Review

- First, operating systems solves time-sharing multi-tasking
  - context = memory address space + stack pointer + instruction pointer
  - IBM360 uses context-switch for time-sharing multi-tasking
- Second, operating systems solves interprocess communication (IPC)
  - AT&T UNIX V provides message queue, shared memory and semaphore
- Third, operating systems handles exception control flow (today's lecture).

Exception Control Flow (ECF)

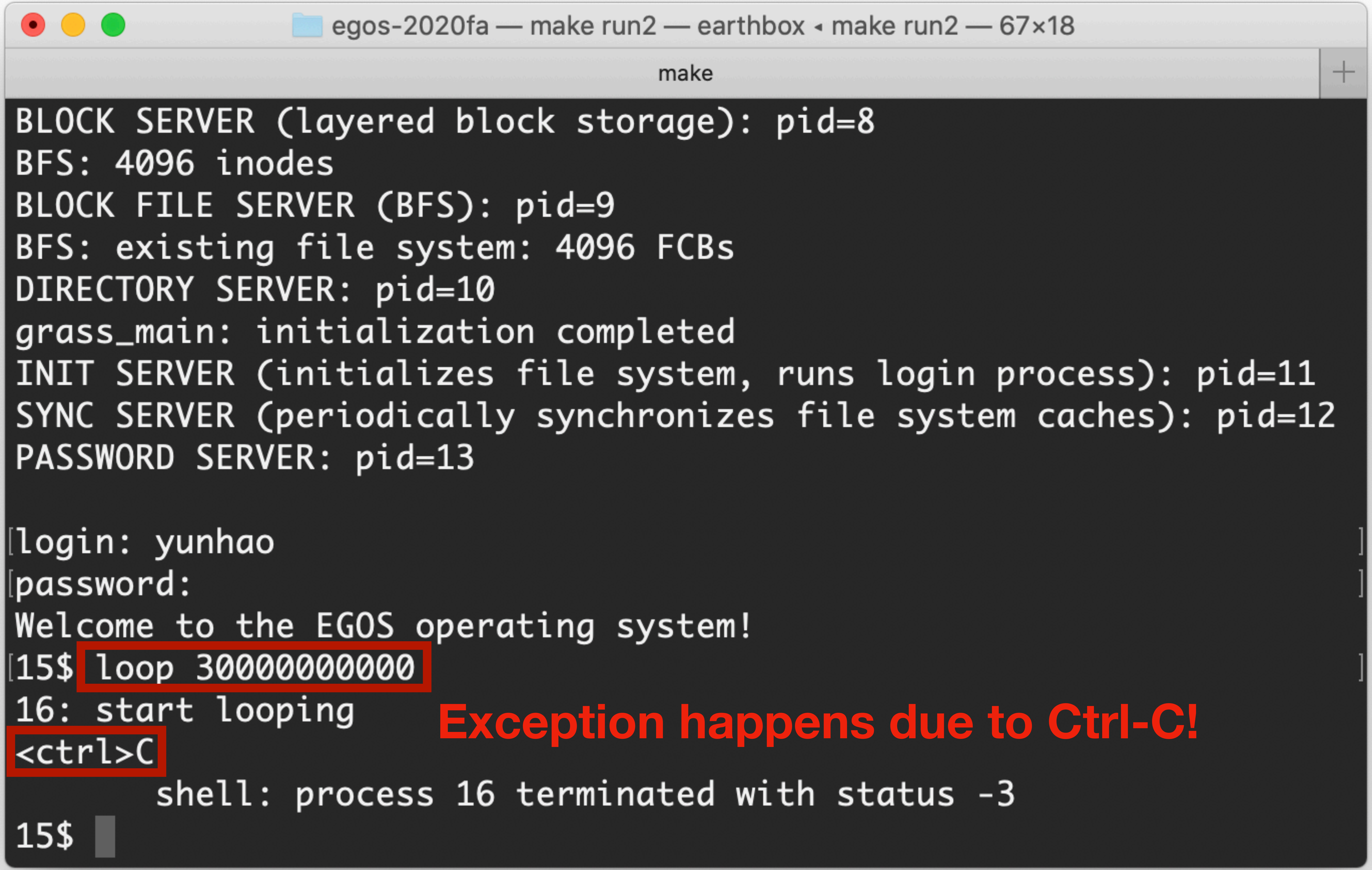
tmp — yunhao@YunhaodeMacBook-Pro — -zsh — 67×18

~/tmp

```
➜  tmp cat tmp.c
#include <stdio.h>

int main() {
    int a = 8, b = 2;
    printf("%d / %d = %d\n", a, b, a/b);
    b = 0;
    printf("%d / %d = %d\n", a, b, a/b);
    return 0;
}
➜  tmp gcc tmp.c
➜  tmp ./a.out
8 / 2 = 4
[1]    24859 floating point exception  ./a.out
➜  tmp
```

Exception happens due to divide 0!


egos-2020fa — make run2 — earthbox ◂ make run2 — 67×18
make
BLOCK SERVER (layered block storage): pid=8
BFS: 4096 inodes
BLOCK FILE SERVER (BFS): pid=9
BFS: existing file system: 4096 FCBs
DIRECTORY SERVER: pid=10
grass_main: initialization completed
INIT SERVER (initializes file system, runs login process): pid=11
SYNC SERVER (periodically synchronizes file system caches): pid=12
PASSWORD SERVER: pid=13
login: yunhao
password:
Welcome to the EGOS operating system!
15$ loop 30000000000
16: start looping
<ctrl>C
shell: process 16 terminated with status -3
15$
Exception happens due to Ctrl-C!

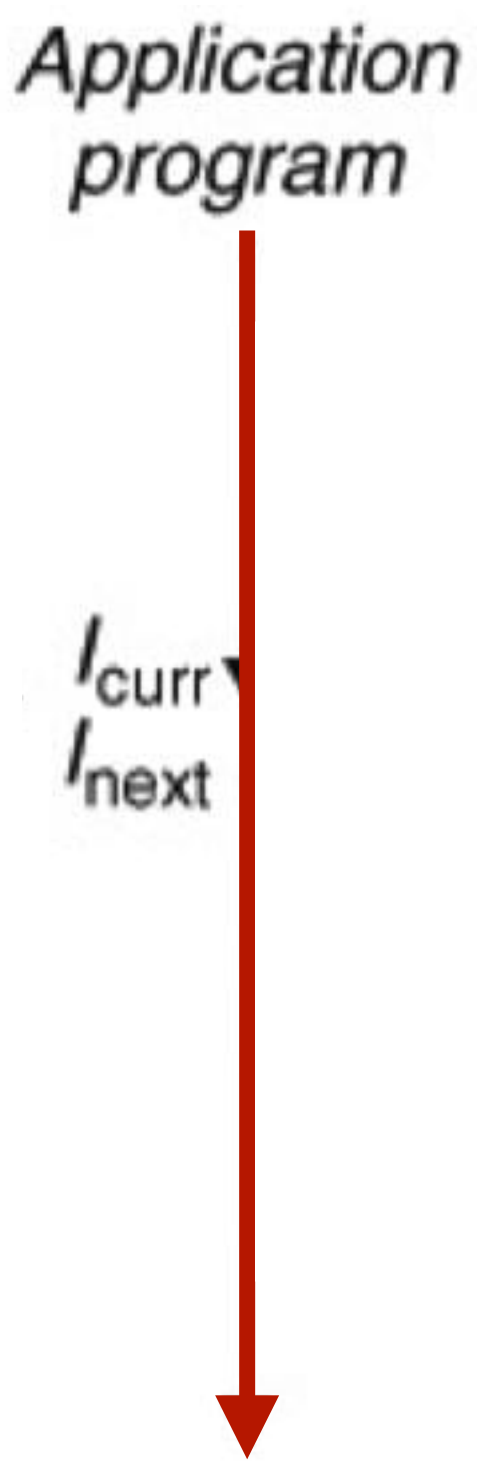
More examples of exception

Who **initiates** exception?	Who **handles** exception?	Examples
CPU / Hardware	Operating System	Timer interrupt, I/O interrupt
User Application	Operating System	Divide zero, Ctrl-C interrupt, kill a process
User Application	User Application	Try-catch in C++ or Java

Control flow is the sequence of instructions executed by one CPU.

CPU executes instructions sequentially: $I_1, I_2, I_3, I_4, \ldots.$

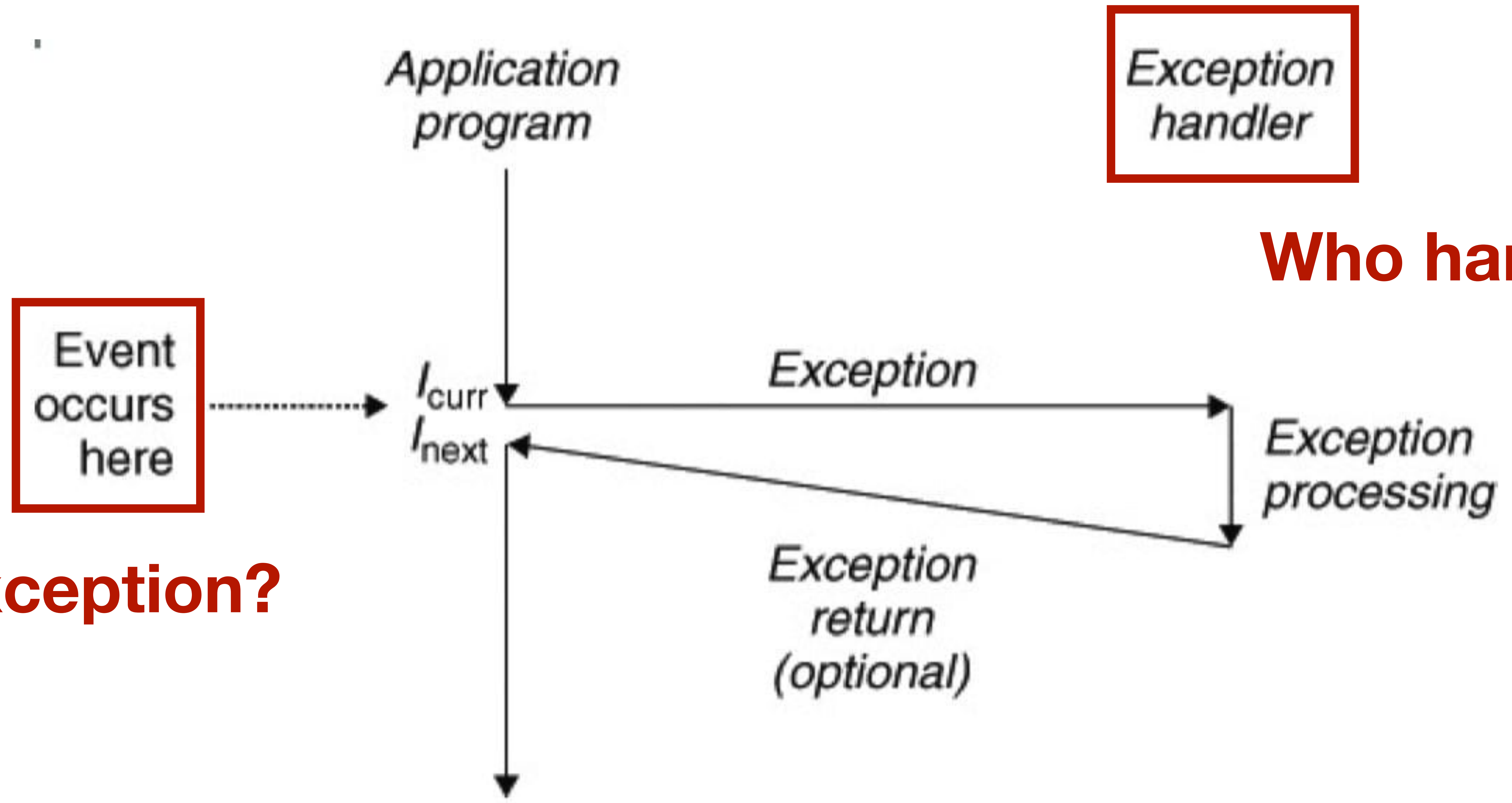
Normal control flow



I_{curr} **is the current CPU instruction,** I_{next} **is the expected next CPU instruction**

*** Image from CSAPP: Computer Systems A Programmer's Perspective**

General picture of exception control flow



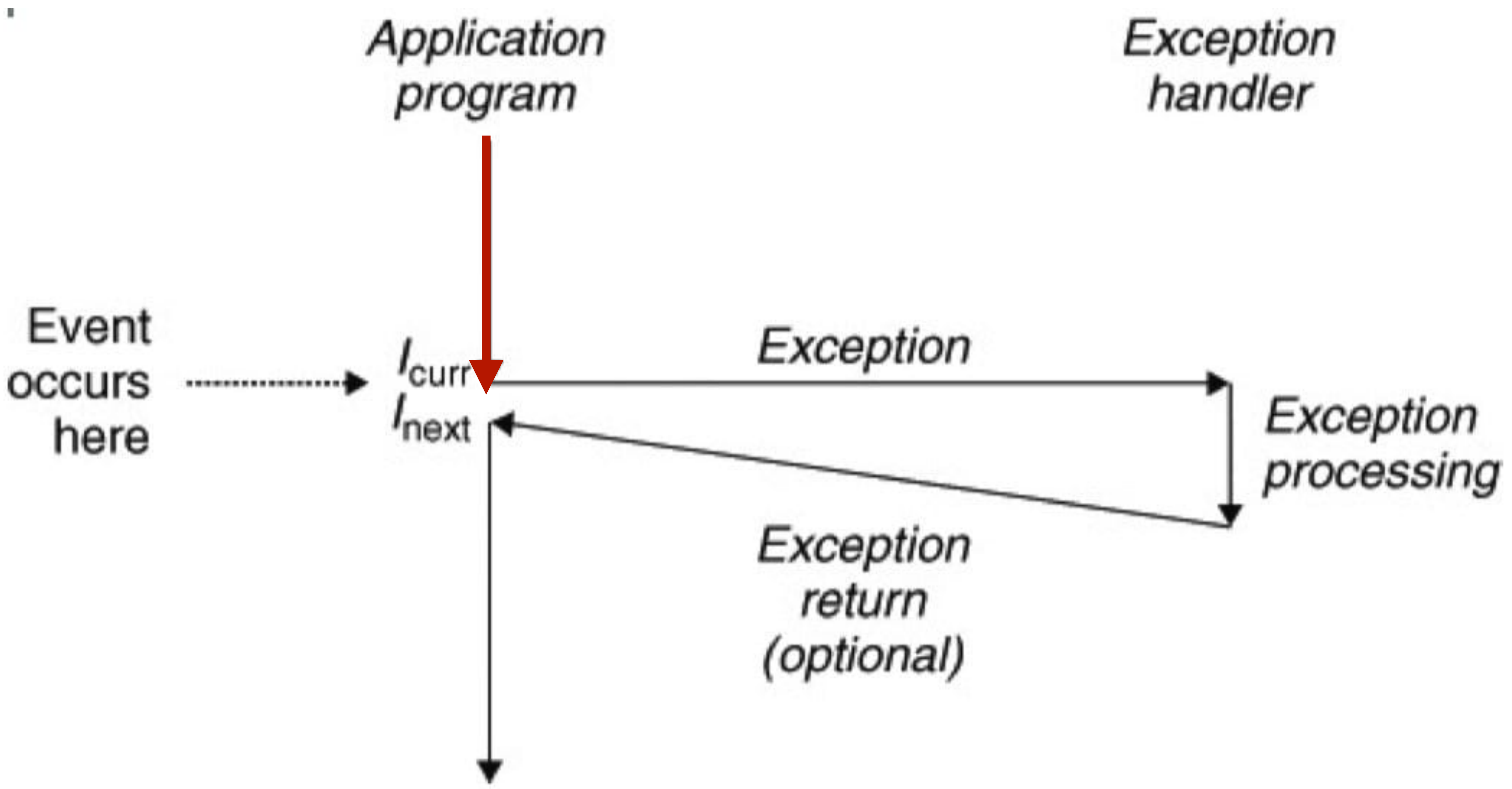
Who handles exception?

Who initiates exception?

Key of ECF: an event occurs between I_{curr} and I_{next} !

* Image from CSAPP: Computer Systems A Programmer's Perspective

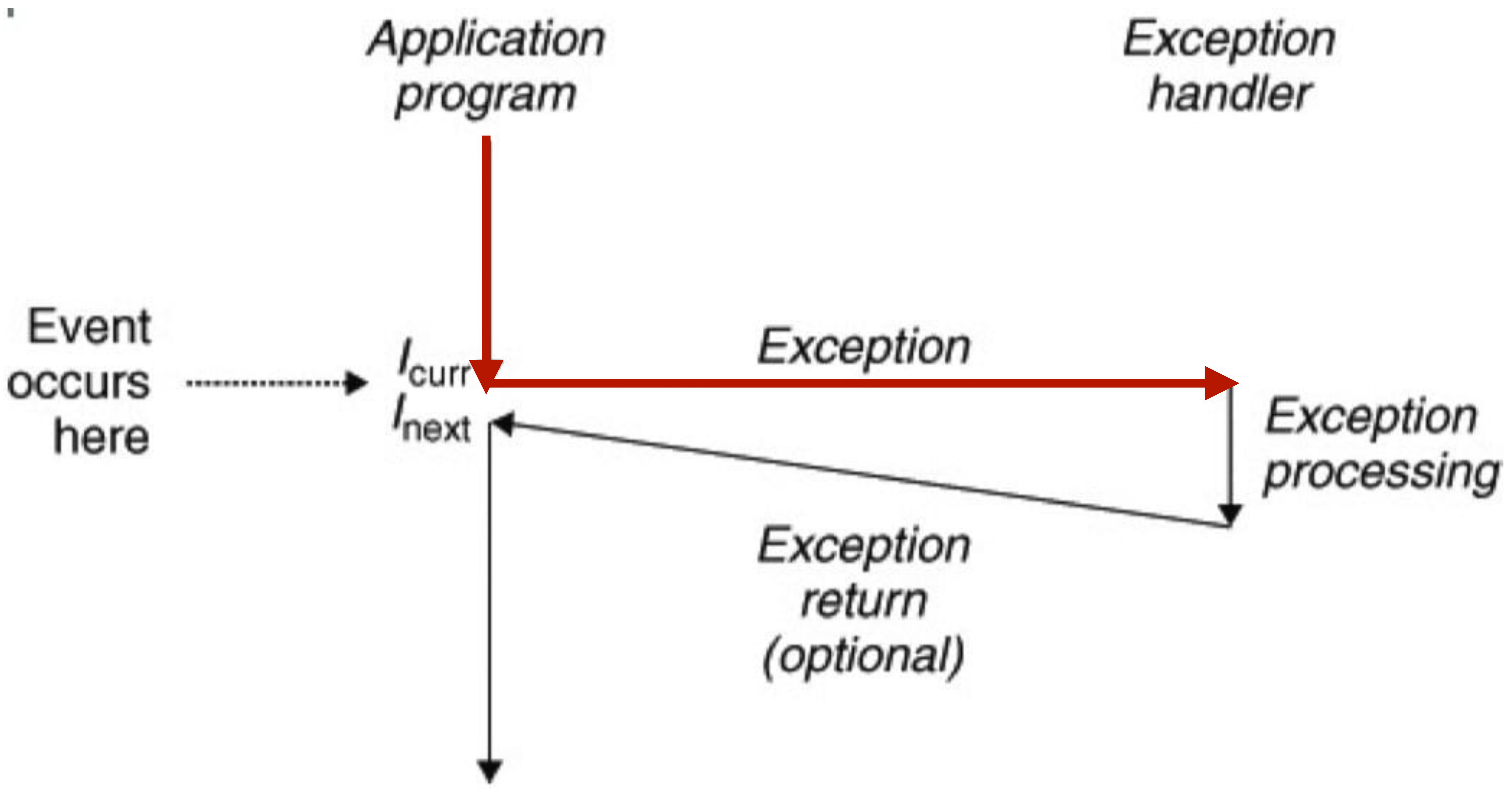
Step1: CPU executes normally till I_{curr}



I_{curr} is the current CPU instruction, I_{next} is the expected next CPU instruction

*** Image from CSAPP: Computer Systems A Programmer's Perspective**

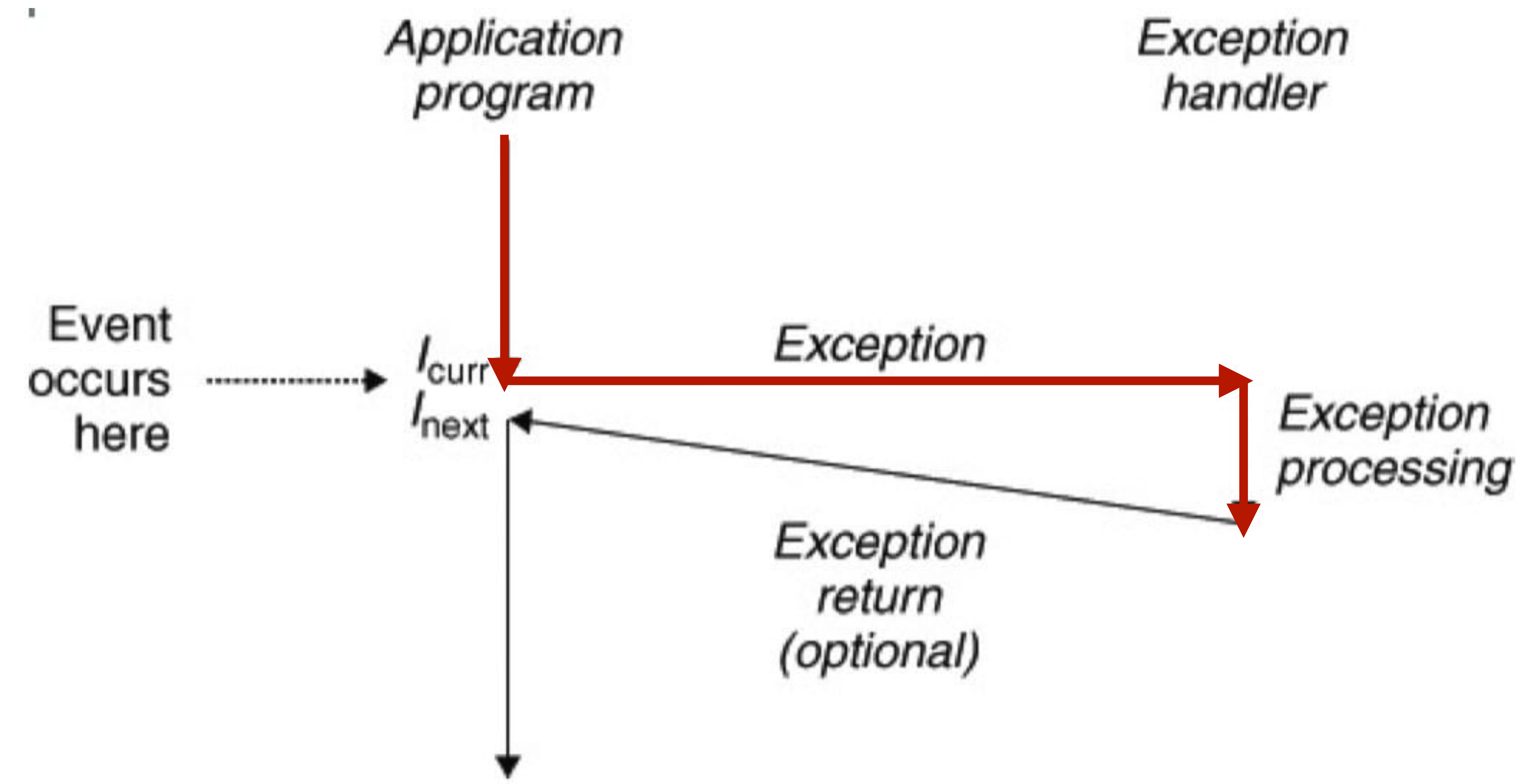
Step2: an exception is initiated at I_{curr}



I_{curr} is the current CPU instruction, I_{next} is the expected next CPU instruction

*** Image from CSAPP: Computer Systems A Programmer's Perspective**

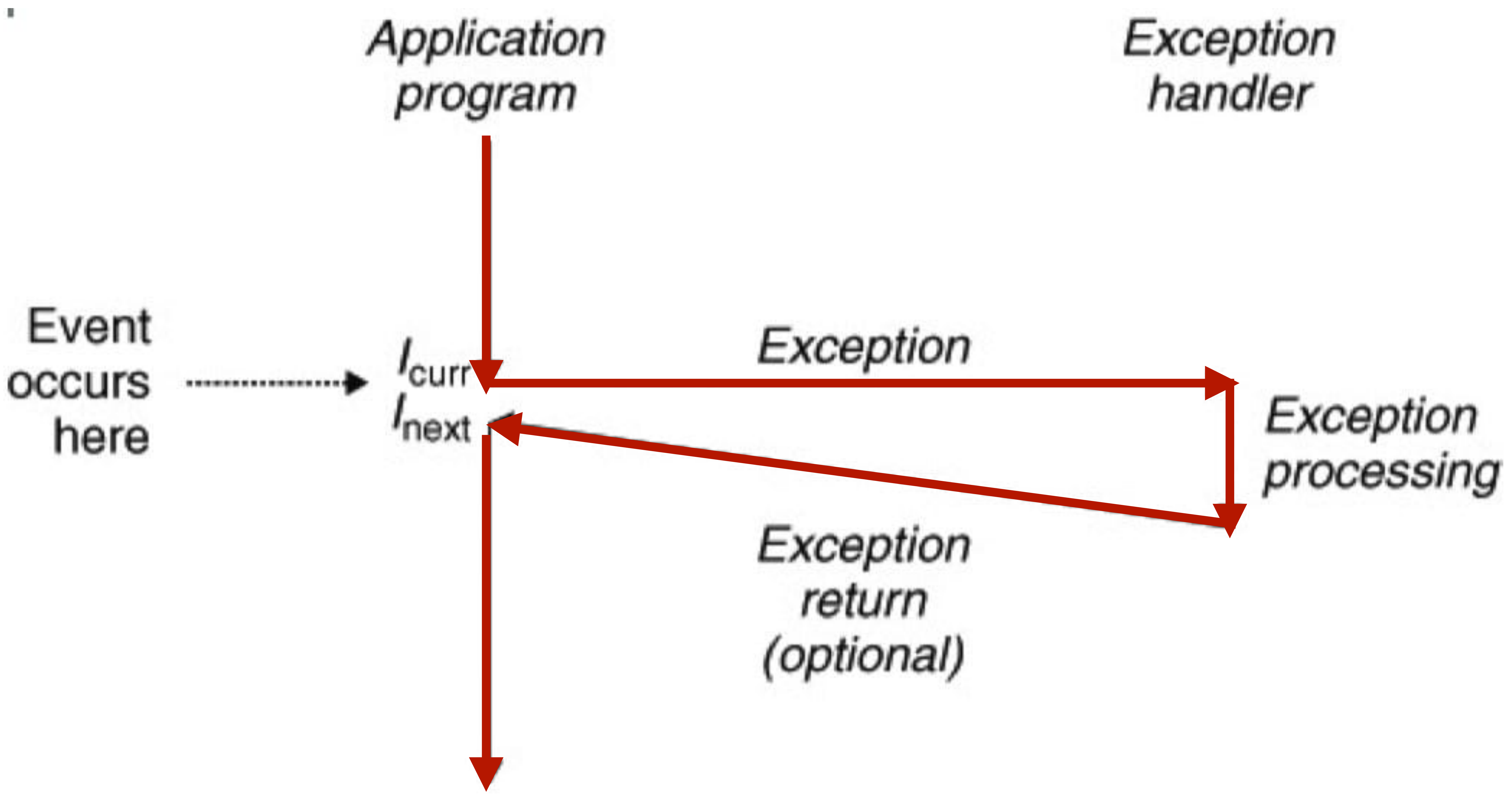
Step3: exception is being handled



I_{curr} **is the current CPU instruction,** I_{next} **is the expected next CPU instruction**

*** Image from CSAPP: Computer Systems A Programmer's Perspective**

Step4: CPU (may) switch back to I_{next}



I_{curr} **is the current CPU instruction,** I_{next} **is the expected next CPU instruction**

*** Image from CSAPP: Computer Systems A Programmer's Perspective**

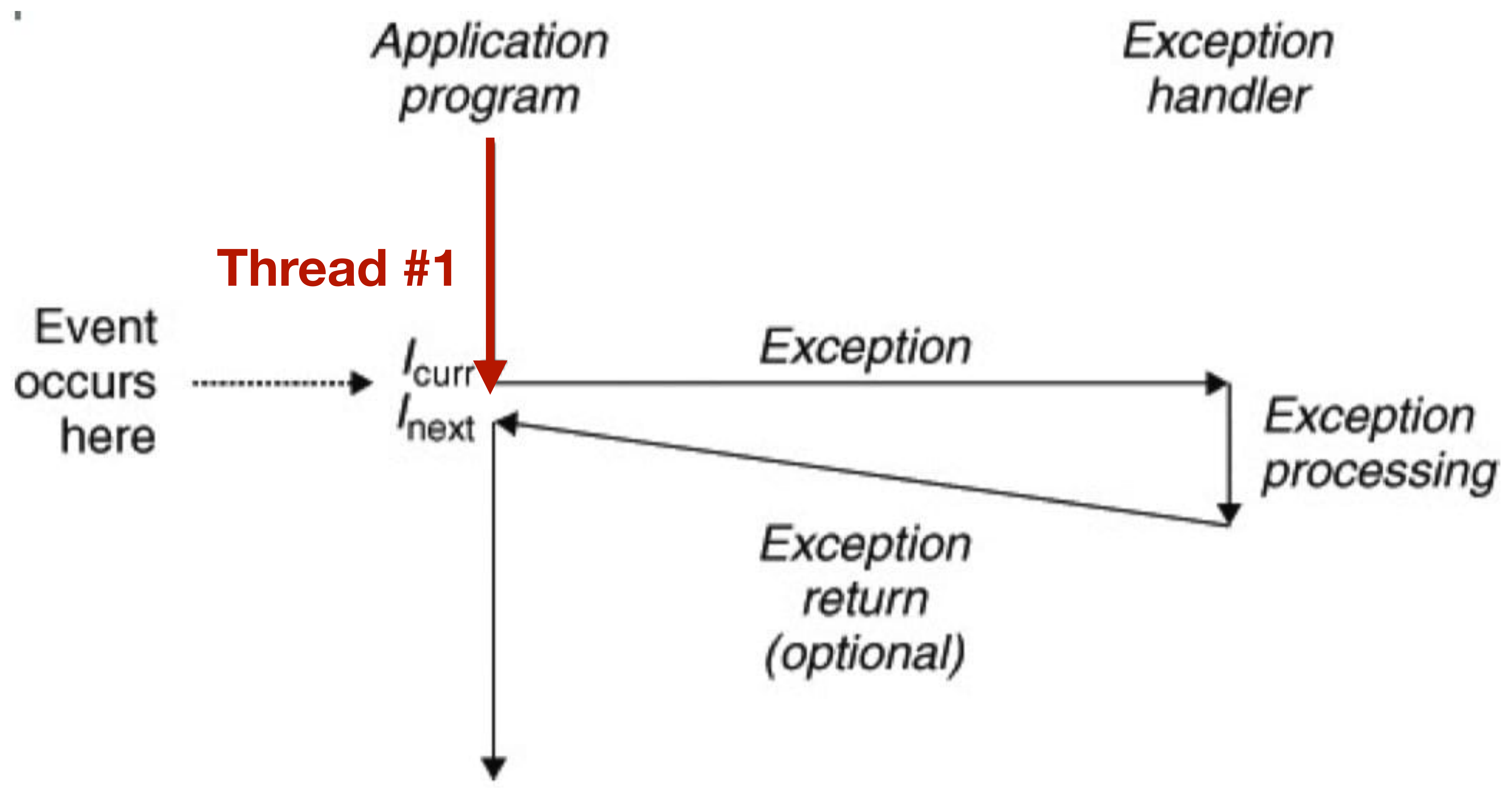
General steps of exception control flow

- Step1: CPU executes normally (normal control flow).
- Step2: An event occurs between I_{curr} and I_{next}, the CPU control flow transfers to an exception handler.
- Step3: Exception is being handled.
- Step4: CPU may switch control flow back to I_{next}

Exception control flow enables preemptive context-switch.

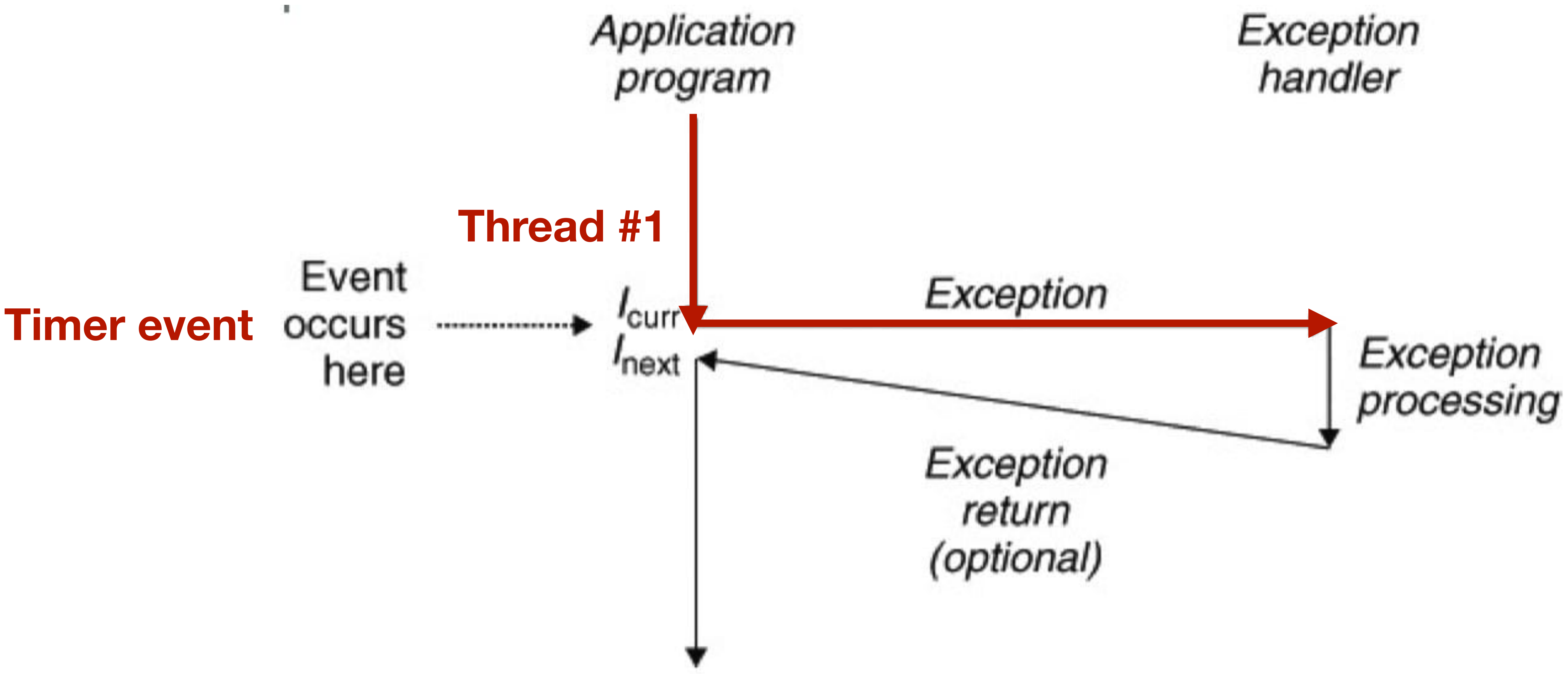
Who initiates exception?	Who handles exception?	Examples
CPU / Hardware	Operating System	Timer interrupt

CPU executes thread #1



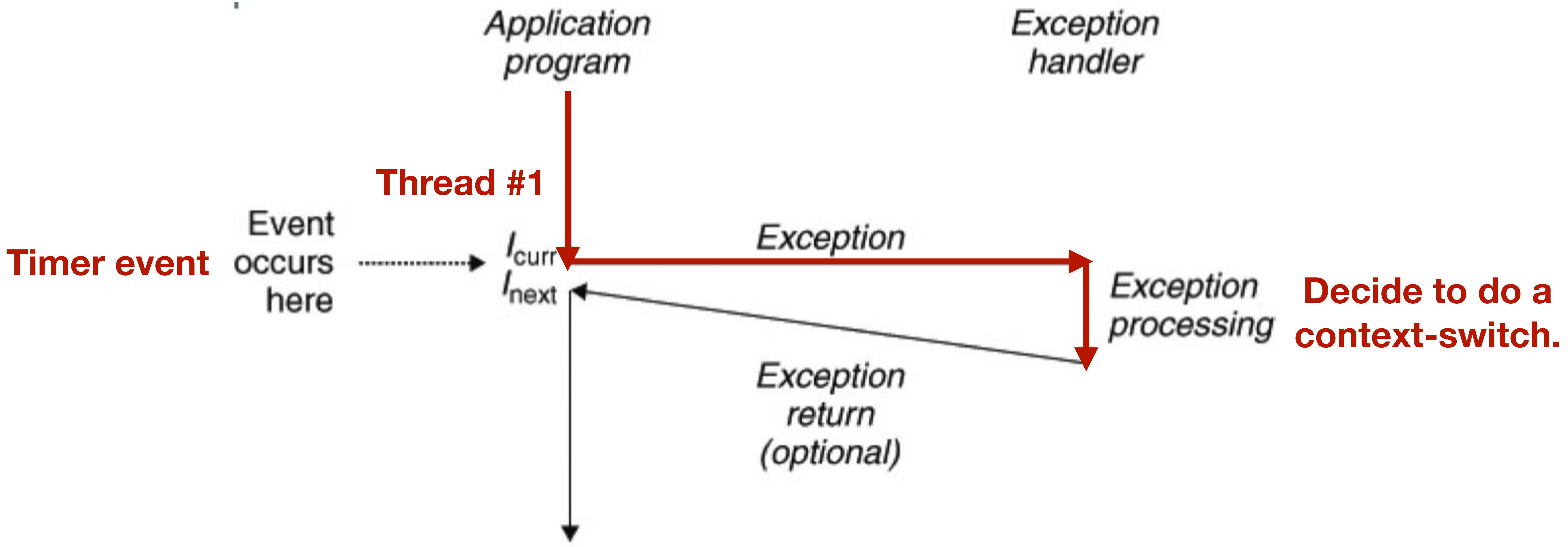
Who initiates exception?	Who handles exception?	Examples
CPU / Hardware	Operating System	Timer interrupt

Timer hardware sends an interrupt to CPU



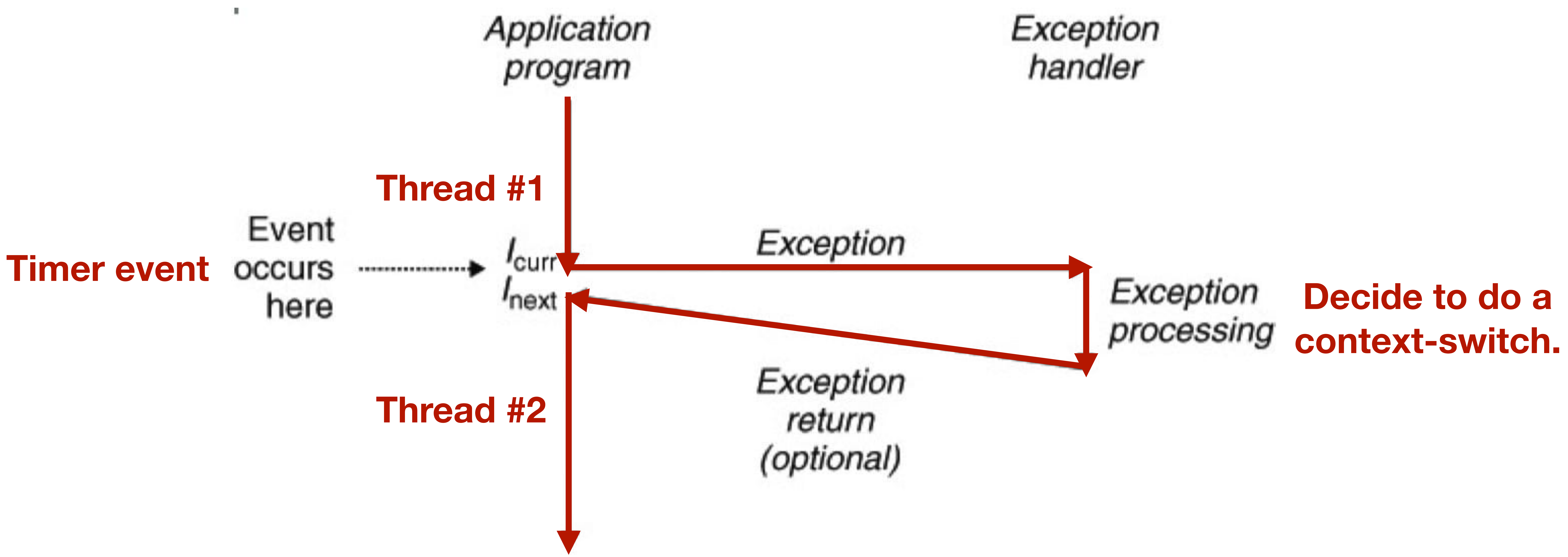
Who **initiates** exception?	Who **handles** exception?	Examples
CPU / Hardware	Operating System	Timer interrupt

OS can decide to do context-switch



Who **initiates** exception?	Who **handles** exception?	Examples
CPU / Hardware	Operating System	Timer interrupt

OS switches context to thread #2



Who initiates exception?	Who handles exception?	Examples
CPU / Hardware	Operating System	Timer interrupt

The two "Yes" is due to exception control flow

		Switching stack pointer?	Switching instruction pointer?	Switching memory address space?	Switching kernel/user mode?
4411 P1	User-level Threads	Yes	No	No	No
Beyond 4411 P1	Kernel-level Threads	Yes	Yes	No	Yes
	Processes	Yes	Yes	Yes	Yes

Exception control flow enables preemptive context-switch and also system calls.

tmp — yunhao@YunhaodeMacBook-Pro — -zsh — 67×18

~/tmp

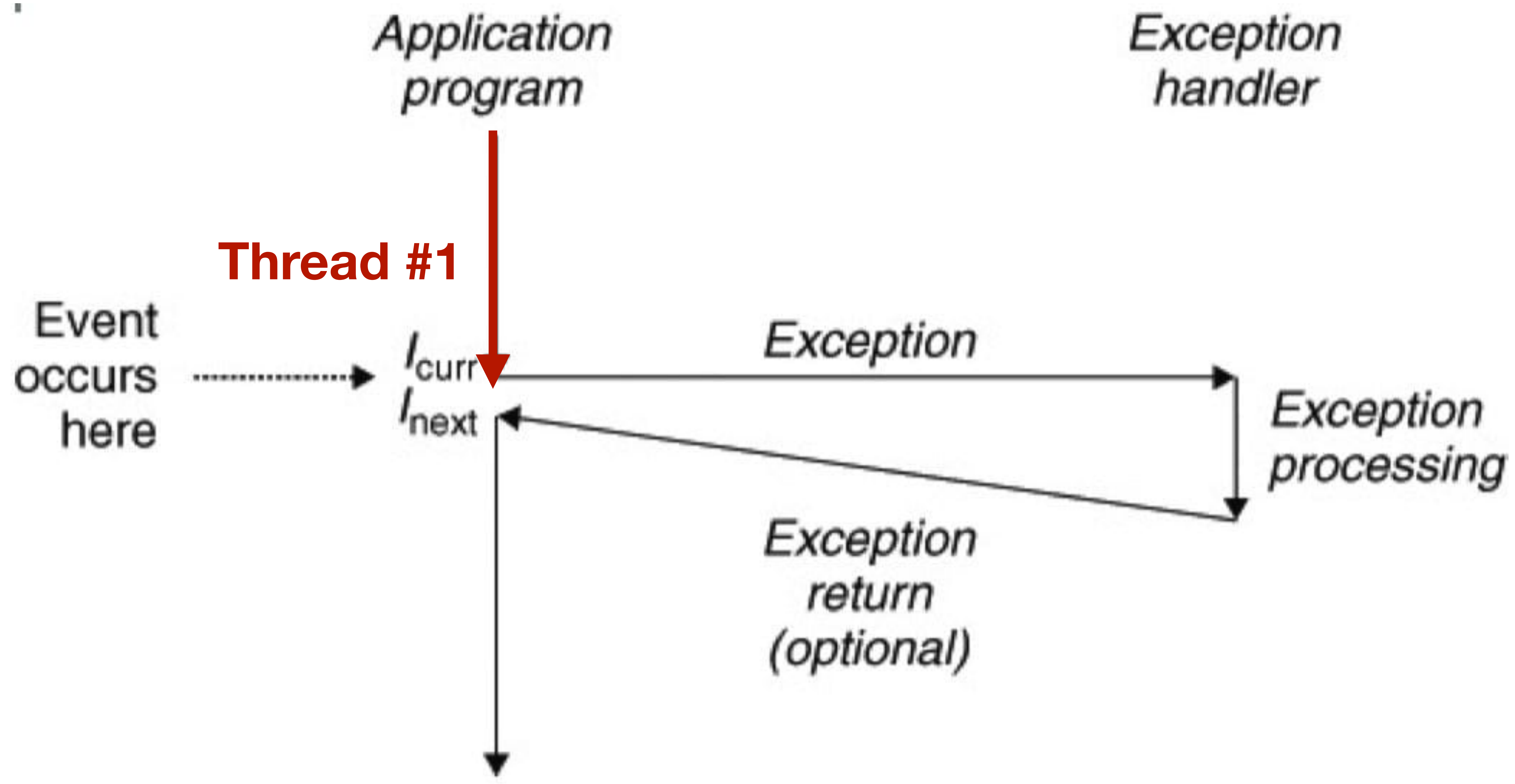
```
➜  tmp cat tmp.c
#include <stdio.h>

int main() {
    int a = 8, b = 2;
    printf("%d / %d = %d\n", a, b, a/b);
    b = 0;
    printf("%d / %d = %d\n", a, b, a/b);
    return 0;
}
➜  tmp gcc tmp.c
➜  tmp ./a.out
8 / 2 = 4
[1]    24859 floating point exception  ./a.out
➜  tmp
```

Exception also happens here! Surprise?

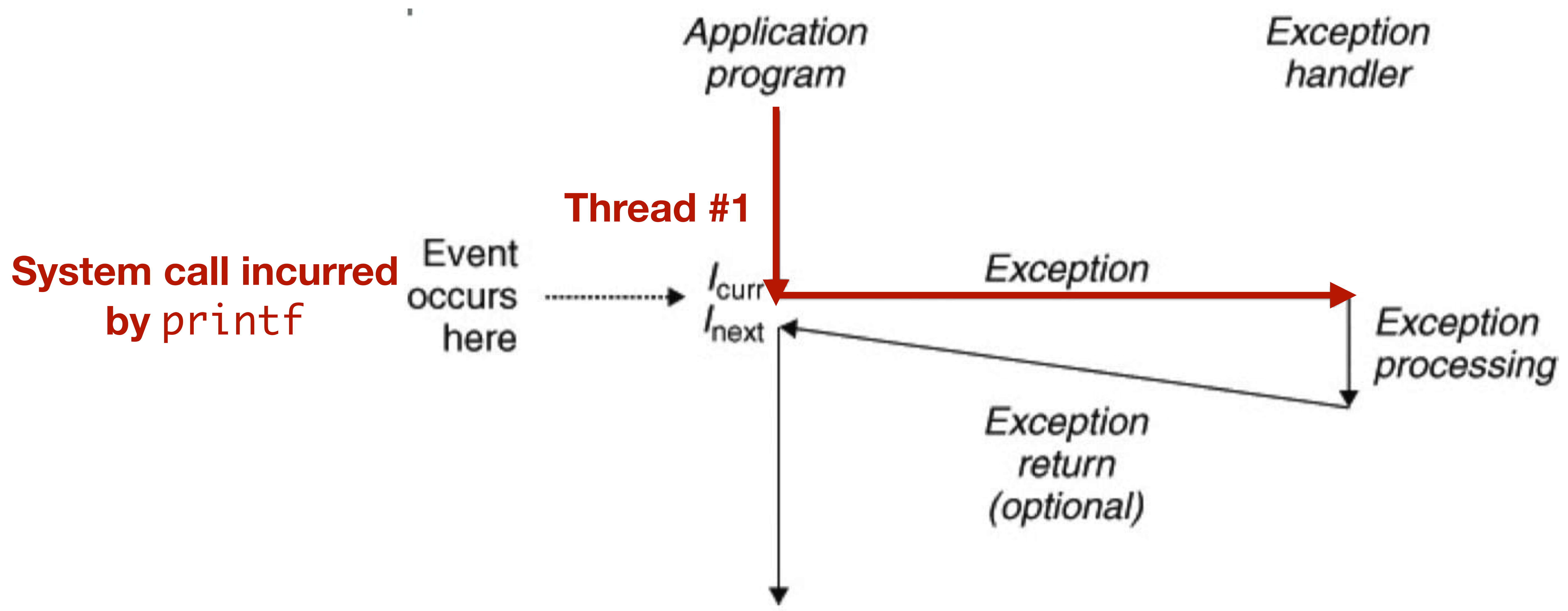
System calls also incur exception control flow

CPU executes thread #1 till I_{curr}



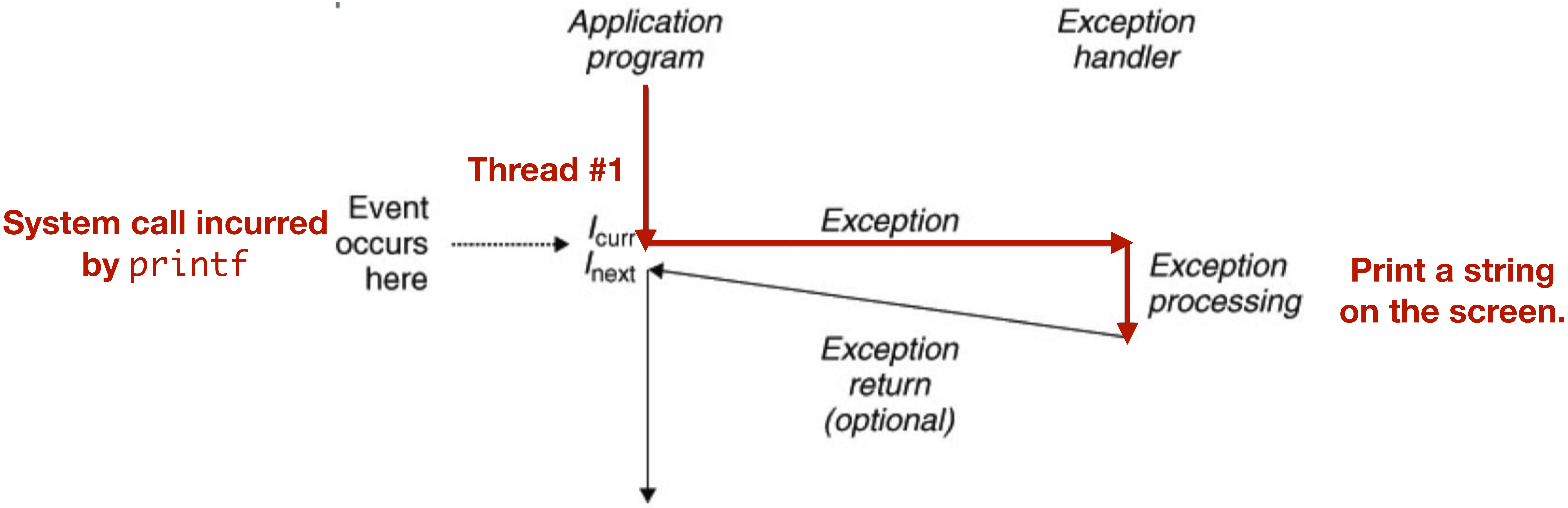
Who **initiates** exception?	Who **handles** exception?	Examples
User Application	Operating System	System Call

I_{curr} is a syscall instruction within `printf`



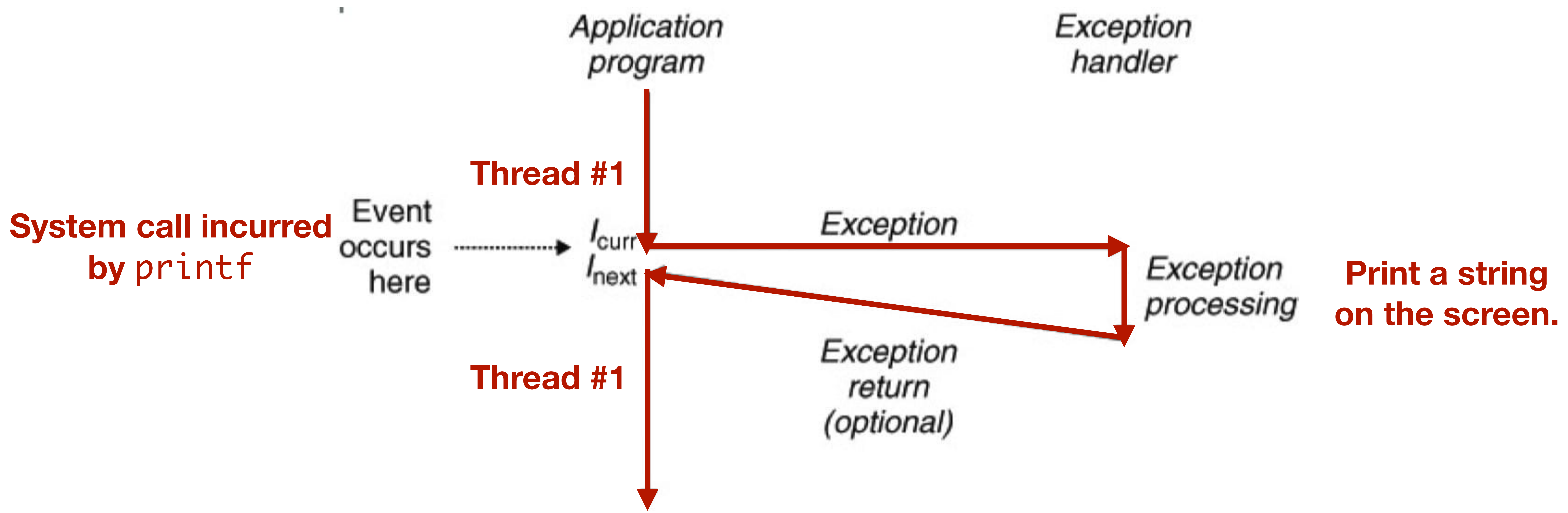
Who **initiates** exception?	Who **handles** exception?	Examples
User Application	Operating System	System Call

OS helps thread #1 print on screen



Who **initiates** exception?	Who **handles** exception?	Examples
User Application	Operating System	System Call

Thread #1 continues to execute I_{next}



Who **initiates** exception?	Who **handles** exception?	Examples
User Application	Operating System	System Call

Exception control flow enables preemptive context-switch, system calls and also safe crash of user application.

tmp — yunhao@YunhaodeMacBook-Pro — -zsh — 67×18

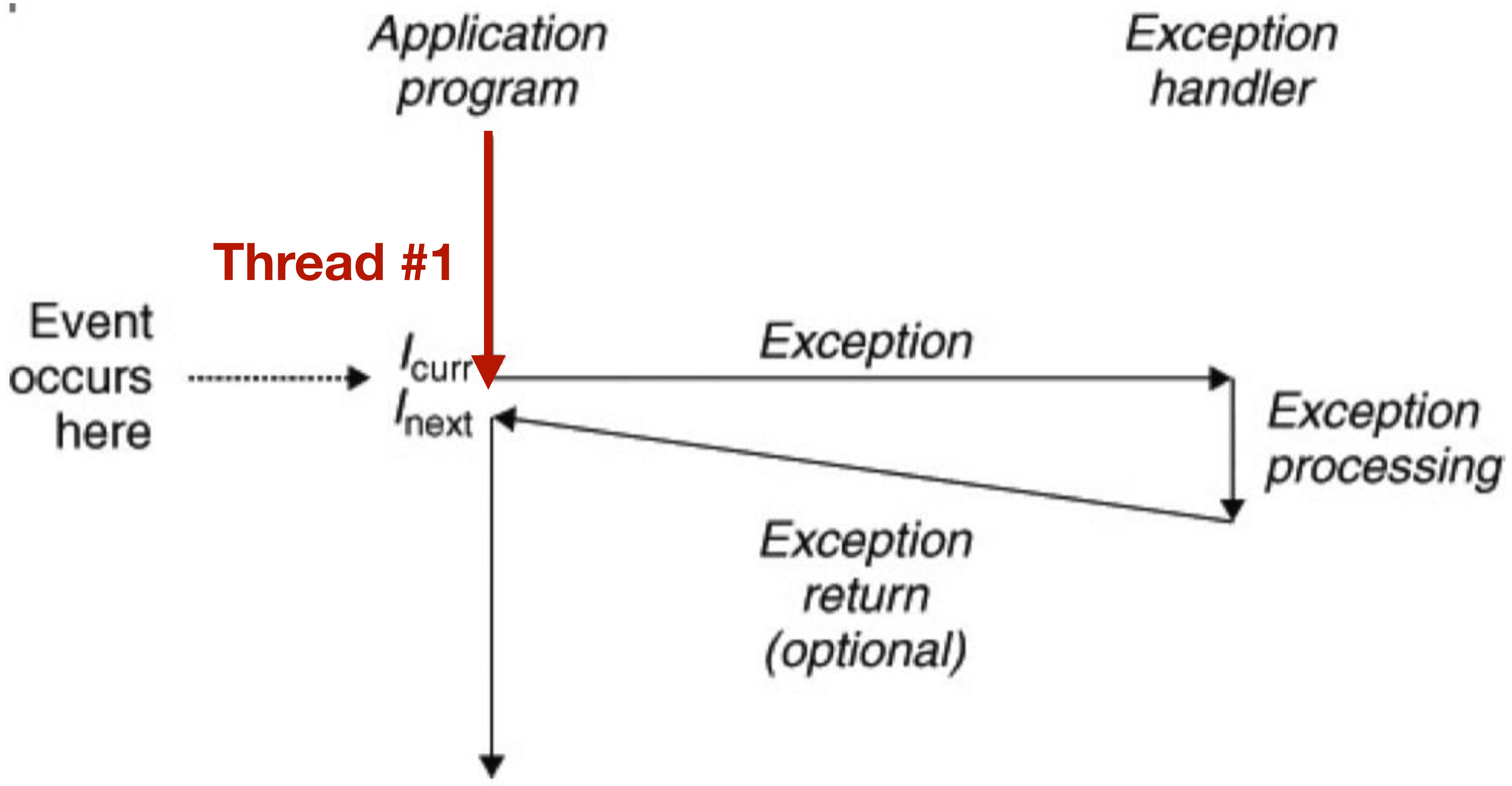
~/tmp

```
➜  tmp cat tmp.c
#include <stdio.h>

int main() {
    int a = 8, b = 2;
    printf("%d / %d = %d\n", a, b, a/b);
    b = 0;
    printf("%d / %d = %d\n", a, b, a/b);
    return 0;
}
➜  tmp gcc tmp.c
➜  tmp ./a.out
8 / 2 = 4
[1]    24859 floating point exception  ./a.out
➜  tmp
```

Exception happens due to divide 0!

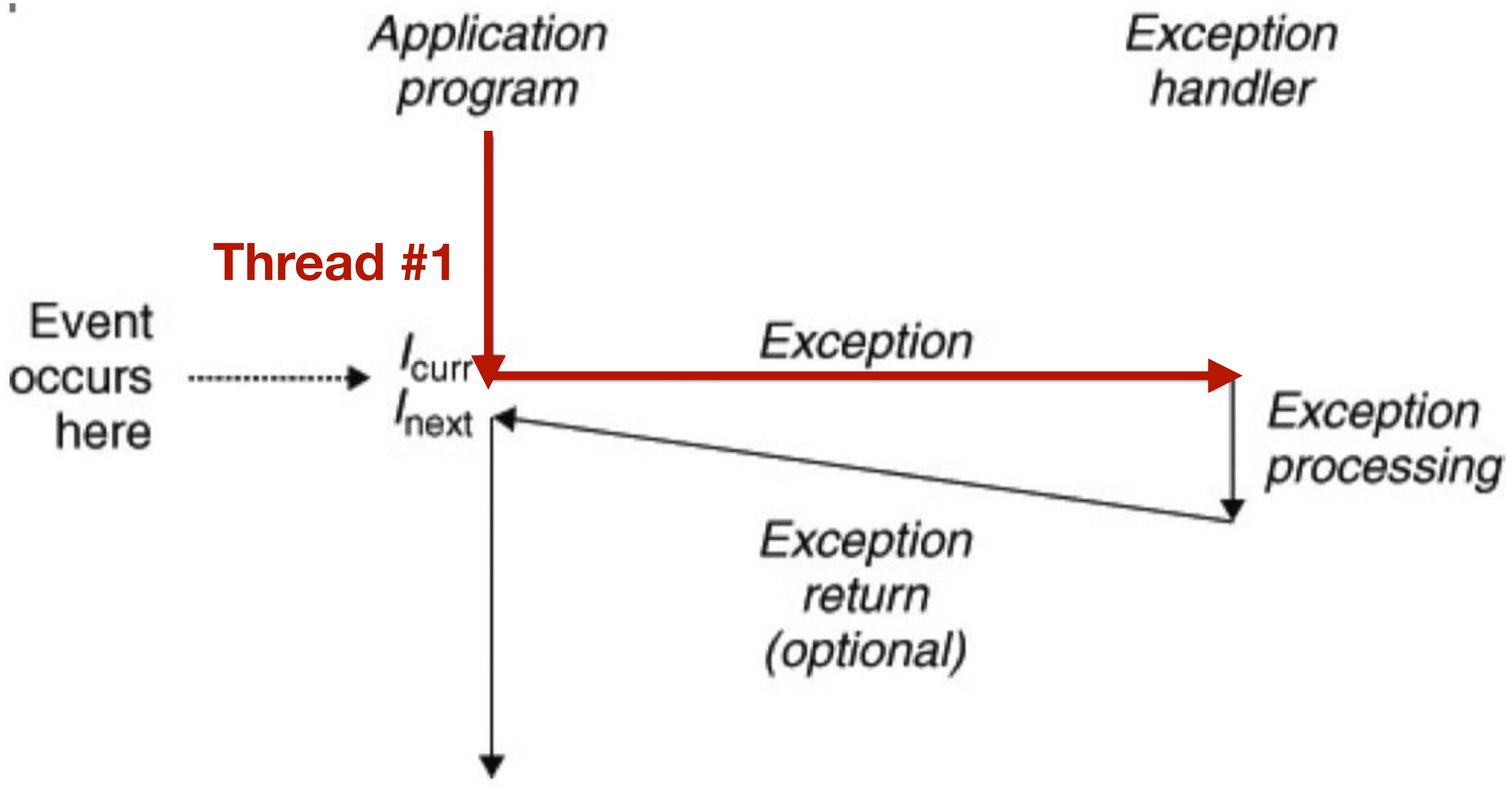
CPU executes thread #1 till I_{curr}



Who initiates exception?	Who handles exception?	Examples
User Application	Operating System	Divide-zero

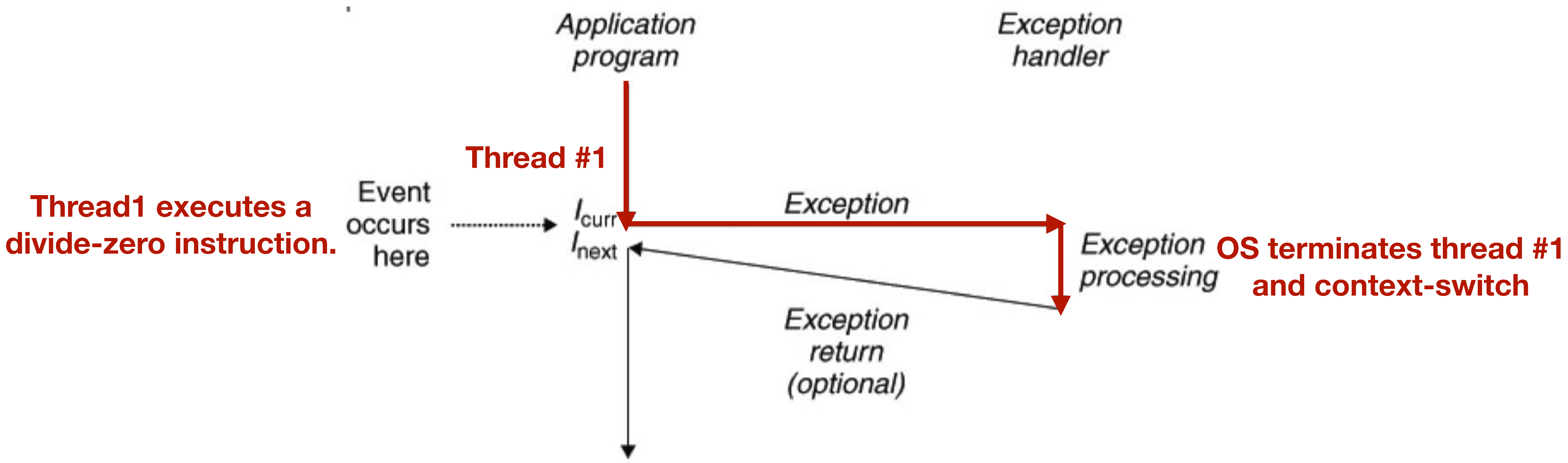
I_{curr} is a divide-zero instruction

Thread1 executes a divide-zero instruction.



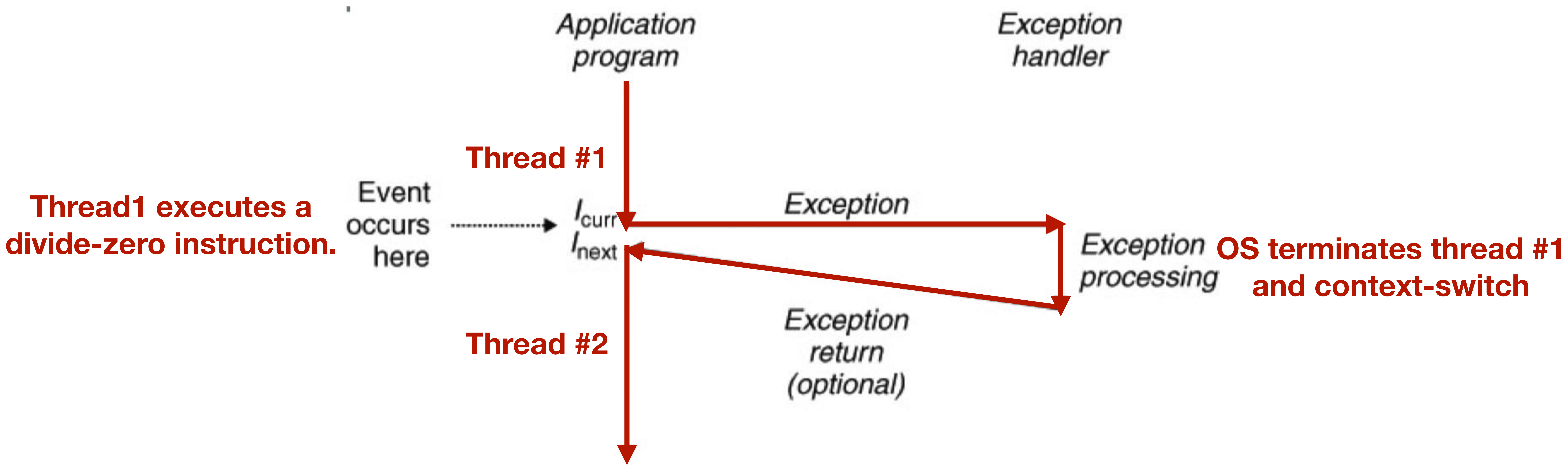
Who initiates exception?	Who handles exception?	Examples
User Application	Operating System	Divide-zero

OS terminates thread #1



Who **initiates** exception?	Who **handles** exception?	Examples
User Application	Operating System	Divide-zero

CPU executes some other thread

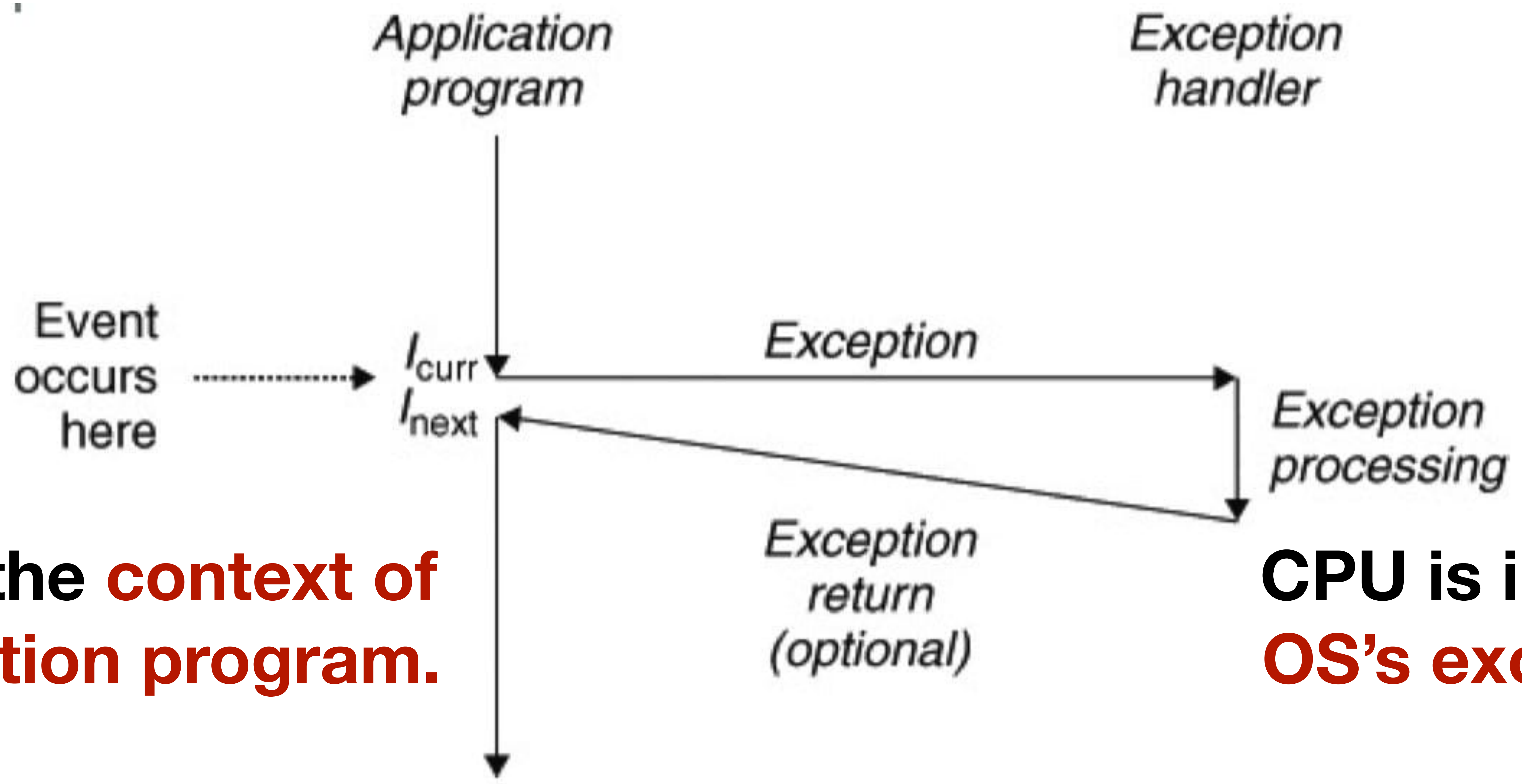


Who initiates exception?	Who handles exception?	Examples
User Application	Operating System	Divide-zero

Lesson: exception control flow enables preemptive context-switch, system calls and safe crash of user application.

These exceptions are handled by a handler function in the OS.

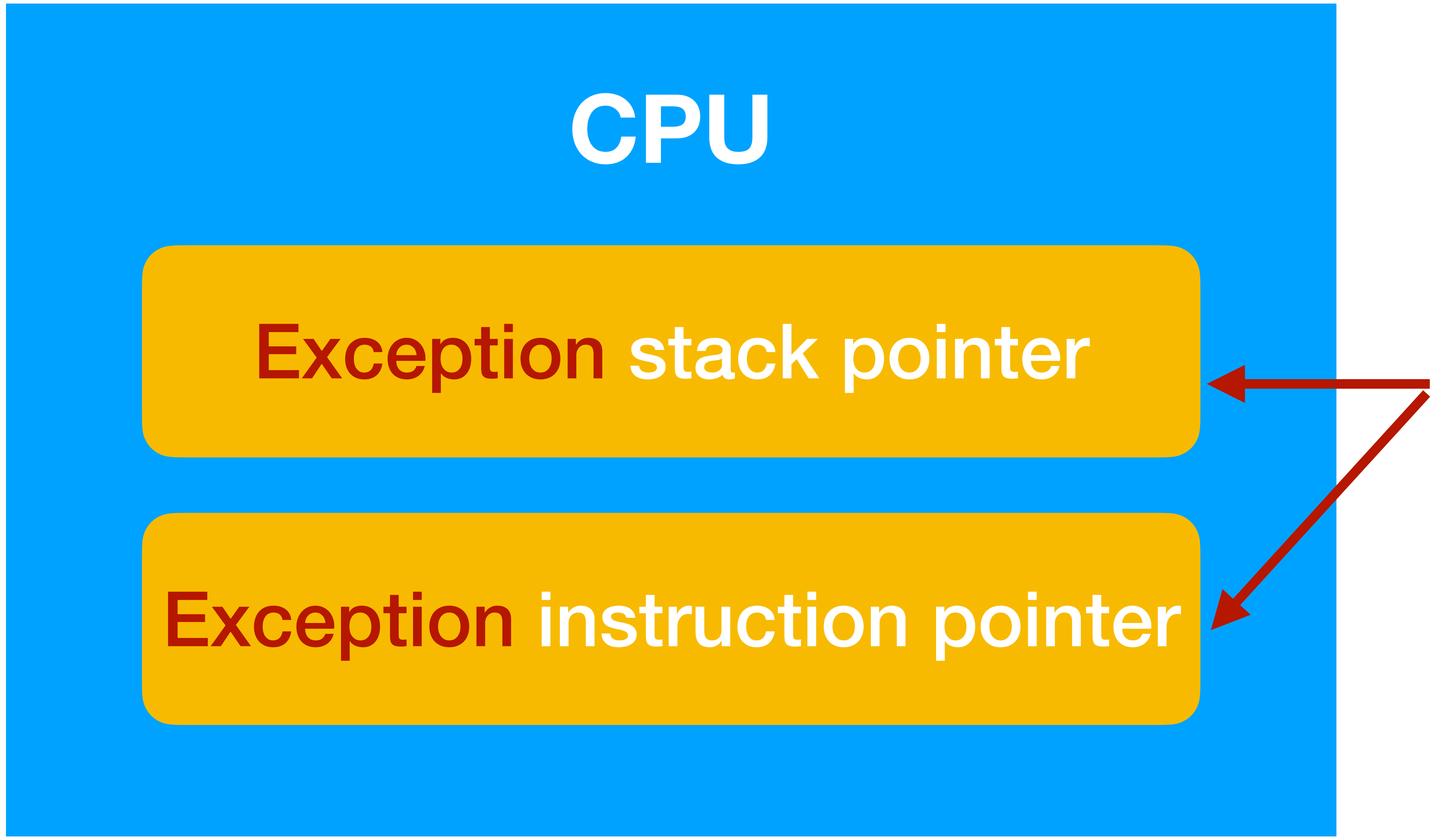
Question: how does the CPU know the context of exception handler?



CPU is in the context of an application program.

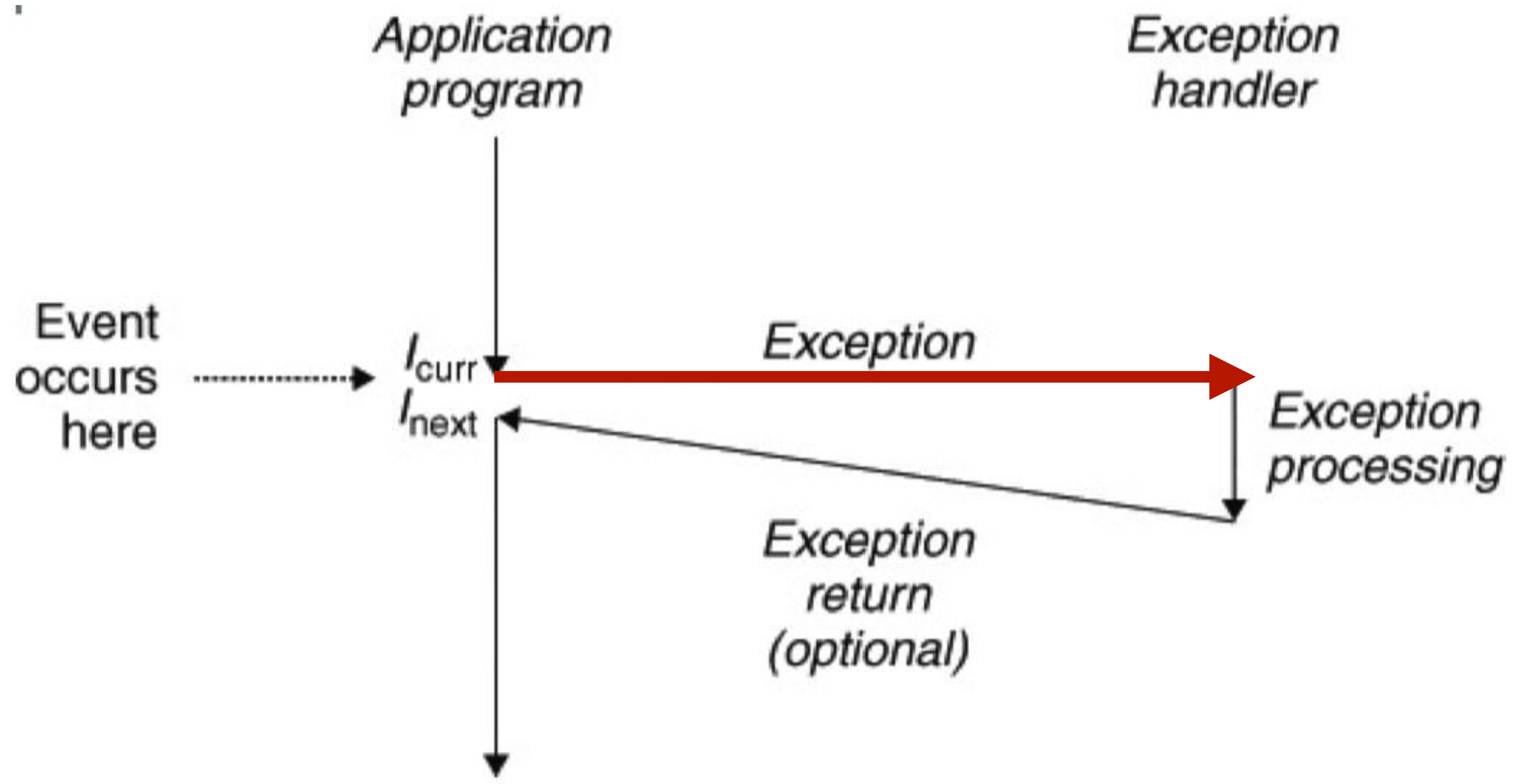
CPU is in the context of OS's exception handler.

CPU has special registers for exception

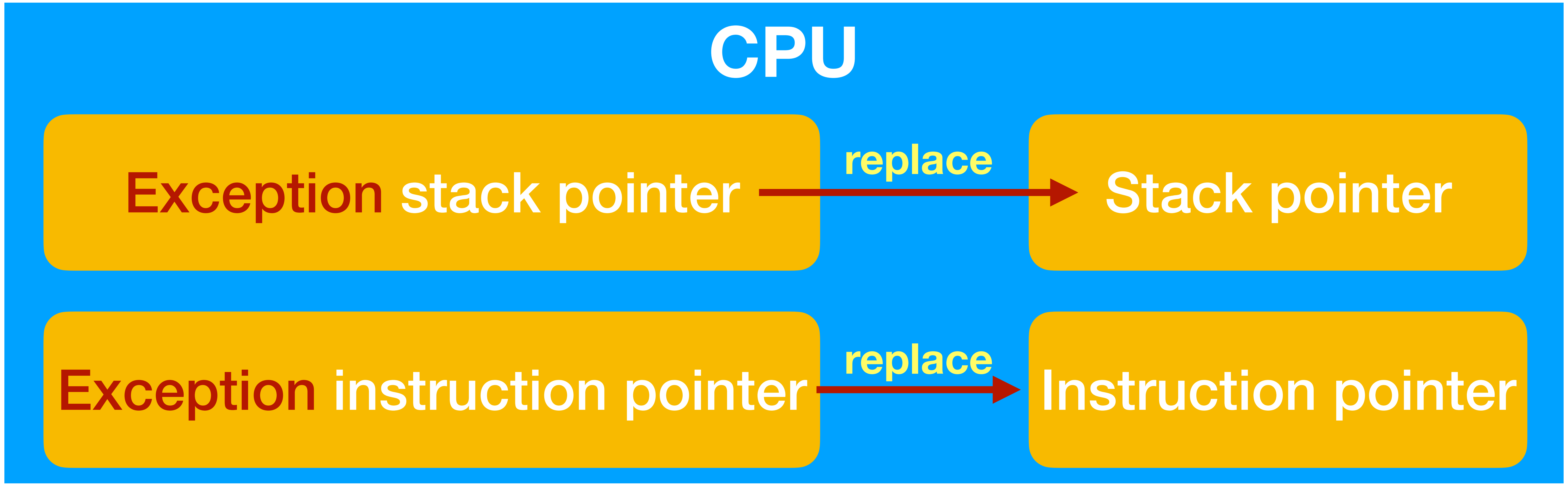


During initialization, OS record in these registers the pointers to the code & stack of its exception handler function.

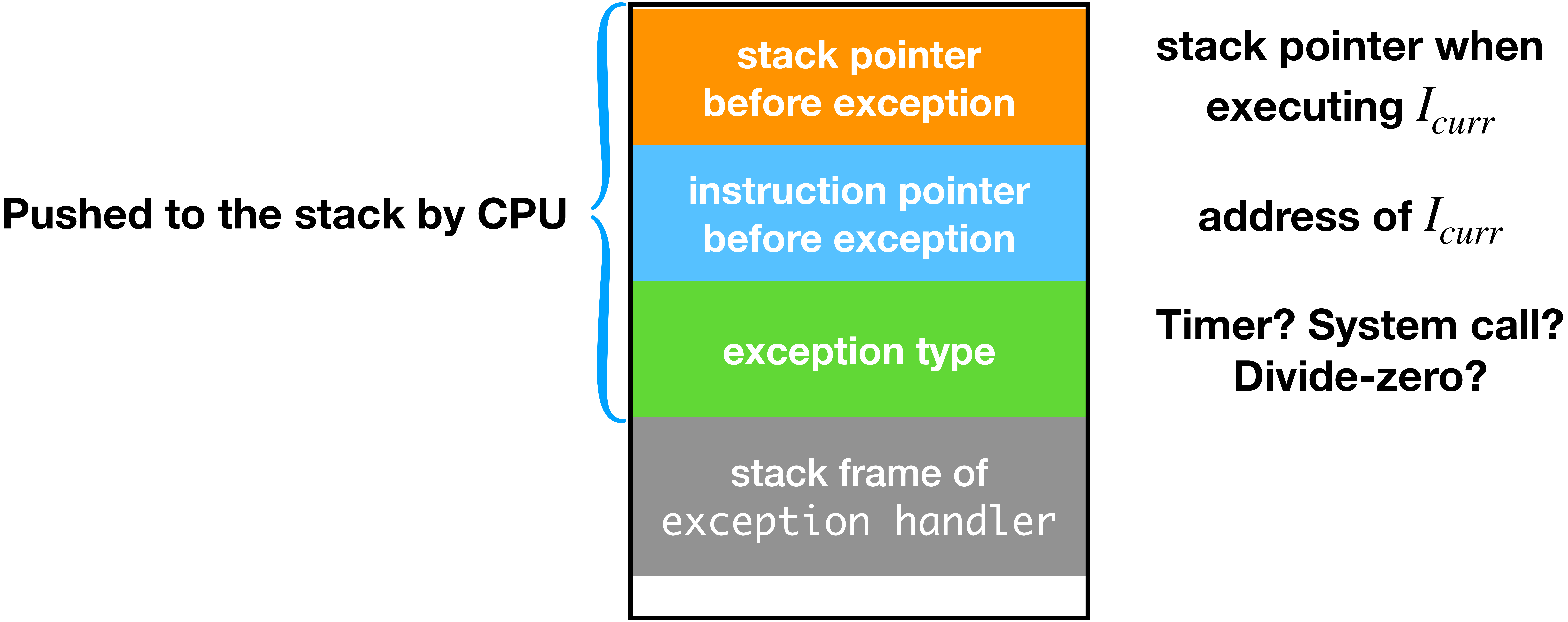
Transfer to exception handler



Transfer to the exception handler (the red arrow in left picture) is done by the two "replace" in the below picture.



Exception handler stack

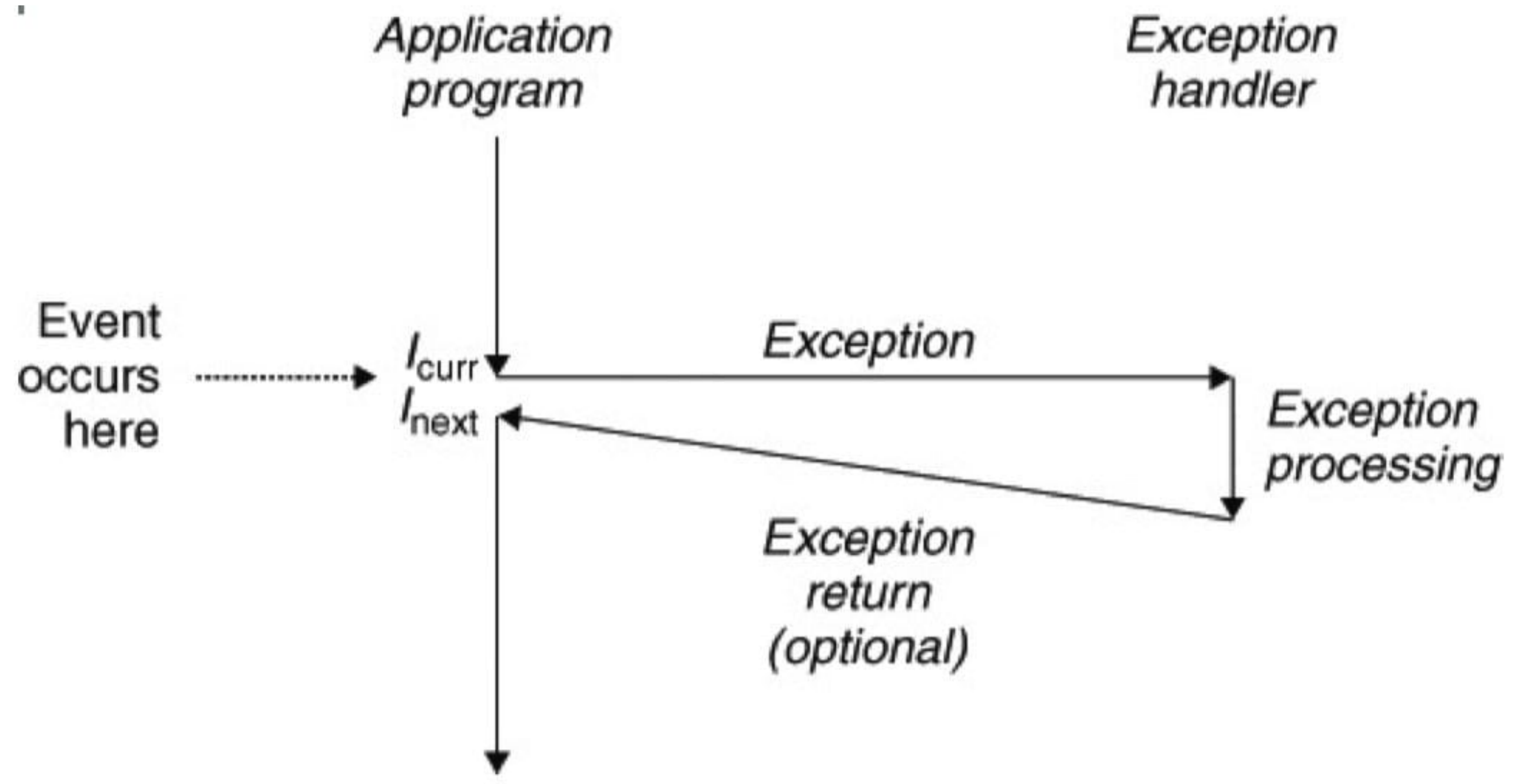


Exception handler in EGOS

```
/* This function is in src/grass/process.c */
void proc_got_interrupt(){
    switch (proc_current->intr_type) {
    case INTR_PAGE_FAULT:
        proc_pagefault((address_t) proc_current->intr_arg, true);
        break;
    case INTR_SYSCALL:
        proc_syscall();
        break;
    case INTR_CLOCK:
        proc_yield();
        break;
    case INTR_IO:
        proc_yield();
        break;
    default:
        assert(0);
    }
}
```

Summary

- Control flow is a sequence of instructions.
- An event can cause a CPU to switch from normal control flow to exception control flow, which looks like the picture below.
- Exception control flow enables preemptive context-switch, system calls and safe crash of user application.
- Exception control flow is made possible by both the OS exception handler function and the related CPU registers.



Homework

- P1 is due on Oct 2.
- P2 will be released today and due on Oct 23. Implement the concepts of preemptive context-switch and the MLFQ scheduling algorithm (next lecture).
- Further reading: the concept of IRQ: https://en.wikipedia.org/wiki/Interrupt_request_(PC_architecture)