- SIGINT (2)


1
Process P
<Kernel-space>
Default signal
handling actions
[ignore|stop|
terminate|core dump]
<Userspace>
3
2
^C : signal SIGINT
Terminate
4
Process dies
Linux kernel

Why signals?

- Systems programmer requires the **asynchronous facilities from OS**
- A process can trap or subscribe to a signal
  - The process is asynchronously notified by the OS
  - Then run the code of a function in response: a signal handler.

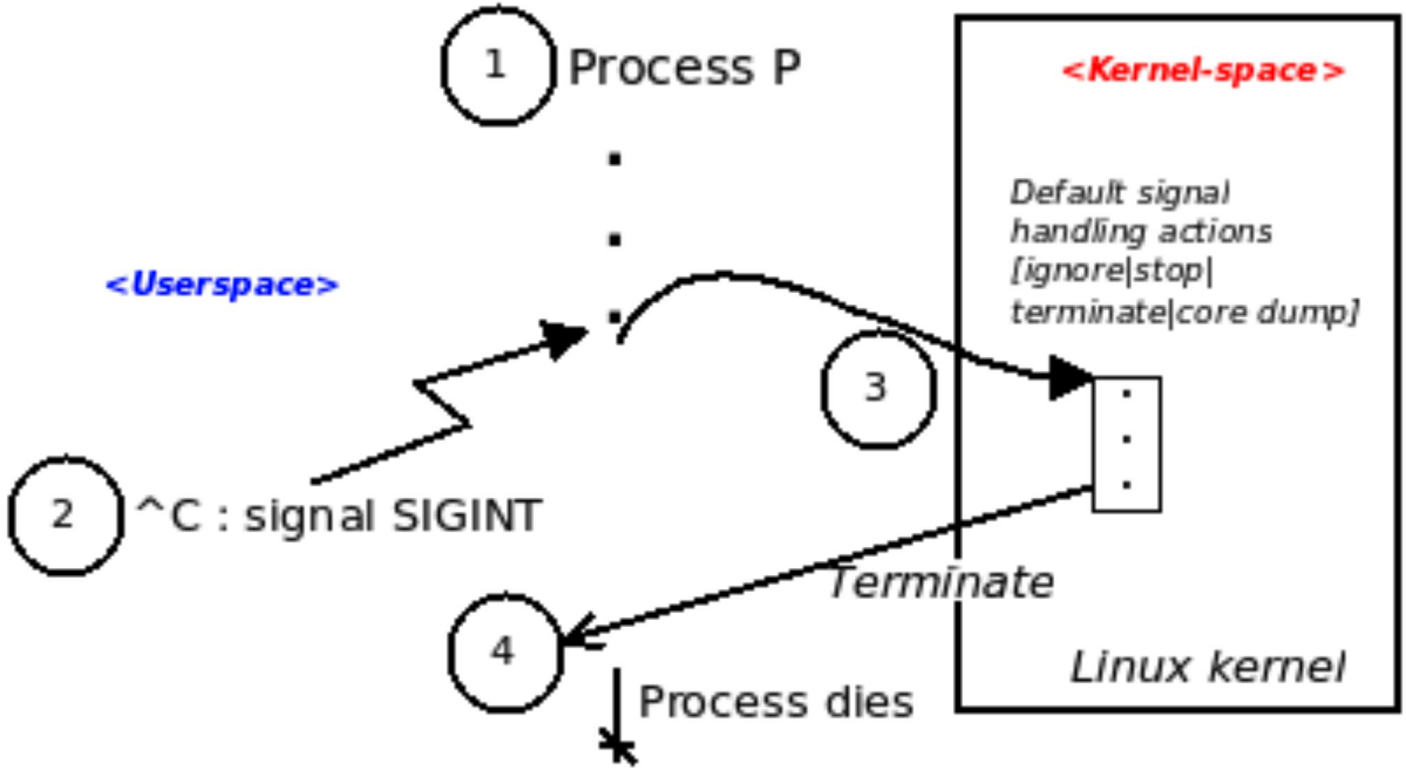
Signal	Integer value	Default action	Comment
SIGHUP	1	Terminate	Hang up detected on controlling terminal or death of controlling process
SIGINT	2	Terminate	Interrupt from keyboard : ^C
SIGQUIT	3	Term&Core	Quit from keyboard : ^\
SIGILL	4	Term&Core	Illegal Instruction
SIGABRT	6	Term&Core	Abort signal from abort(3)
SIGFPE	8	Term&Core	Floating-point exception
SIGKILL	9	Terminate	(Hard) kill signal
SIGSEGV	11	Term&Core	Invalid memory reference
SIGPIPE	13	Terminate	Broken pipe: write to pipe with no readers; see pipe(7)
SIGALRM	14	Terminate	Timer signal from alarm(2)
SIGTERM	15	Terminate	Termination signal (soft kill)
SIGUSR1	30,10,16	Terminate	User-defined signal 1
SIGUSR2	31,12,17	Terminate	User-defined signal 2
SIGCHLD	20,17,18	Ignore	Child stopped or terminated
SIGCONT	19,18,25	Continue	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at terminal : ^Z
SIGTTIN	21,21,26	Stop	Terminal input for background process
SIGTTOU	22,22,27	Stop	Terminal output for background process

Default actions

- **Terminate**: Terminate the process.
- **Term&Core**: Terminate the process and emit a **core dump** (a snapshot of the process's dynamic segments, the data and stack segments).

  Sent when the process has done something illegal (buggy).
- **Ignore**: Ignore the signal.
- **Stop**: Process enters the stopped (frozen/suspended) state (T)
- **Continue**: Continue execution of a previously stopped process.

Handling signals



Handling signals

```
int r = signal(int signal_number, void *handler);
```

- **change the handler** function of a selected signal number
- **except** SIGKILL(9) and SIGSTOP(19) (can not be changed)
- problems with **signal()** function
  - Race conditions
  - can not block other signals
  - may not work for threads in a multi-threaded program
- has been replaced by the POSIX **sigaction()** function

```
void handler(int sig, siginfo_t *siginfo, void *context)
{
  printf("handler: sig=%d from PID=%d UID=%d\n",
                  sig, siginfo->si_pid, siginfo->si_uid);
}

int main(int argc, char *argv[])
{
  struct sigaction act;
  memset(&act, 0, sizeof(act));
  act.sa_sigaction = &handler;
  act.sa_flags = SA_SIGINFO;
  sigaction(SIGTERM, &act, NULL);
  printf("proc PID=%d looping\n”);
  printf(“enter kill PID to send SIGTERM signal to it\n", getpid());
  while(1)(
    sleep(10);
  }
}
```

Sending signals (**kill** not just kills)

```
$ kill -l
 1) SIGHUP 2) SIGINT 3) SIGQUIT 4) SIGILL 5) SIGTRAP
 6) SIGABRT 7) SIGBUS 8) SIGFPE 9) SIGKILL 10) SIGUSR1
11) SIGSEGV 12) SIGUSR2 13) SIGPIPE 14) SIGALRM 15) SIGTERM
16) SIGSTKFLT 17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP
21) SIGTTIN 22) SIGTTOU 23) SIGURG 24) SIGXCPU 25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO 30) SIGPWR
31) SIGSYS    34) SIGRTMIN 35) SIGRTMIN+1 36) SIGRTMIN+2 37)
SIGRTMIN+3
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42)
SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12  53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7 58) SIGRTMAX-6 59)
SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2 63) SIGRTMAX-1
64) SIGRTMAX
```

Proper use of signals

Unix signals are originally designed for these purposes.

- A unified treatment of process exceptions:
  - When a process encounters an exception, it traps to kernel mode, converts the trap reason to a signal number and sends the signal to itself.
  - If the exception occurred in kernel mode, the kernel prints a PANIC message and stops.
  - If the exception occurred in user mode, the process typically terminates with a memory dump for debugging.
- To allow processes to handle program errors in user mode by preinstalled signal catchers.
- Under unusual conditions, it allows a process to kill another process by a signal.
  - Note that kill does not kill a process outright; it is only a “please die” plea to the target process. **Why can’t we kill a process outright?**

(11). Misuse of Signals: In many OS books, signals are classified as a mechanism for inter-process communication. The rationale is that a process may send a signal to another process, causing it to execute a preinstalled signal handler function. The classification is highly debatable, if not inappropriate, for the following reasons.

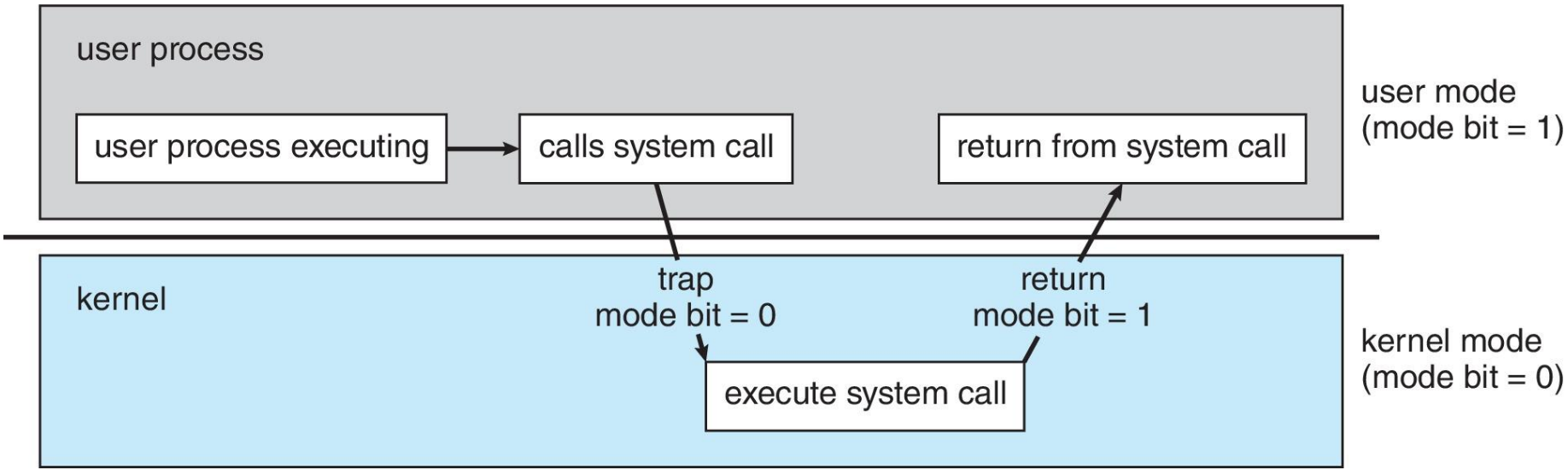
. The mechanism is unreliable due to possible missing signals. Each signal is represented by a single bit in a bit-vector, which can only record one occurrence of a signal. If a process sends two or more identical signals to another process, they may show up only once in the recipient PROC. Real-time signals are queued and guaranteed to be delivered in the same order as they are sent, but an OS kernel may not support real-time signals.

. Race condition: Before processing a signal, a process usually resets the signal handler to DEFault. In order to catch another occurrence of the same signal, the process must reinstall the catcher function BEFORE the next signal arrives. Otherwise, the next signal may cause the process to terminate. Although the race condition could be prevented by blocking the same signal while executing the signal catcher, there is no way to prevent missing signals.

. Most signals have predefined meaning. Indiscriminate use of signals may not achieve communication but confusion. For example, sending a SIGSEGV(11) segmentation fault signal to a looping process is like yelling to a swimmer in the water: "Your pants are on fire!".

Therefore, trying to use signals as a means of interprocess communication is over stretching the intended purpose of signals, which should be avoided.

System Calls



Some POSIX system calls

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

Some POSIX system calls

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing, or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Some POSIX system calls

Directory- and file-system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Some POSIX system calls

Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970