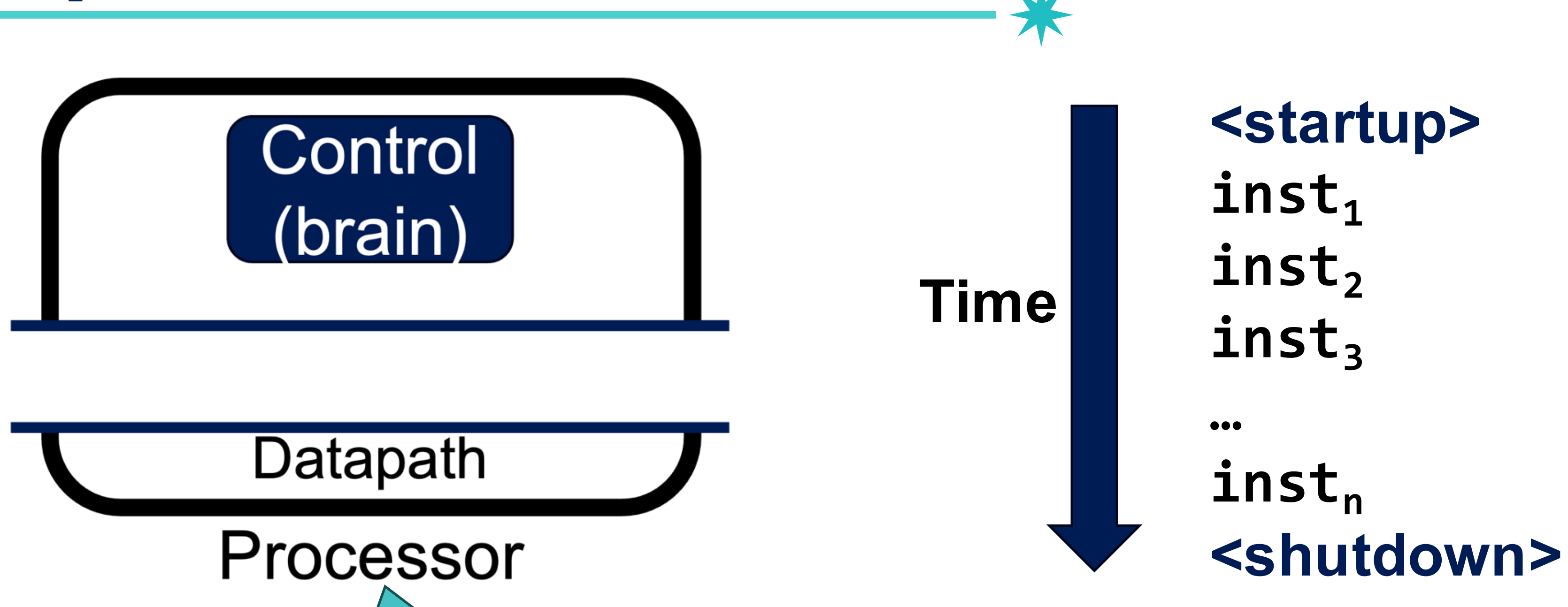
CSE251: System Programming

23. Exceptional Control Flow (ECF)

Seongil Wi

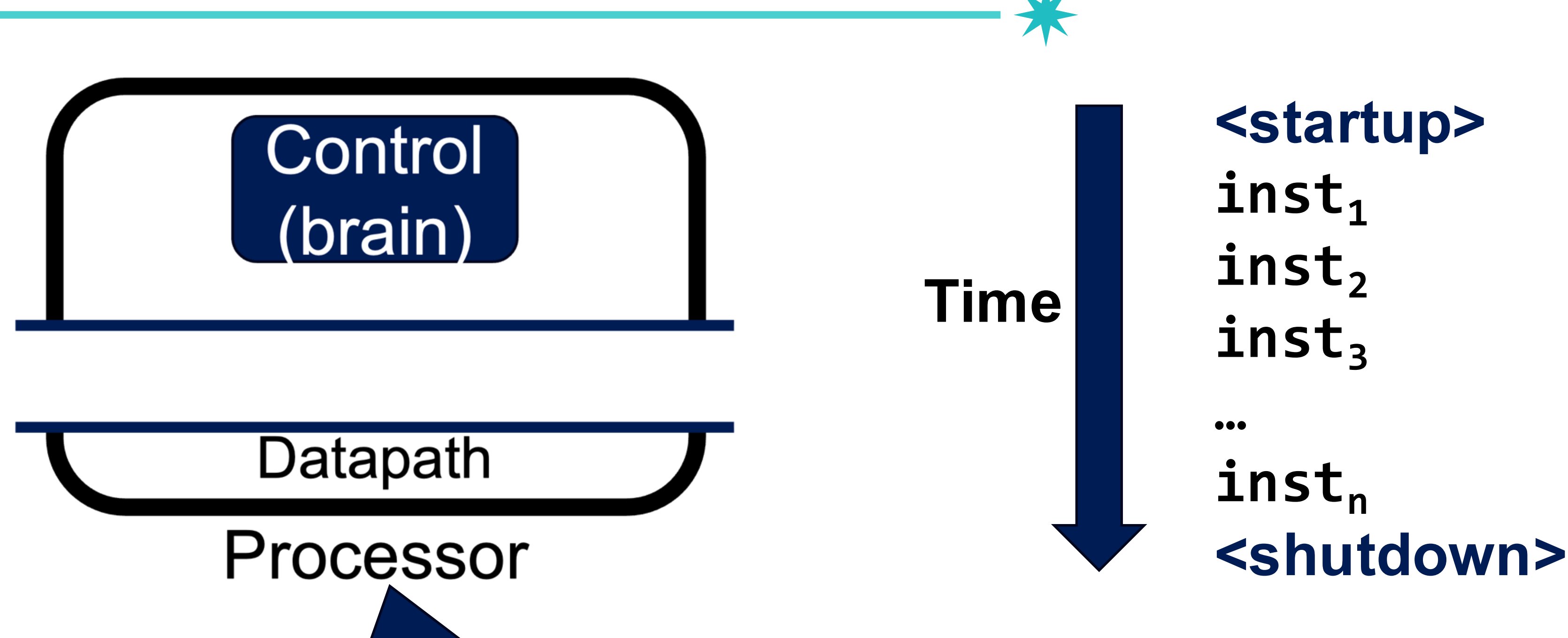
Department of Computer Science and Engineering

Recap: Control Flow



Only do one thing: simply reads and executes a sequence of instructions, one at a time

Question!



How to alter the control flow?

Altering the Control Flow

- Up to now: two mechanisms for changing control flow:
  - Jumps and branches
  - Call and return

- Insufficient for a useful system: Difficult to react to changes in ***system state***. For example, …
  - Data arrives from a disk or a network adapter
  - Instruction divides by zero
  - User hits Ctrl-C at the keyboard
  - System timer expires

System needs mechanisms from "***exceptional control flow (ECF)***"

Exceptions

Exceptions

A transfer of control to the ***OS kernel*** in response to some event

Exceptions

A transfer of control to the ***OS kernel*** in response to some event

- Kernel is the memory-resident part of the OS

Exceptions

A transfer of control to the ***OS kernel*** in response to some event

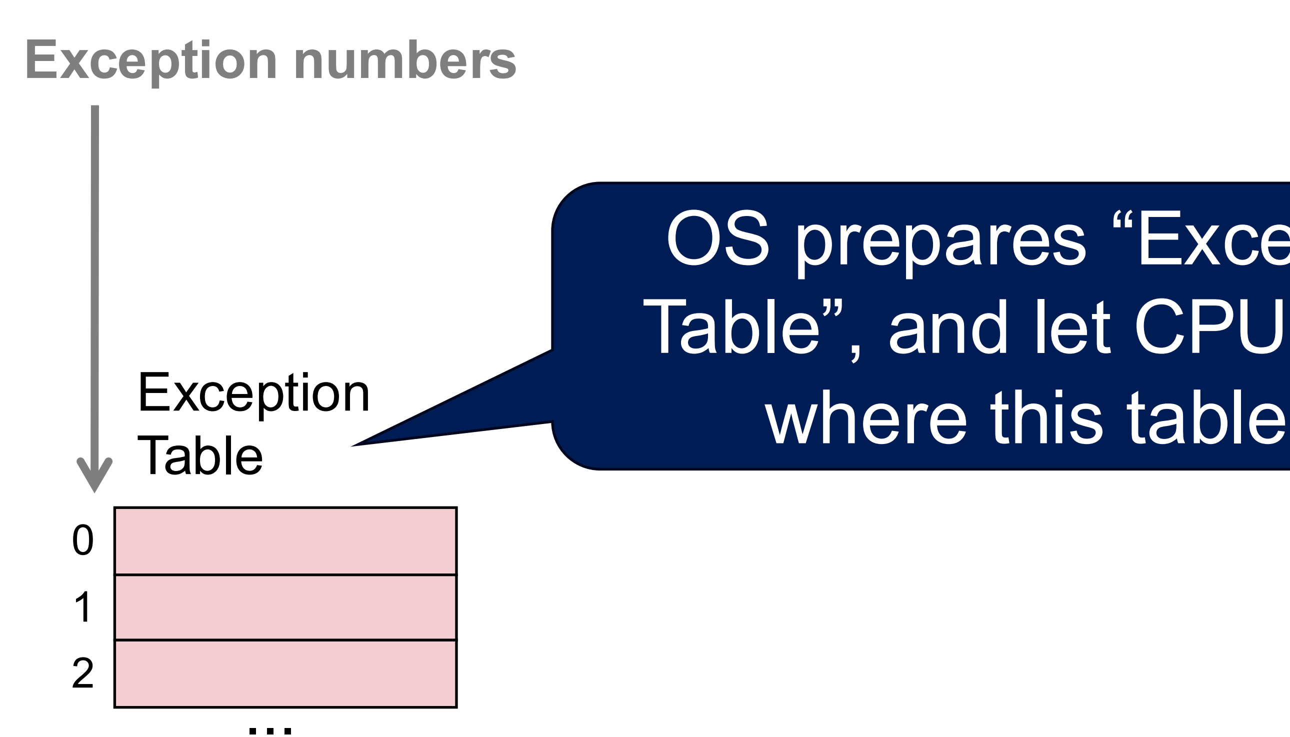
- Kernel is the memory-resident part of the OS
- Examples of events: Divide by 0, arithmetic overflow, page fault, I/O requests completes, typing Ctrl-C

Exceptions

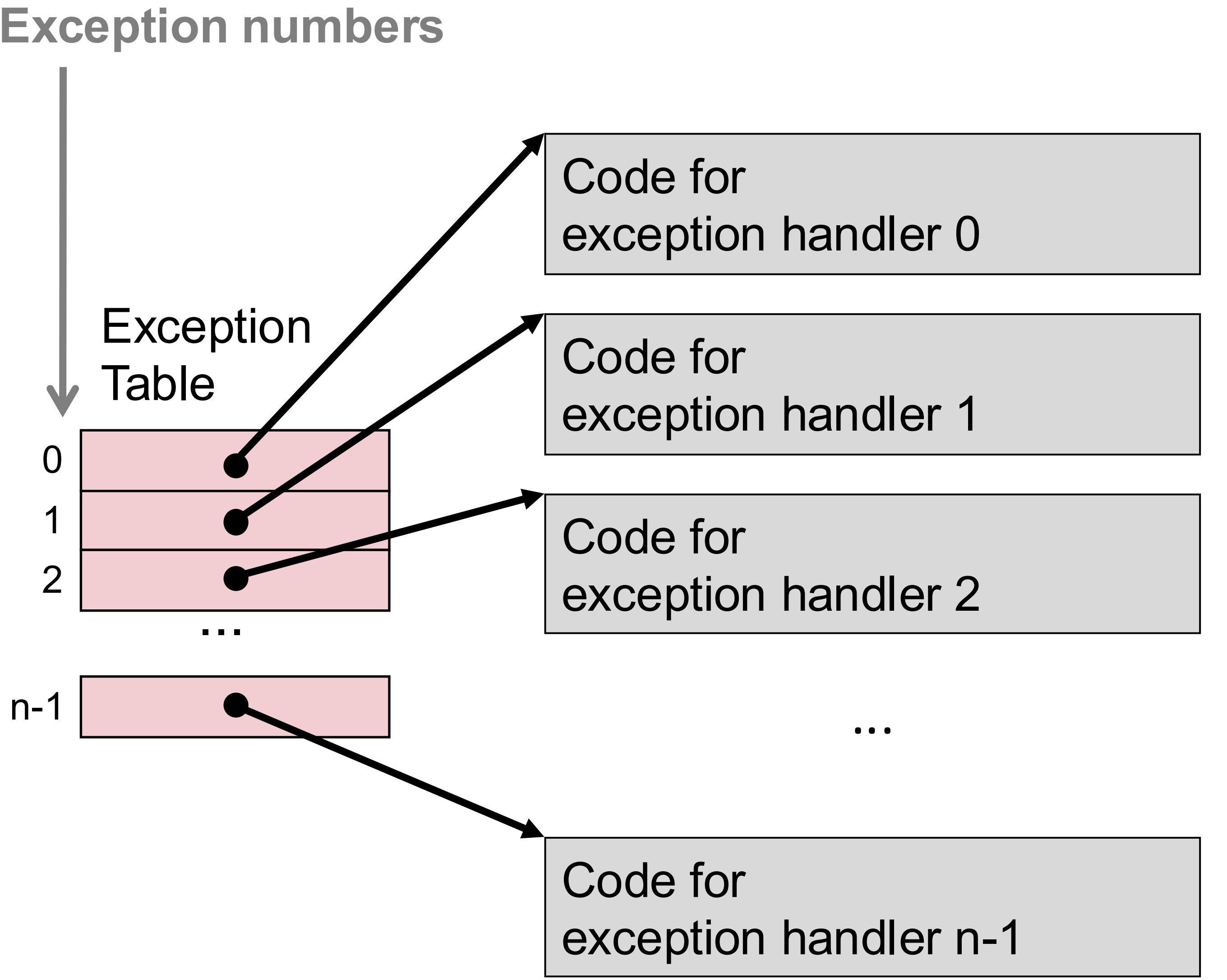
A transfer of control to the ***OS kernel*** in response to some event

- Kernel is the memory-resident part of the OS
- Examples of events: Divide by 0, arithmetic overflow, page fault, I/O requests completes, typing Ctrl-C

Exception Tables

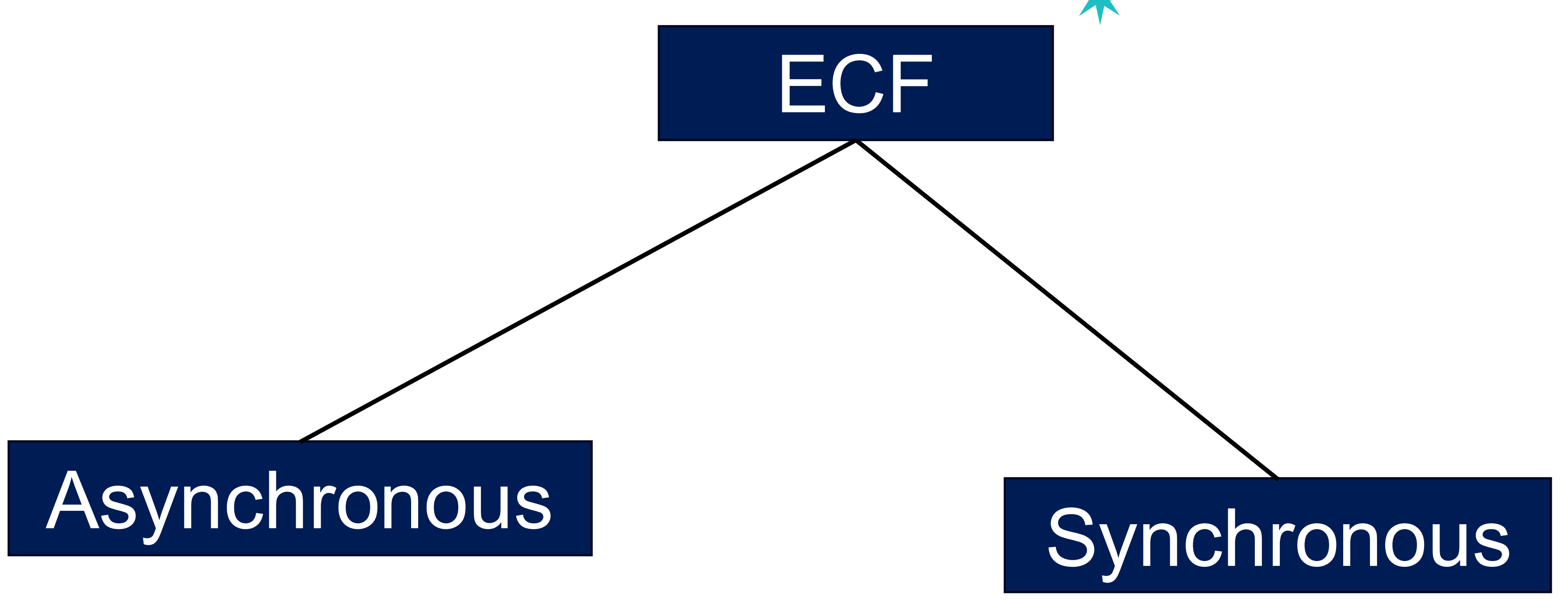


Exception Tables



- Each type of event has a unique exception number k
- k = index into exception table (a.k.a. interrupt vector)
- Handler k is called each time exception k occurs

Taxonomy of Exceptional Control Flow (ECF)



FYI: Synchronous vs. Asynchronous

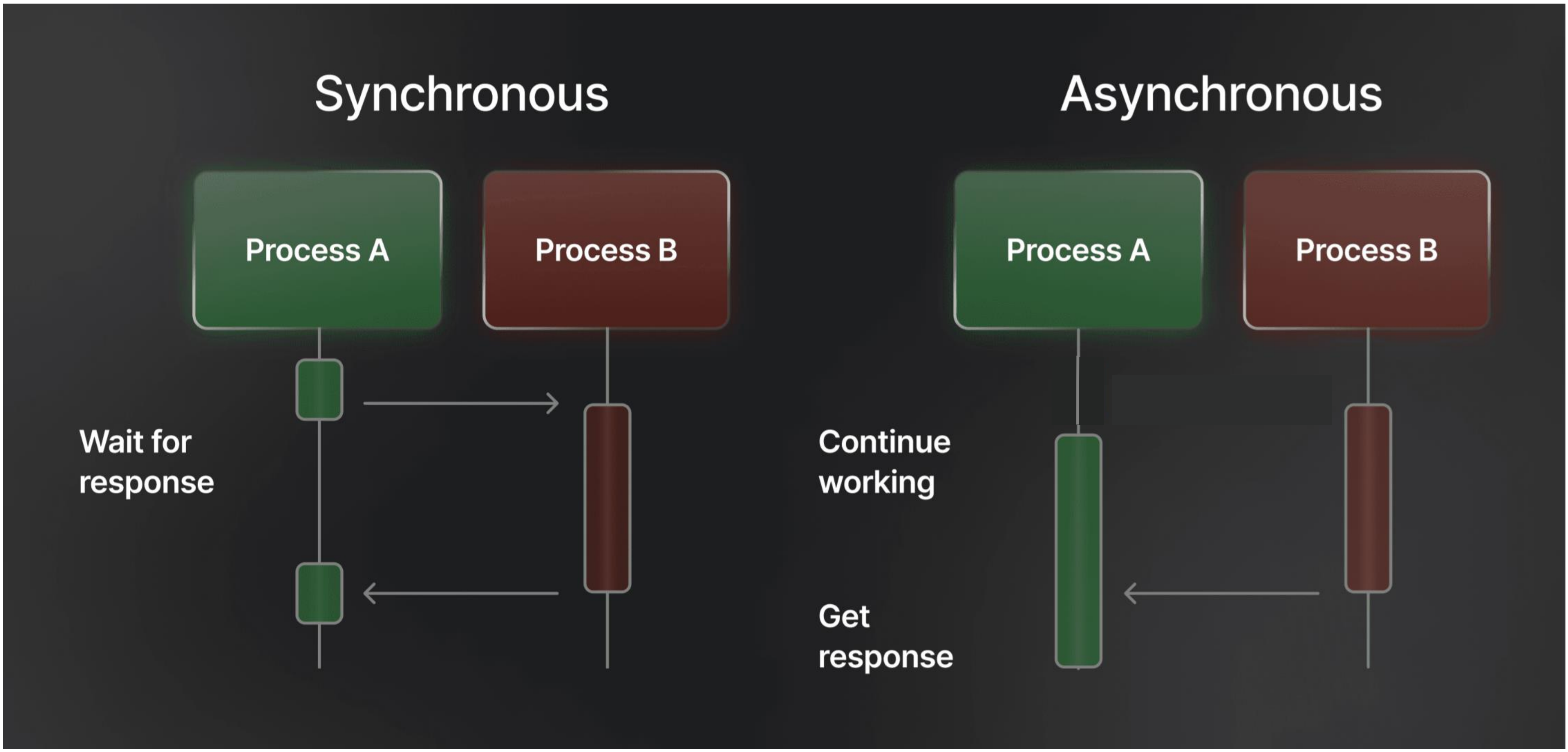
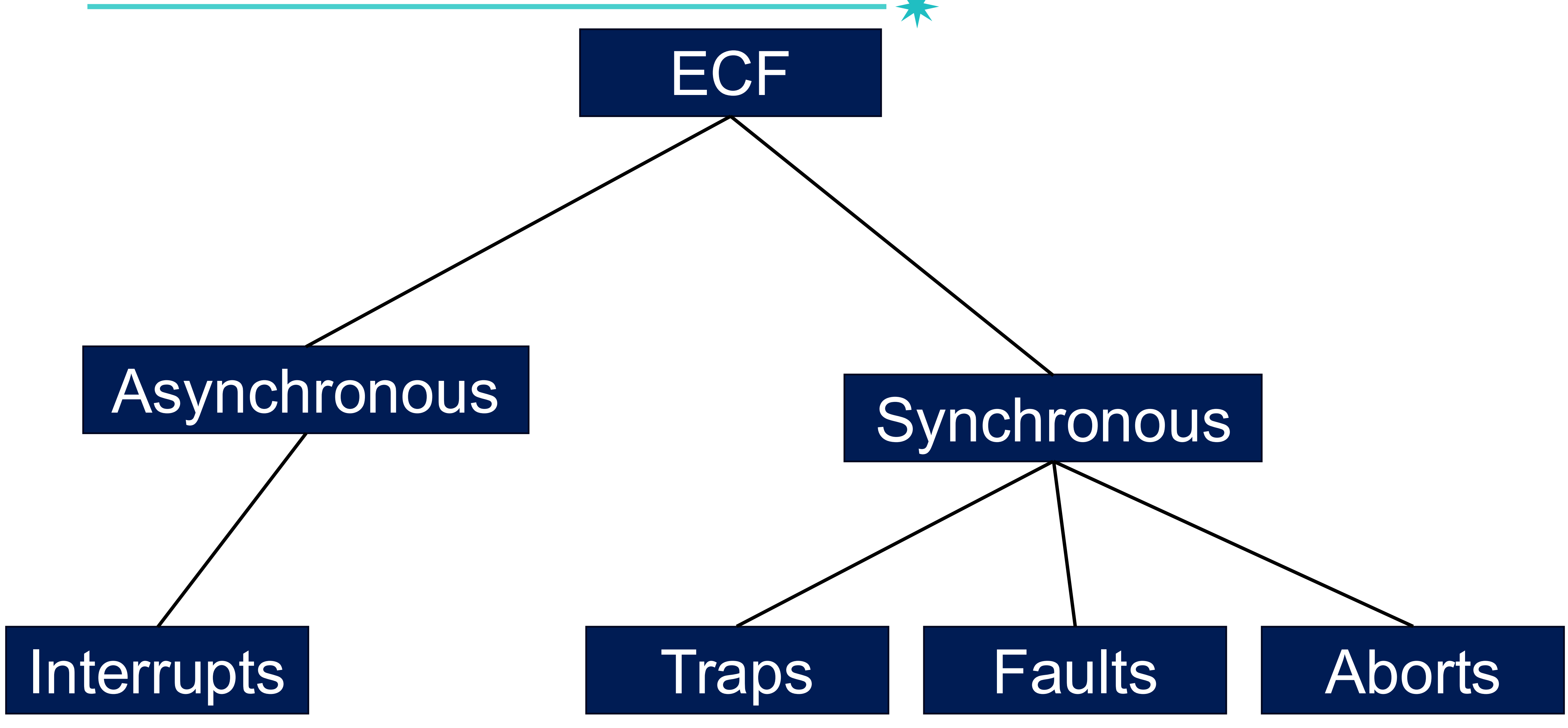


Image from https://www.koyeb.com/blog/introduction-to-synchronous-and-asynchronous-processing#executing-tasks-sync-versus-async

Taxonomy of Exceptional Control Flow (ECF)



Asynchronous Exceptions (Interrupts)

- Caused by events external to the processor
  - Indicated by setting the processor’s interrupt pin
  - Handler returns to “next” instruction

- **Examples:**
  - **Timer interrupt**
    - Every few ms, an external timer ship triggers an interrupt
    - Used by the kernel to take back control from user programs
      - Called “kernel preemption”
  - **I/O interrupt**
    - From keyboard: Hitting Ctrl-C at the keyboard
    - From NIC: Arrival of a packet from a network
    - From disk: Arrival of data from a disk

Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction

FYI: Synchronous vs. Asynchronous

Executing an instruction

Exception is occurred

Asynchronous

ocess A

ocess B

Process A

Process B

Wait for response

Continue working

Get response

Image from https://www.koyeb.com/blog/introduction-to-synchronous-and-asynchronous-processing#executing-tasks-sync-versus-async

Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - ***Traps***
    - Intentional, set program up to “trip the trap” and do something
    - Examples: ***system calls***, gdb breakpoints
    - Returns control to “next” instruction

Recall: syscall

Instruction	System Call Num.	Arguments	Return
`syscall`	`rax`	`rdi, rsi, rdx, r10, r8, r9`	`rax`

```
$ cat /usr/include/asm/unistd_64.h

#ifndef _ASM_UNISTD_64_H
#define _ASM_UNISTD_64_H

#define __NR_read 0
#define __NR_write 1
#define __NR_open 2
#define __NR_close 3
#define __NR_stat 4
#define __NR_fstat 5
...
```

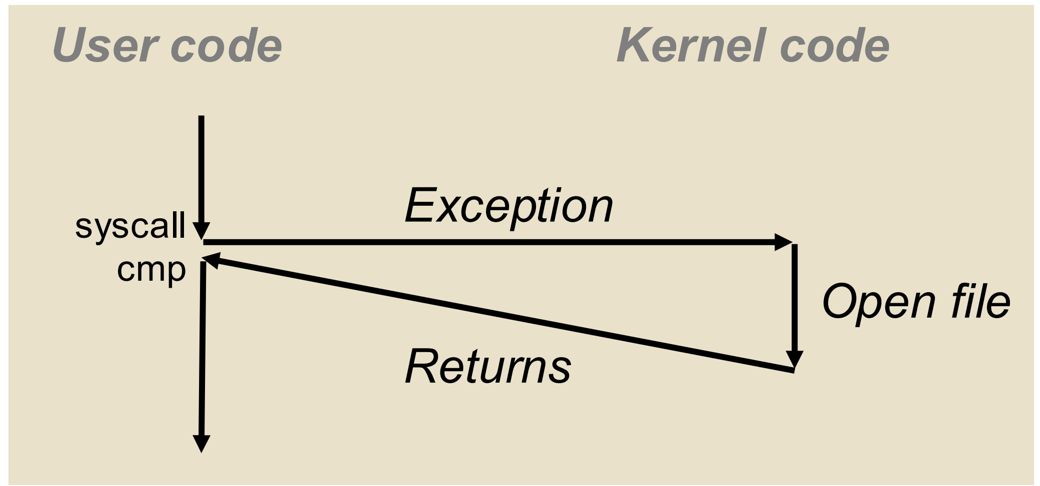
For example, the `printf()` function contains the following assembly code:

```
printf:
    ...
    mov  rax, 1
    syscall
    ...
```

System Call Example: Opening File

- User calls: `open(filename, options)`
- Calls __open function, which invokes system call

```
00000000000e5d70 <__open>:
...
e5d79:   b8 02 00 00 00      mov  eax, 0x2   # open is syscall #2
e5d7e:   0f 05               syscall         # Return value in rax
e5d80:  48 3d 01 f0 ff ff    cmp  rax, 0xfffffffffffff001
...
```



Synchronous Exceptions

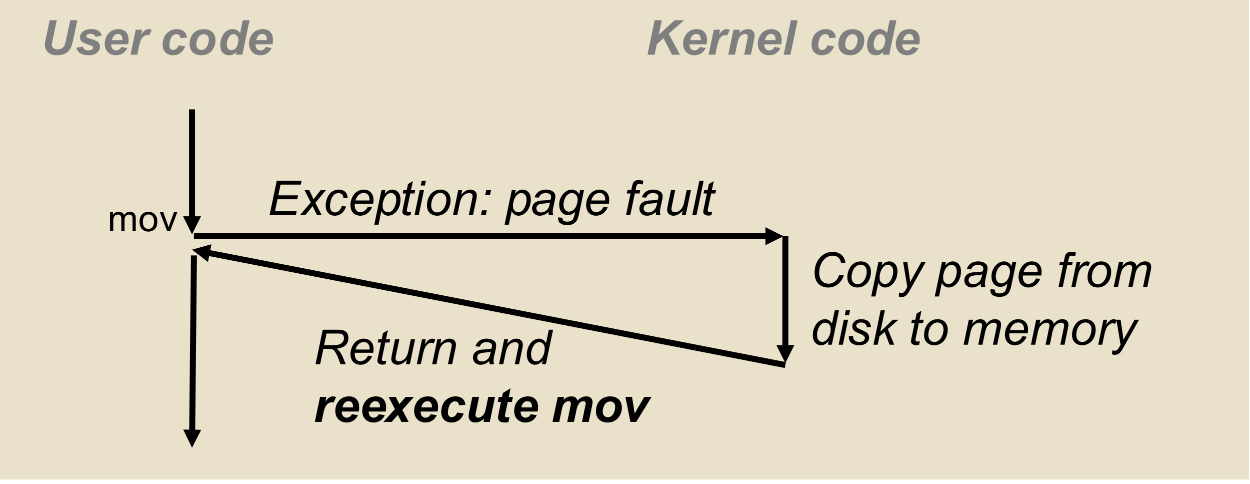
- Caused by events that occur as a result of executing an instruction:
  - *Traps*
    - Intentional, set program up to "trip the trap" and do something
    - Examples: ***system calls***, gdb breakpoints
    - Returns control to "next" instruction
  - *Faults*
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions
    - Either re-executes faulting ("current") instruction or aborts

Fault Example: Page Fault

```
int a[1000];
main ()
{
  a[500] = 13;
}
```

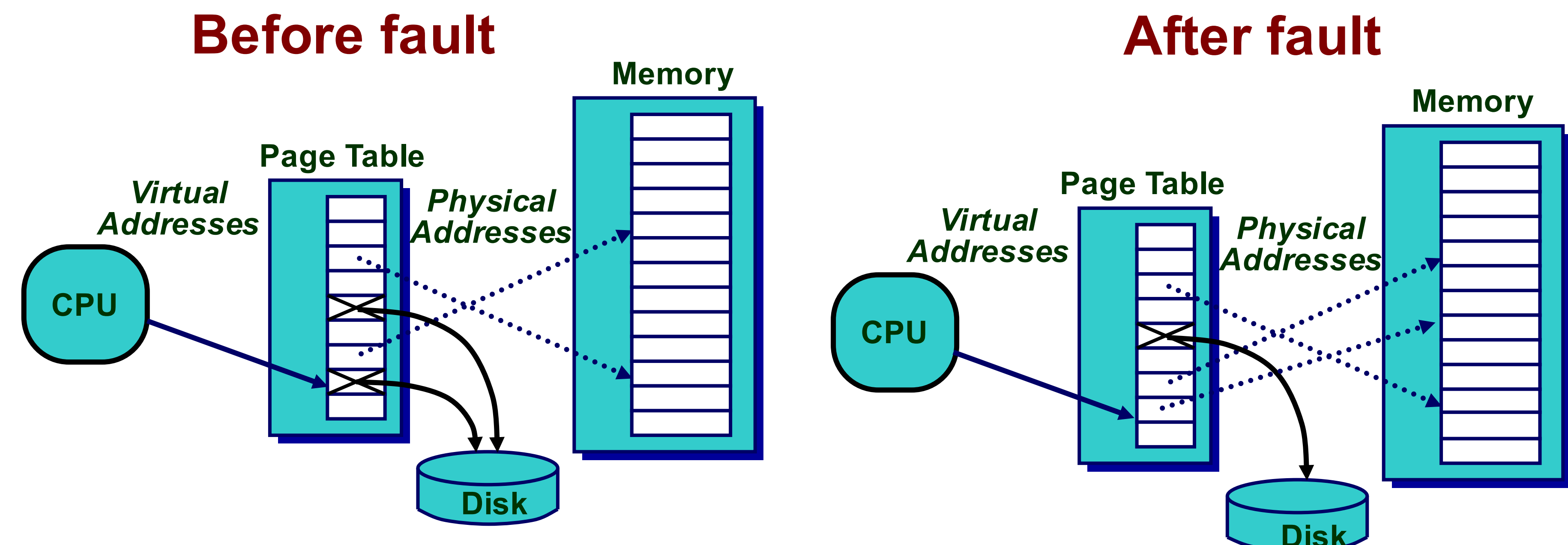
- User writes to memory location
- That portion (page) of user's memory is currently on disk

```
80483b7:      c7 05 10 9d 04 08 0d        mov    [0x8049d10], 0xd
```



Recap: Page Fault

- If a page is not in physical memory but disk (if valid == 0)
  - Page table entry (especially, valid bit) indicates that the page not in memory
  - Page fault exception is occurred! OS trap handler invoked to move data from disk into memory
    - OS has full control over placement!



Fault Example: Invalid Memory Reference

```
int a[1000];
main ()
{
  a[5000] = 13;
}
```

```
80483b7:      c7 05 60 e3 04 08 0d      mov    [0x804e360], 0xd
```

User code | **Kernel code**

mov → *Exception: page fault* → *Detect invalid address* → *Signal process*

- Sends `SIGSEGV` signal to user process (will be covered later)
- User process exits with “segmentation fault”

Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - ***Traps***
    - Intentional, set program up to “trip the trap” and do something
    - Examples: ***system calls***, gdb breakpoints
    - Returns control to “next” instruction
  - ***Faults***
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions
    - Either re-executes faulting (“current”) instruction or aborts
  - ***Aborts***
    - Unintentional and unrecoverable
    - Examples: illegal instruction, parity error, machine check
    - Aborts current program

Recap: Problem with Simple Shell

- Shell designed to run indefinitely
  - Should not accumulate unneeded resources
    - Memory
    - Child processes
    - File descriptors
- Our example shell correctly waits for and reaps foreground jobs
- But what about background jobs?
  - Will become zombies when they terminate
  - Will never be reaped because shell (typically) will not terminate
  - Will create a memory leak that could run the kernel out of memory
- How do we solve this problem?

Signal Comes to the Rescue!

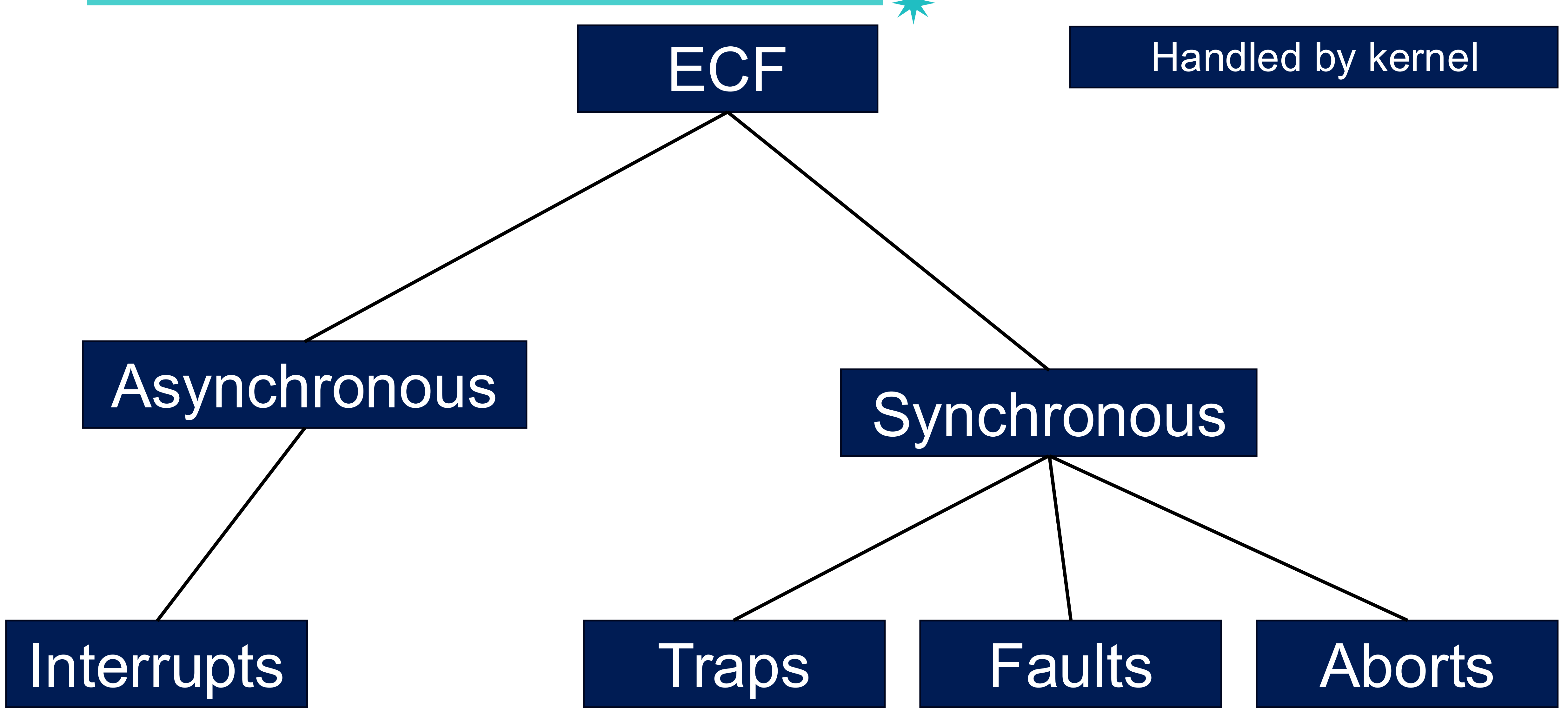
- Solution: exceptional control flow
  - **Idea:** the kernel will “signal” a designated process to alert us when a background process completes
  - In Unix, the alert mechanism is called a ***signal***

Signals

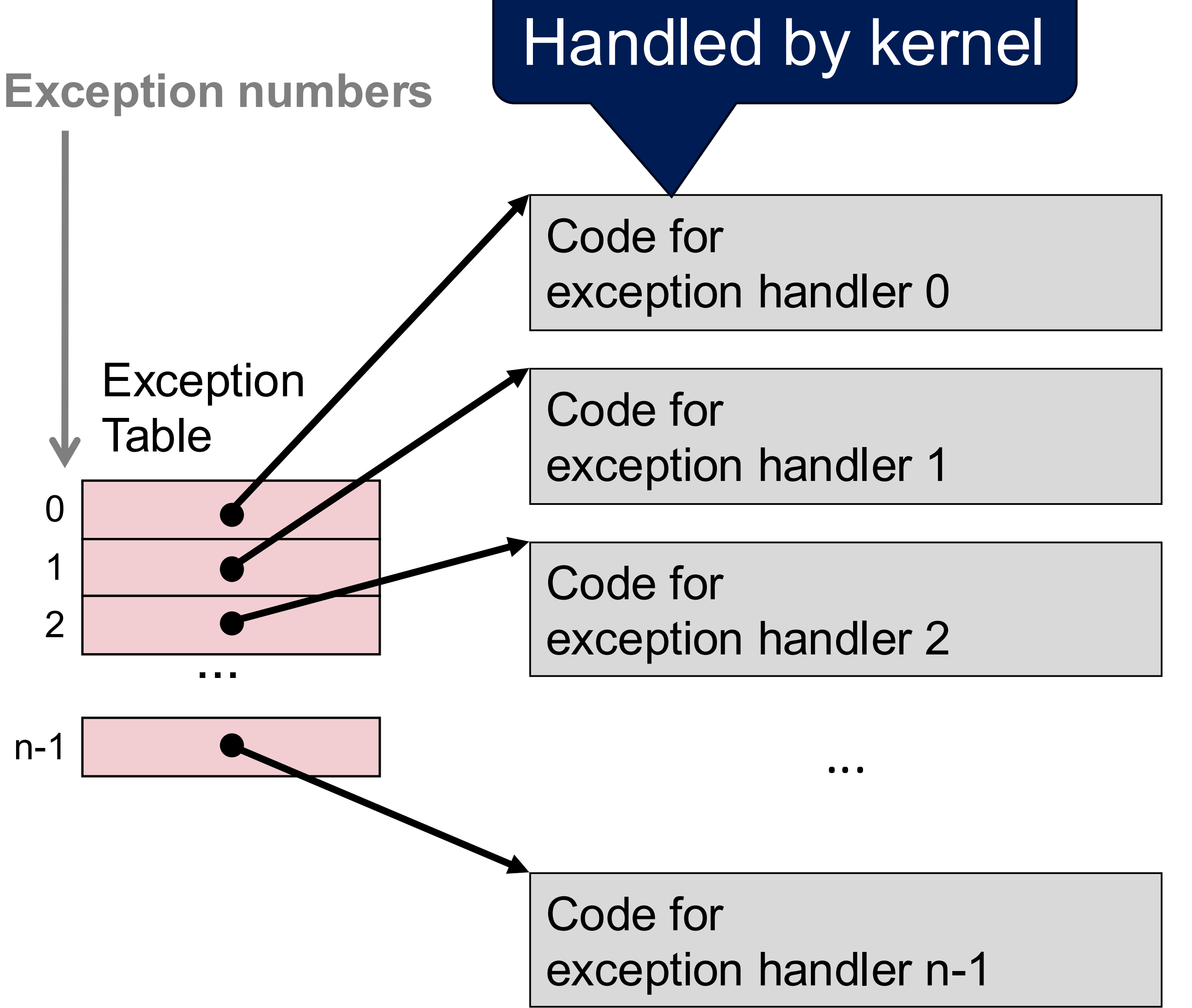
Signals

A *signal* is a small message that notifies a process that an event of some type has occurred in the system

Taxonomy of Exceptional Control Flow (ECF)

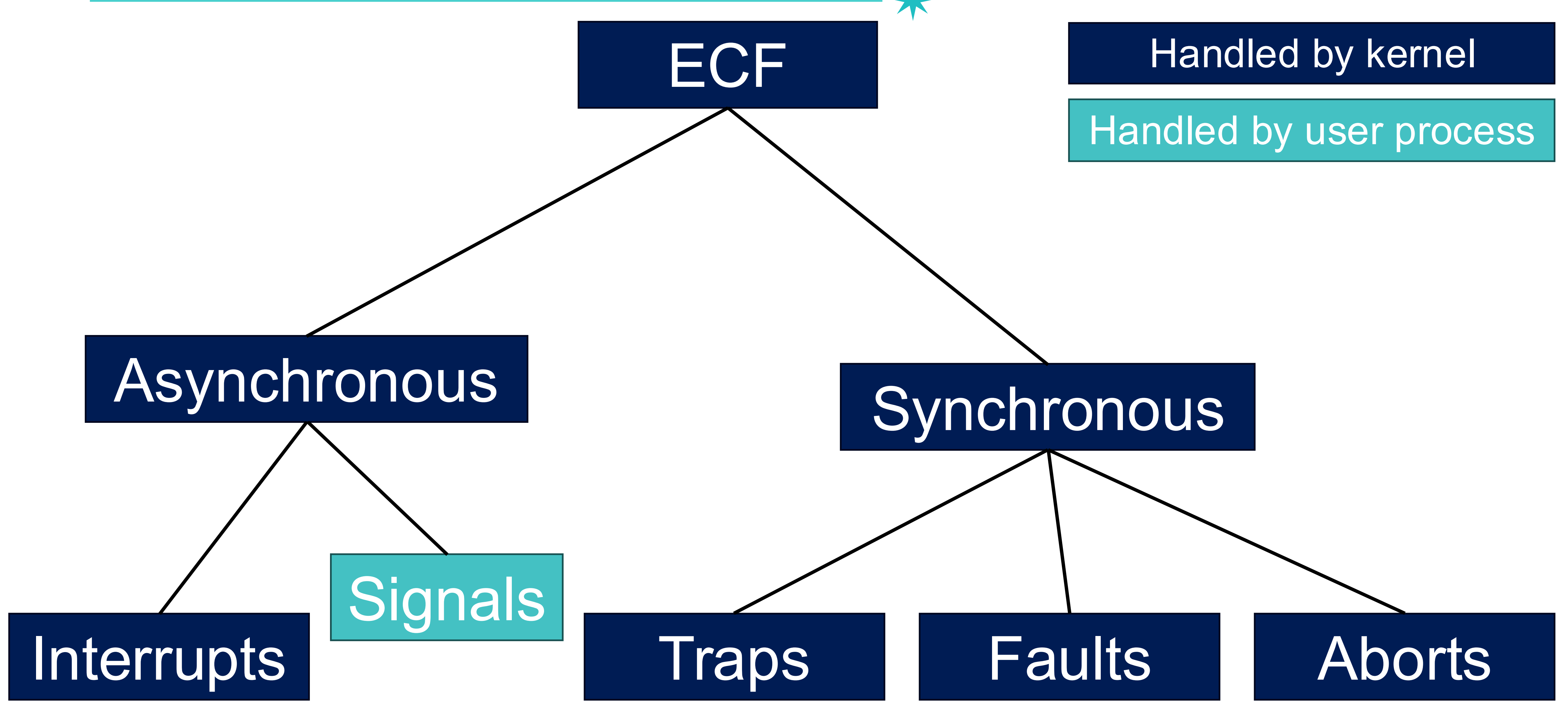


Recap: Exception Handlers Are in Kernel



- Each type of event has a unique exception number k
- k = index into exception table (a.k.a. interrupt vector)
- Handler k is called each time exception k occurs

Taxonomy of Exceptional Control Flow (ECF)



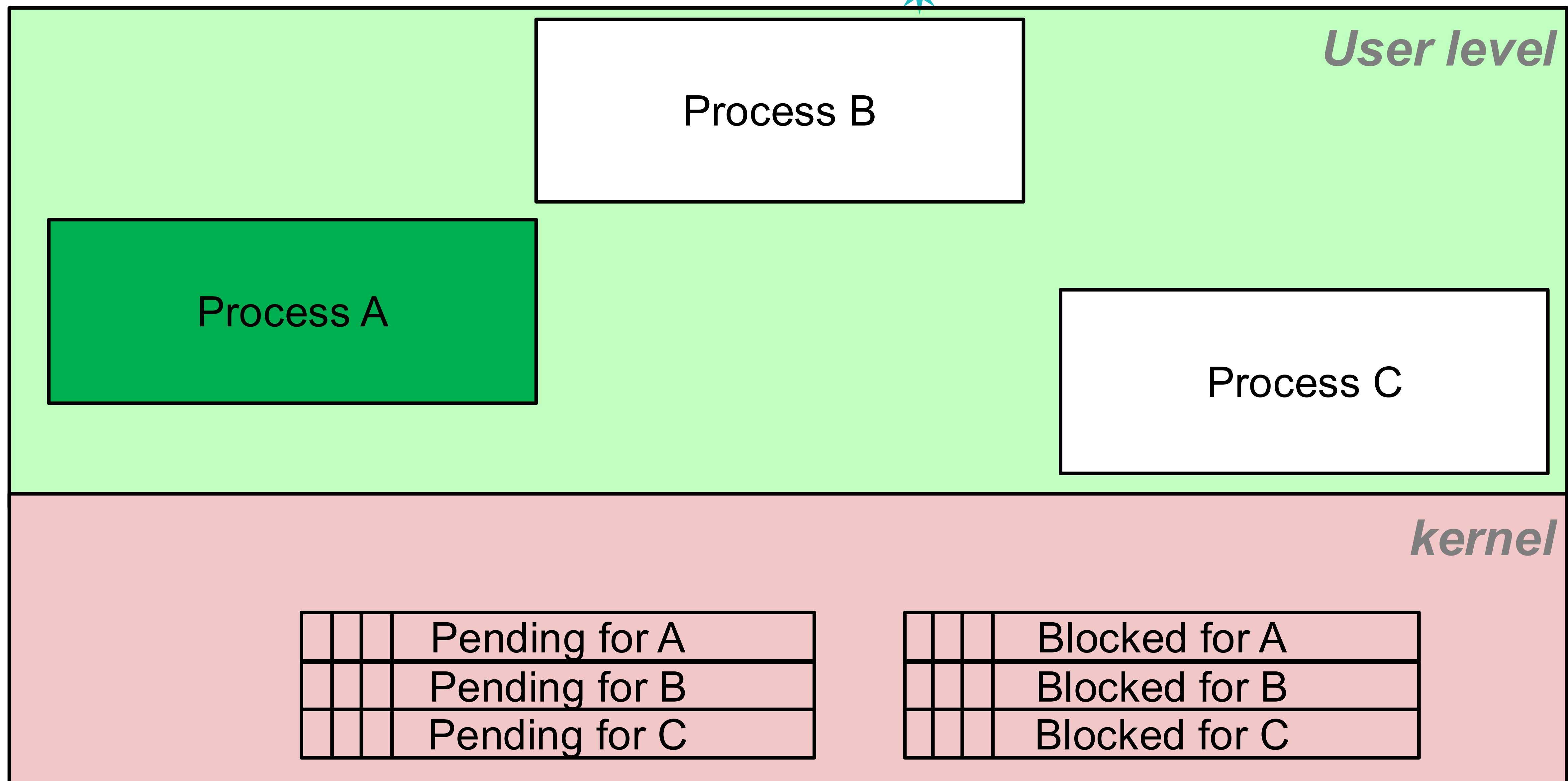
Signals

A *signal* is a small message that notifies a process that an event of some type has occurred in the system

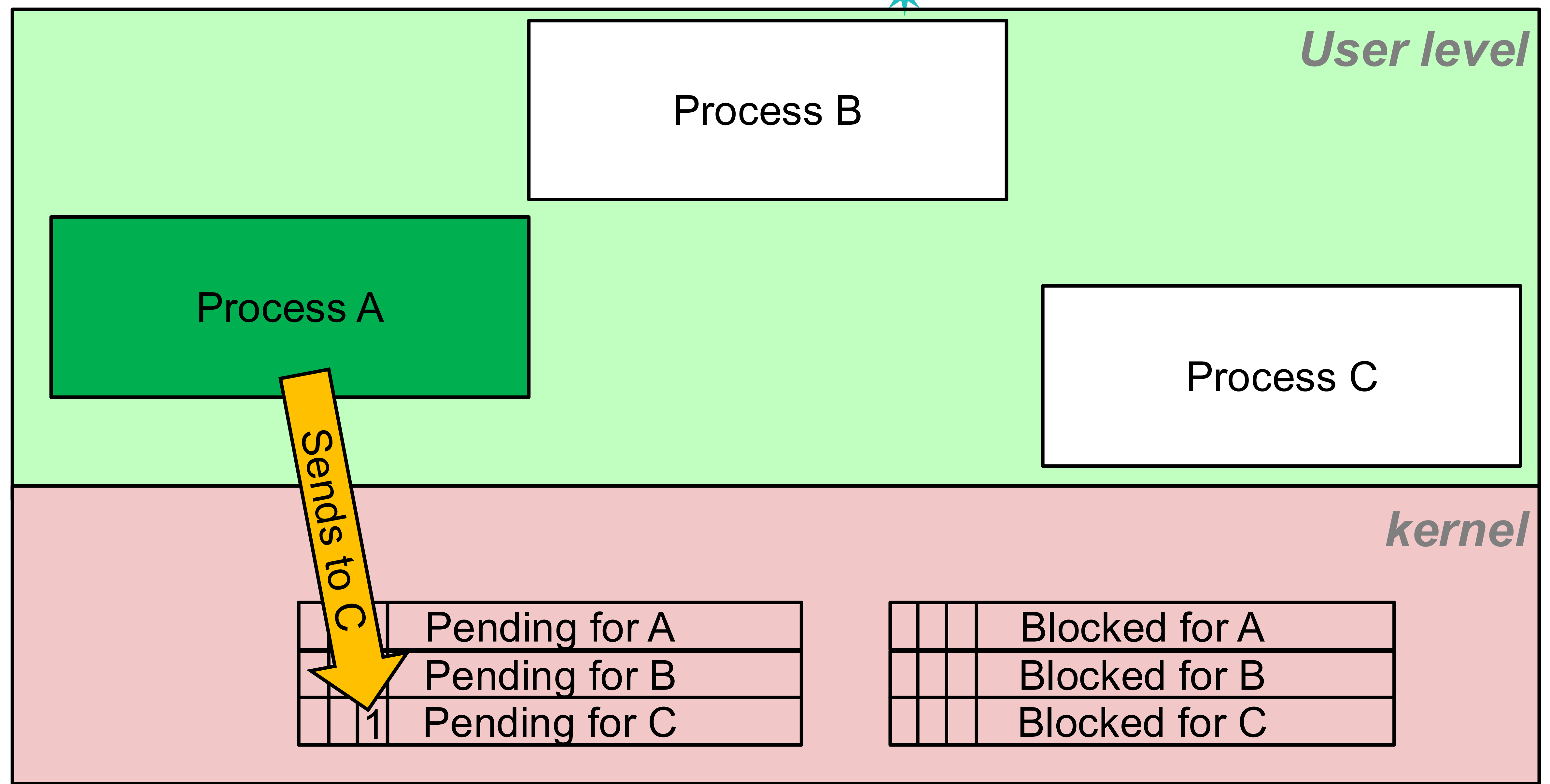
- Similar with exceptions and interrupts
- Sent from the kernel (sometimes at the request of another process) **to a process**
- Signal type is identified by small integer ID's (1-30)
- Only information in a signal is its ID and the fact that it arrived

ID	***Name***	***Default Action***	***Corresponding Event***
2	SIGINT	Terminate	User typed ctrl-c
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

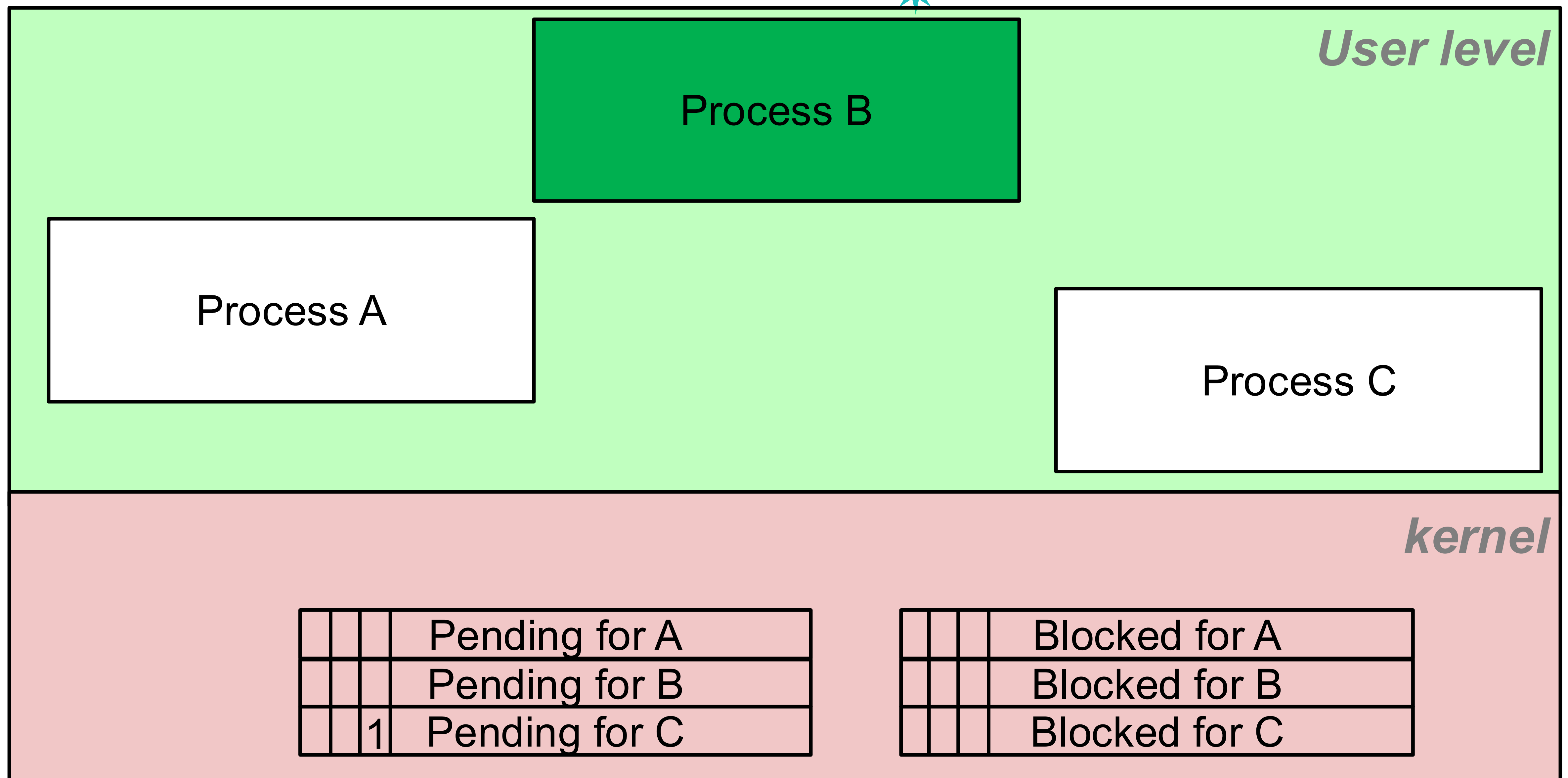
Signal Concepts: Sending a Signal



Signal Concepts: Sending a Signal



Signal Concepts: Sending a Signal



Signal Concepts: Receiving a Signal

User level

Process B

Process A

Process C

kernel

Received by C

			Pendin[illegible] A
			[illegible]ng for B
		1	Pending for C

			Blocked for A
			Blocked for B
			Blocked for C

Signal Concepts: Receiving a Signal

User level

Process B

Process A

Process C

kernel

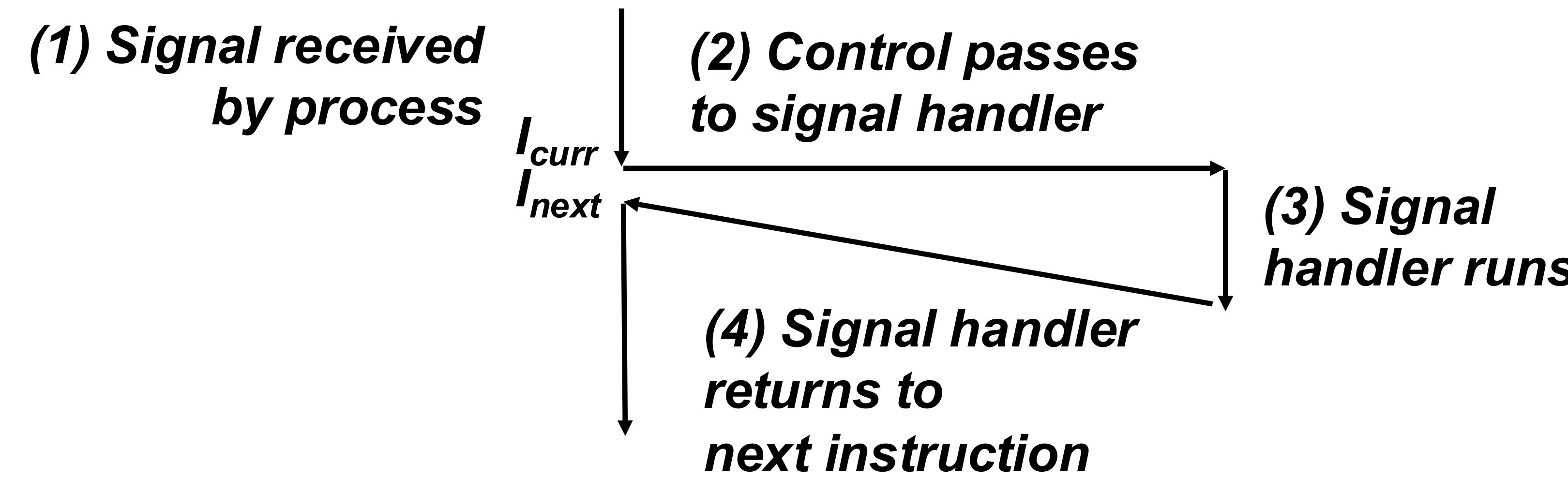
			Pending for A
			Pending for B
			Pending for C

			Blocked for A
			Blocked for B
			Blocked for C

Signal Concepts: Receiving a Signal

- A destination process *receives* a signal when it is forced by the kernel to react in some way to the signal

- Some possible ways to react:
  - ***Ignore*** the signal (do nothing)
  - ***Terminate*** the process (with optional core dump)
  - ***Catch*** the signal by executing a user-level function called ***signal handler***



Signal Concepts: Pending and Blocked Signals

- A signal is *pending* if sent but not yet received
  - There can be at most one pending signal of each type
  - Important: Signals are not queued
    - If a process has a pending signal of type k, then subsequent signals of type k that are sent to that process are discarded
- A process can *block* the receipt of certain signals
  - Blocked signals can be sent, but will not be received until the signal is unblocked
  - Some signals cannot be blocked (SIGKILL, SIGSTOP) or can only be blocked when sent by other processes (SIGSEGV, SIGILL, etc)
- A pending signal is received at most once

Sending Signals

Sending Signals with /bin/kill Program

- /bin/kill program sends arbitrary signal to a process or process group
- Examples

  $/bin/kill –9 24818
  Send SIGKILL to process 24818

```
linux> ps
  PID TTY          TIME CMD
24788 pts/2    00:00:00 tcsh
24818 pts/2    00:00:02 forks
24820 pts/2    00:00:00 ps

linux> /bin/kill -9 -24818

linux> ps
  PID TTY          TIME CMD
24788 pts/2    00:00:00 tcsh
24823 pts/2    00:00:00 ps
```

Sending Signals using `kill` Function

- `int kill(pid_t pid, int sig)`
  - Send a specific signal to a process or process group
  - Pass standard signals via `sig` argument
    - `SIGTERM`: The polite, default termination
    - `SIGKILL`: The forced, immediate termination
    - …

Sending Signals from the Keyboard

- SIGINT (ctrl-c): default action is to terminate each process
- SIGTSTP (ctrl-z): default action is to stop (suspend) each process

Example of ctrl-c and ctrl-z


```
linux> ./infinite-loop
<types ctrl-z>
Suspended

linux> ps w
  PID TTY      STAT   TIME COMMAND
27699 pts/8    Ss     0:00 -tcsh
28108 pts/8    T      0:01 ./infinite-loop
28109 pts/8    R+     0:00 ps w

linux> fg
./infinite-loop
<types ctrl-c>

linux> ps w
  PID TTY      STAT   TIME COMMAND
27699 pts/8    Ss     0:00 -tcsh
28110 pts/8    R+     0:00 ps w
```

- S: sleeping
- T: stopped
- R: running

Move a background or suspended job into the foreground

Receiving Signals

Receiving Signals

- The signal is delivered when kernel is about to pass control to process *p*

Process q | *Process p*

Time

user code

kernel code } *context switch*

user code

kernel code } *context switch*

user code

Default Actions

- By default, a program that receives a signal takes the default action
- Each signal type has a predefined *default action*, which is one of:
  - The process terminates
  - The process stops until restarted by a SIGCONT signal
  - The process ignores the signal

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	User typed ctrl-c
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

Motivation of Custom Signal Handlers

Q. What if a process wants to perform an action other than the default action when it receives a signal?

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	User typed ctrl-c
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

Installing Signal Handlers

- The **`signal`** function modifies the default action associated with the receipt of signal **`signum`**:

```
handler_t *signal(int signum, handler_t *handler)
```

- Different values for **`handler`**:
  - SIG_IGN: ignore signals of type **`signum`**
  - SIG_DFL: revert to the default action on receipt of signals of type **`signum`**
  - Otherwise, handler is **the address of a user-level signal handler (function)**
    - Called when process receives signal of type **`signum`**
    - Executing handler is called ***"catching"*** or ***"handling"*** the signal

Signal Handling Example


```
#include<unistd.h>
#include<signal.h>

void sigint_handler(int sig) { /* SIGINT handler */
    printf("So you think you can stop the bomb with ctrl-c, do you?\n");
    sleep(2);
    printf("Well...");
    fflush(stdout);
    sleep(1);
    printf("OK. :-)\n");
    exit(0);
}

int main(int argc, char** argv) {
    signal(SIGINT, sigint_handler); /* Install the SIGINT handler */
    while(1) {} /* Wait for the receipt of a signal */
    return 0;
}
```

Q. What would happen if you keep typing “ctrl-c”?
(It is highly recommended to run this code!)

Signal Handling Example

```c
#include<unistd.h>
#include<signal.h>

void sigint_handler(int sig) { /* SIGINT handler */
    printf("So you think you can stop the bomb with ctrl-c, do you?\n");
    sleep(2);
    printf("Well...");
    fflush(stdout);
    sleep(1);
    printf("OK. :-)\n");
    exit(0);
}

int main(int argc, char** argv) {
    signal(SIGINT, sigint_handler); /* Install the SIGINT handler */
    while(1) {} /* Wait for the receipt of a signal */
    return 0;
}
```

Signals Handlers as Concurrent Flows

Recap: Problem with Simple Shell

- Shell designed to run indefinitely
  - Should not accumulate unneeded resources
    - Memory
    - Child processes
    - File descriptors
- Our example shell correctly waits for and reaps foreground jobs
- But what about background jobs?
  - Will become zombies when they terminate
  - Will never be reaped because shell (typically) will not terminate
  - Will create a memory leak that could run the kernel out of memory
- How do we solve this problem?

Install a handler for `SIGCHLD` so that the parent process, i.e., the shell, calls wait concurrently only when it catches `SIGCHLD`

Safe Signal Handling

- Handlers are tricky because they are concurrent with main program and share the same global data structures
  - Shared data structures can become corrupted

- For now here are some guidelines to help you avoid trouble

Guidelines for Writing Safe Handlers

- **G0: Keep your handlers as simple as possible**
  - e.g., set a global flag and return
- **G1: Call only async-signal-safe functions in your handlers**
  - `printf`, `sprintf`, `malloc`, and `exit` are not safe!
- **G2: Save and restore `errno` on entry and exit**
  - So that other handlers don't overwrite your value of `errno`
- **G3: Protect accesses to shared data structures by temporarily blocking all signals**
  - To prevent possible corruption
- **G4: Declare global variables as `volatile`**
  - To prevent compiler from storing them in a register
- **G5: Declare global flags as volatile `sig_atomic_t`**
  - flag: variable that is only read or written (e.g. flag = 1, not flag++)
  - Flag declared this way does not need to be protected like other globals
- **Check the textbook for more details!**

Summary

- Signals provide process-level exception handling
  - Can generate from user programs
  - Can define affect by declaring signal handler
  - Be very careful when writing signal handlers

Question?