CS 4400

Computer Systems

LECTURE 16

Exceptions

Processes

Process control

Control Flow

- The program counter assumes a sequence of values

  $$a_0, a_1, \ldots, a_{n\text{-}1}$$

  where a_k is the address of a corresponding instruction I_k.

- Each transition from a_k to a_{k+1} is called *control transfer*.

- A sequence of such control transfers is the *control flow* of the processor.

- Smooth control flow: each I_k, I_{k+1} are adjacent in memory.

- Abrupt changes to smooth flow: familiar jump, call, and return program instructions.

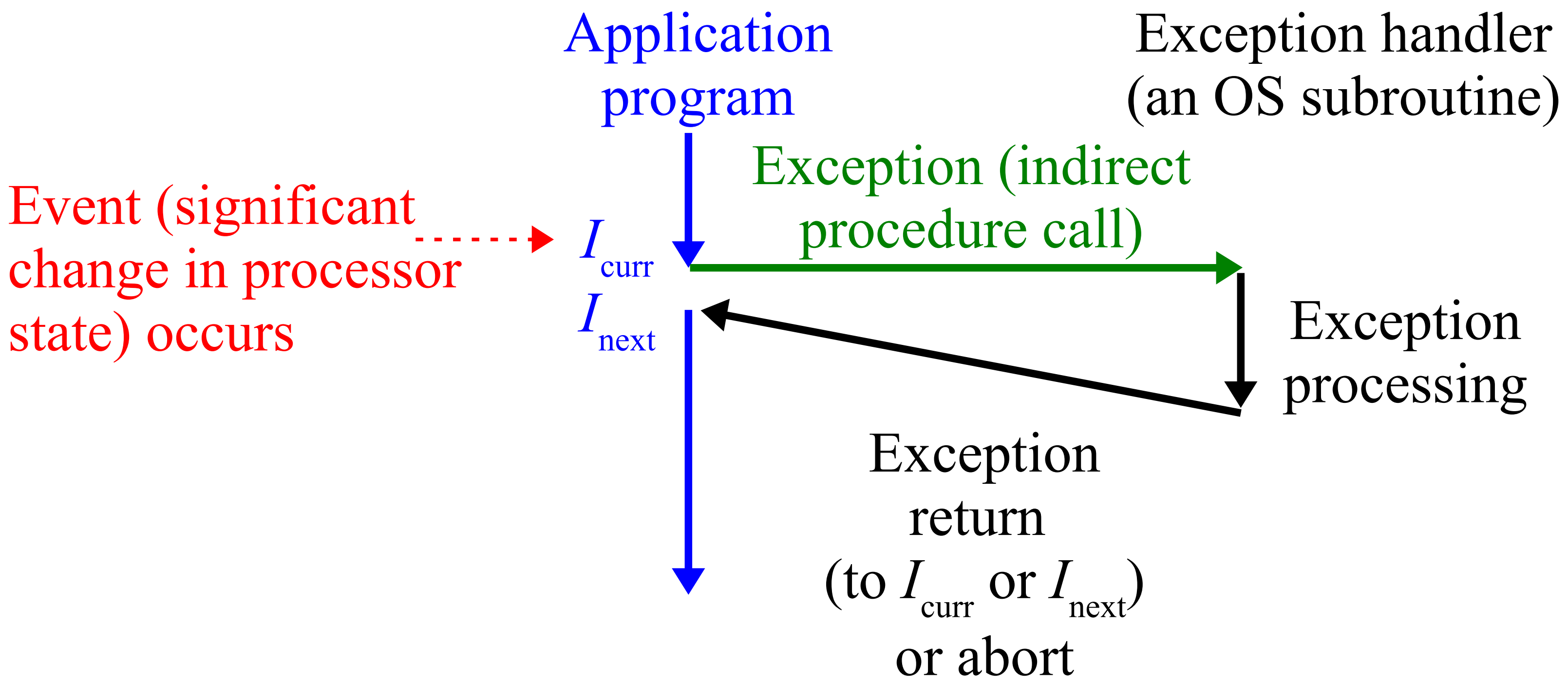
Exceptional Control Flow (ECF)

- ECF—abrupt changes in control flow that are not captured by internal program variables.
- *Hardware*: abrupt control transfers to exception handlers triggered by hardware-detected events. Examples?
- *Operating systems*: the kernel transfers control from one user process to another (via context switches).
- *Applications*: a process can send a signal to another process that abruptly transfers control to a signal handler (at the receiving process).

Why Care About ECF

- *To understand important systems concepts*
  - the basic mechanism OSs use to implement I/O, processes, VM
- *To understand how apps interact with the OS*
  - apps request services from the OS using a trap (or system call)
- *To write interesting new application programs*
  - the OS provides apps with mechanisms for ... (writing a shell)
- *To understand how software exceptions work*
  - C++/Java provide software exception mechanisms, allowing a program to make nonlocal jumps (high level)
  - nonlocal jump functions are provided in C (low level)

Exceptions



The event might be directly related to I_{curr} (e.g., divide by 0), or the event may be unrelated (e.g., an I/O request completes).

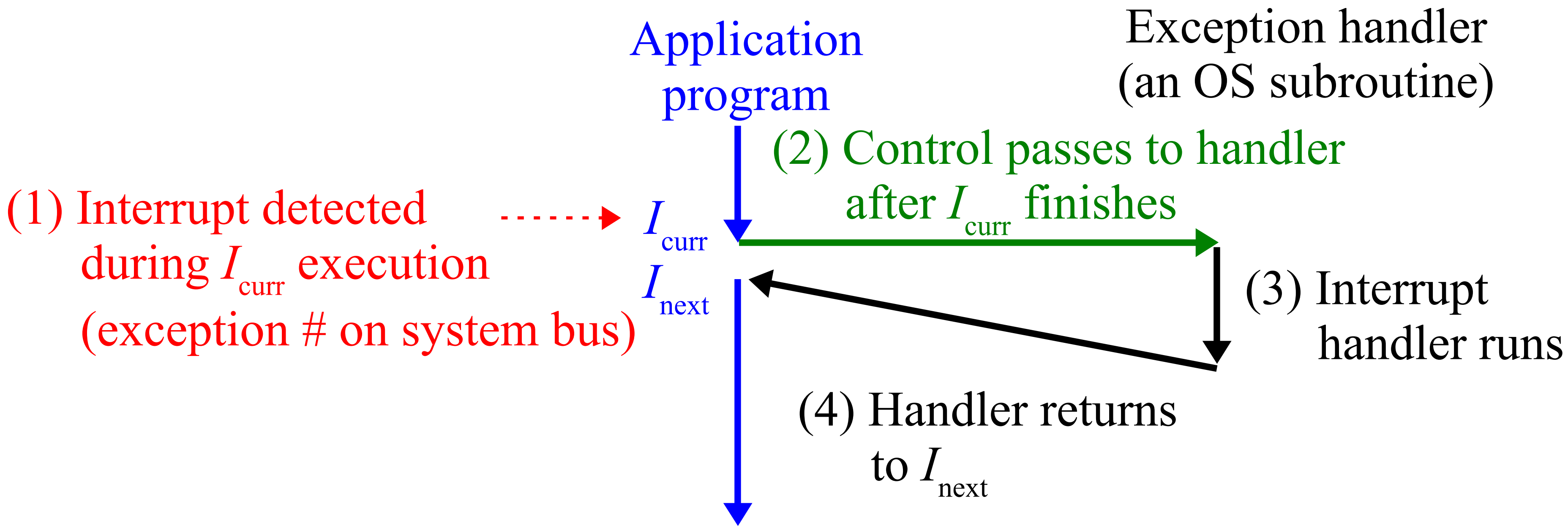
Exception Handling

- Each possible type of exception gets a unique integer > 0
  - some assigned by processor designers (div by 0, page fault, ...)
  - others assigned by OS kernel designers (system calls, signals)
- At boot time, the OS allocates and initializes an *exception table* (a jump table).
  - entry *k* contains the address of the handler code for exception *k*
- When the processor detects an event, it determines *k* and makes an indirect procedure call to the handler for *k*.
  - a special CPU register holds starting address of exception table
  - the exception handler is an index into the exception table

Exception vs. Procedure Call

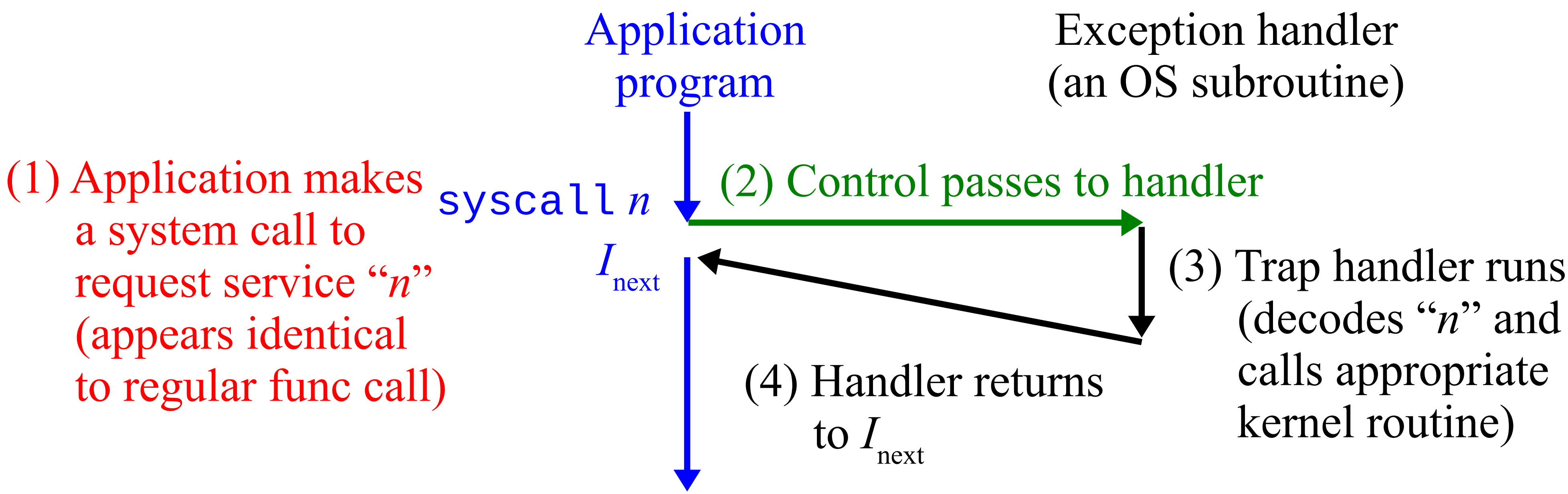
- Both push a return address onto the stack before branching to handler. For exception, may be I_{curr} or I_{next}.
- For exception, also pushes some processor state necessary to restart the interrupted program on return.
- For exception, if control is being transferred to the kernel, all items are pushed onto the kernel's stack (instead of the user's stack).
- Exception handlers run in kernel mode (complete access to all system resources).

Exception Class: Interrupts



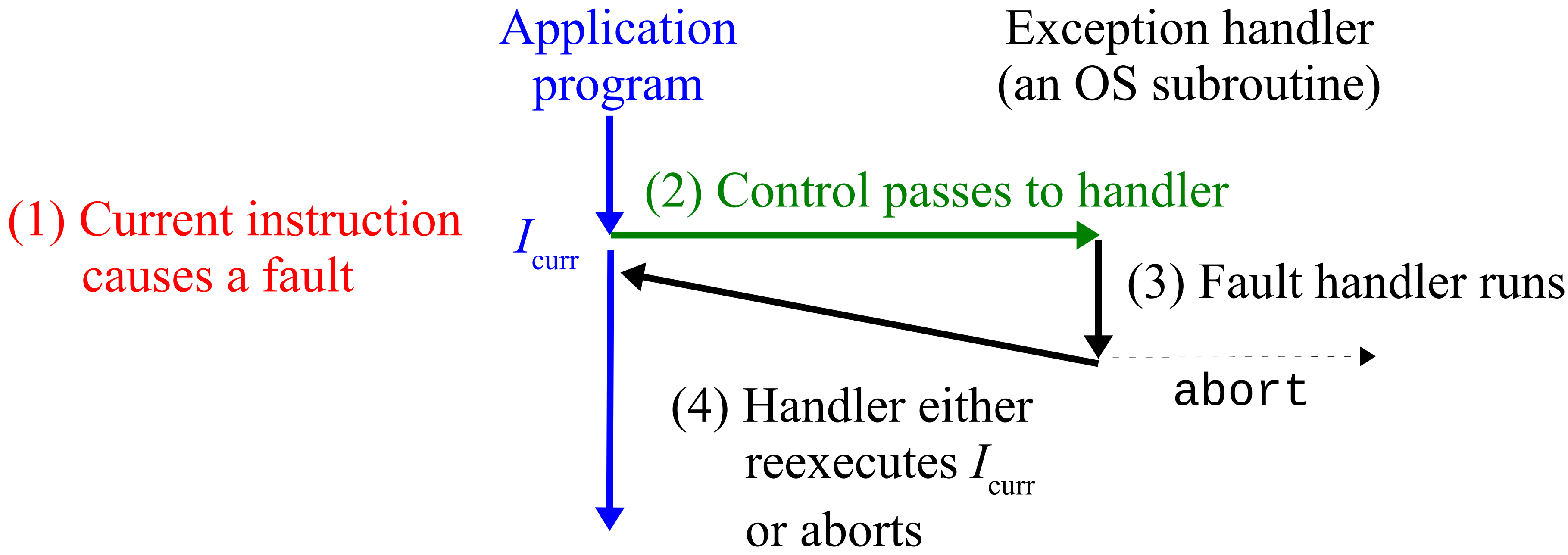
- *Interrupts* occur asynchronously as a result of signals from I/O devices external to the processor.
    - asynchronous because not caused by execution of an instruction
- Effect—program executes as if the interrupt never happened.

Exception Class: Traps



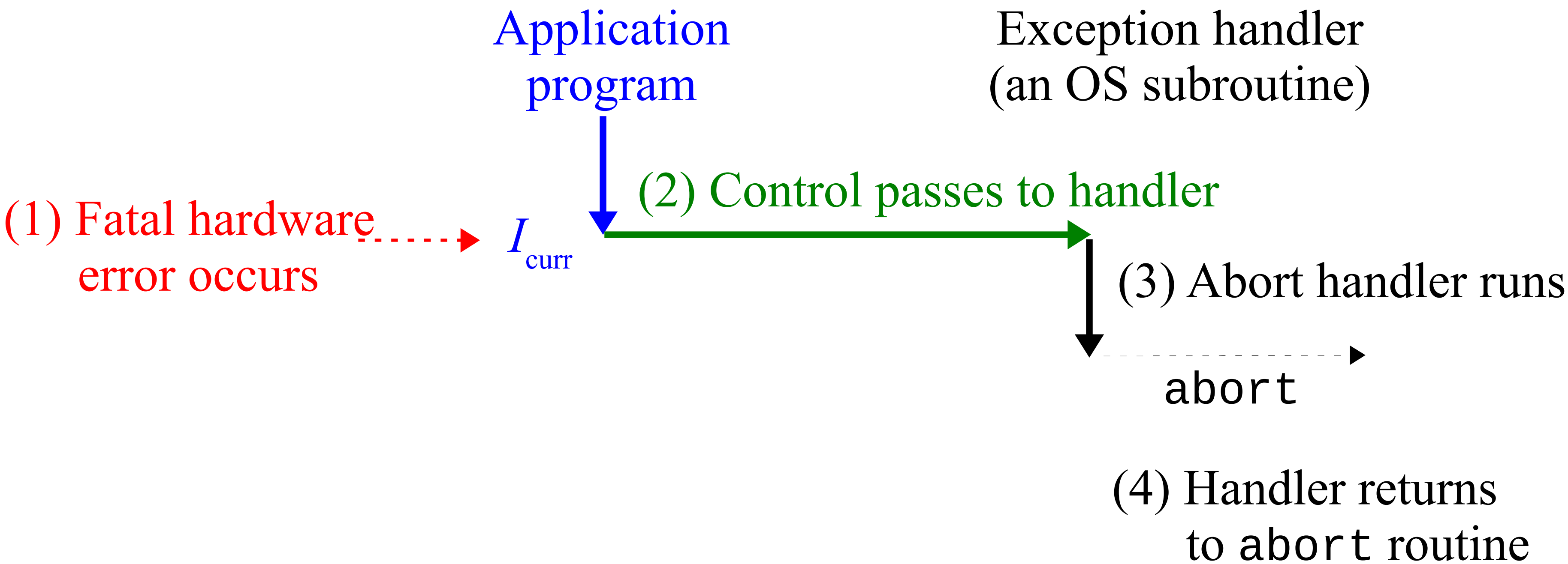
- Traps are intentional exceptions that occur as a result of executing an instruction.
    - provides procedure-like interface between user programs and kernel (system call)

Exception Class: Faults



- *Faults* result from error conditions that a handler might be able to correct.
  - *classic example*: page fault exception

Exception Class: Aborts



- Aborts result from unrecoverable fatal errors.
  - such as parity errors, when DRAM or SRAM bits are corrupted

Example: Pentium Exceptions

Number	*Description*	*Class*
0	Divide error	Fault (Unix does not recover)
13	General protection fault (ref to undefined memory)	Fault (Unix does not recover)
14	Page fault	Fault (faulting instruction restarted)
18	Machine check (fatal hardware error)	Abort
32-127	OS-defined exceptions	Interrupt or Trap
128	System call (trapping instruction `INT` *n*)	Trap
129-255	OS-defined exceptions	Interrupt or Trap

Processes

- *Process*—an instance of a program in execution
- Provides the illusion that our program is the only one currently running in the system.
  - exclusive use of processor and memory
  - instructions executed one after another without interruption
  - program code and data are the only objects in system's memory
- Each program runs in the *context* of some process.
  - program's code and data stored in memory, run-time stack, register contents, program counter, environment variables, and set of open file descriptors

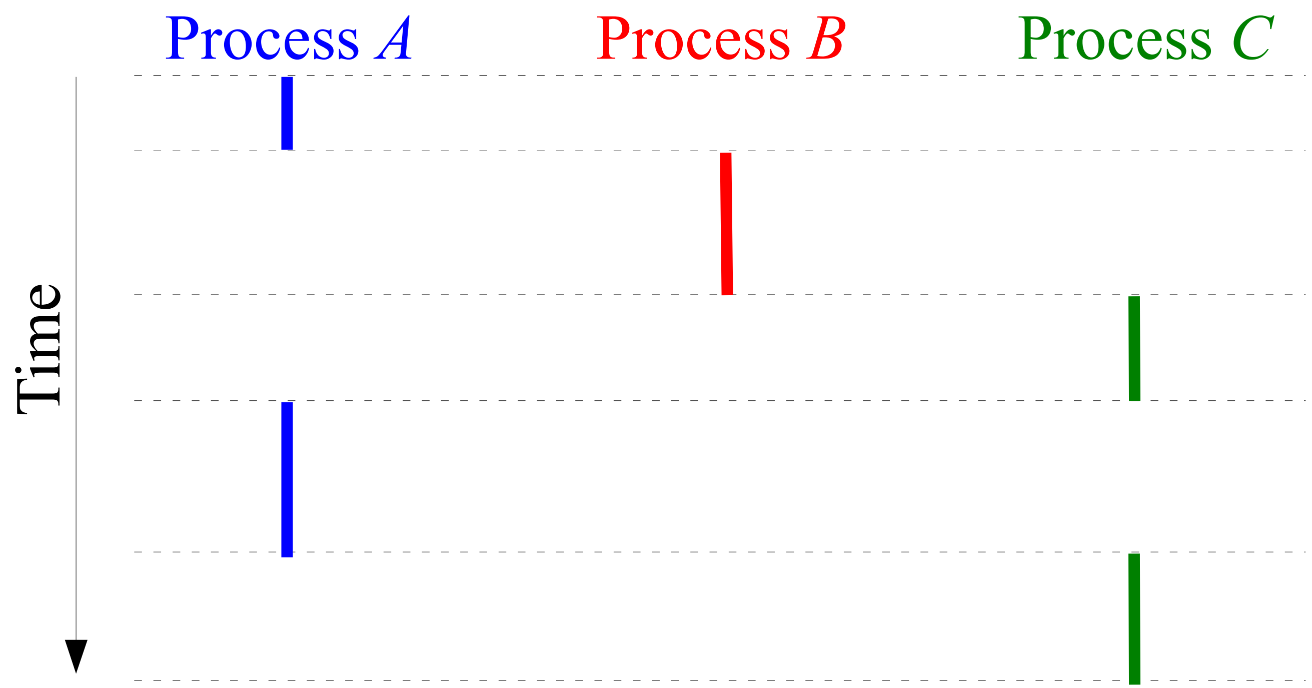
Processes

- When the user types the name of an executable object file at the shell prompt,
  - the shell creates a new process
  - the shell runs the program in the context of this new process
- Applications can also create new processes.
- Two key abstractions are provided by processes:
  - an independent logical control flow (illusion of exclusive use of processor)
  - private address space (illusion of exclusive use of memory)

Logical Control Flow

- Logical control flow—a sequence of PC values that correspond exclusively to instructions in our program's executable object file.
  - or in shared objects linked into our program dynamically
- *Multitasking*—each process executes a portion of its flow, then is preempted while other processes take their turns.
  - *time slice*—each time period that process executes a portion of its flow
- A precise timing of instructions is the only evidence that our process does not have exclusive use of the processor.

Example: Logical Control Flow

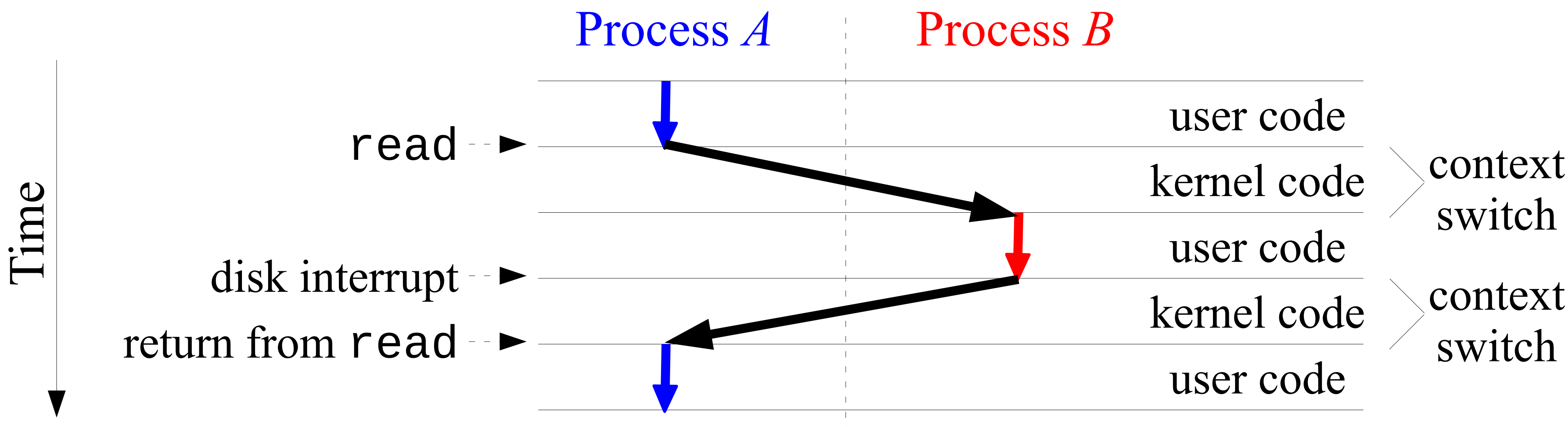


- The single physical control flow of the processor is partitioned into three logical flows.
- *A* & *B* are running concurrently, *B* & *C* are not.

Context Switches

- *Scheduling*—decision by kernel to preempt the current process and restart a previously preempted process.
- After the kernel has scheduled a new process to run, it preempts the current process and transfers control to the new process using a *context switch*.
- The context switch
  - saves the context of the current process
  - restores the saved context of a previously preempted process
  - passes control to the newly restored process

Example: Context Switch



- Process *A* issues a read that requires disk access.
- Instead of waiting for the data, the kernel opts to perform a context switch and run process *B*.
- Once the disk sends an interrupt, the kernel performs a context switch from *B* to *A*.
- Control returns to *A* at the instruction immediately after the read.

System Calls

- Unix provides *system calls* for applications to use when they want to request services from the kernel.
- Rather than invoke a system call directly, the C library offers a set of wrapper functions for most system calls.
- When such system-level functions encounter an error, they set error codes that *should always be checked*.

```
if((pid = fork()) < 0) {
  fprintf(stderr, “fork error: %s\n”,
          strerror(errno));
  exit(0);
}
```

→

```
pid = Fork();
```

- (See the text for these useful error-handling wrappers.)

Getting Process IDs

- Unix provides systems calls for manipulating processes from C programs.
- Each process has a unique *process ID* (PID) > 0.

```
#include <unistd.h>
#include <sys/types.h>

/* returns PID of current process */
pid_t getpid(void);

/* returns PID of parent of current process */
pid_t getppid(void);
```

Process States

From the perspective of the programmer, a process can be in one of three states.

- *Running*—either executing on CPU or waiting to be executed and will eventually be scheduled.
- *Stopped*—execution suspended, will not be scheduled.
  - received a `SIGSTOP`, `SIGTSTP`, `SIGTTIN`, or `SIGTTOU` signal
  - must receive a `SIGCONT` signal to become running again
- *Terminated*—stopped permanently.
  - receiving a signal whose default action is to terminate process
  - returning from `main`
  - calling `exit`

`fork` Function

- A parent process creates a new running child process .

```
pid_t fork(void);
```

- The child process is nearly identical to the parent.
  - duplicate, but separate address spaces (stack, heap, ...)
  - identical copies of parent's open file descriptors
  - parent and child have different PIDs
- The `fork` function returns twice!
  - once in the calling process (the parent)—returns the child's PID
  - once in the newly created child process—returns 0
- Parent and child are separate processes running concurrently.

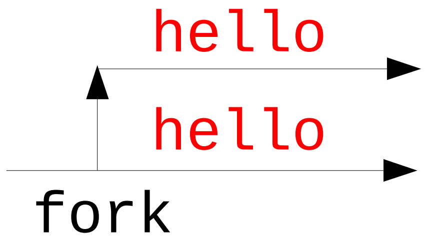
Example: `fork`

```
/* fork.c */

#include “csapp.h”    /* error-handling wrappers */

int main() {
  pid_t pid;
  int x = 1;           /* each process gets copy */

  pid = Fork();
  if(pid == 0) {       /* child */
    printf("child : x=%d\n", ++x);
    exit(0);
  }

  /* parent */
  printf("parent: x=%d\n", --x);
  exit(0);
}
```

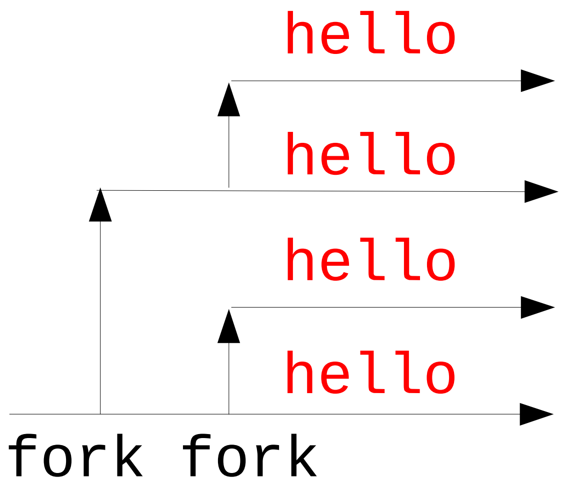
```
unix> ./fork
parent: x = 0
child : x = 2
```

Example: `fork`

```
#include "csapp.h"

int main() {
  Fork();
  printf("hello\n");
  exit(0);
}
```




```
#include "csapp.h"

int main() {
  Fork();
  Fork();
  printf("hello\n");
  exit(0);
}
```



Exercise: `fork`

```
#include “csapp.h”

int main() {
  int x = 1;

  if(Fork() == 0)
    printf(“printf1: x=%d\n”, ++x);
  printf(“printf2: x=%d\n”, --x);
  exit(0);
}
```

Output of parent process?

Output of child process?

Clicker Question

```
#include “csapp.h”

int main() {
  int i;

  for(i = 0; i < 2; i++)
    Fork();
  printf(“hello\n”);
  exit(0);
}
```

How many “hello” output lines does this program print?

CLICK your one-digit answer.

Clicker Question

```
#include “csapp.h”

int doit() {
  Fork();
  Fork();
  printf(“hello\n”);
  exit(0);
}

int main() {
  doit();
  printf(“hello\n”);
  exit(0);
}
```

How many “hello” output lines does this program print?

CLICK your one-digit answer.

Clicker Question

```
#include "csapp.h"

int doit() {
  Fork();
  Fork();
  printf("hello\n");
  return;
}

int main() {
  doit();
  printf("hello\n");
  exit(0);
}
```

How many "hello" output lines does this program print?

CLICK your one-digit answer.