ABNORMAL CONTROL FLOW ABSTRACTIONS

Professor Ken Birman
CS4414 Lecture 7

IDEA MAP FOR TODAY

In many situations, we have a normal control flow but must also deal with abnormal events.

Can Dijkstra's concept of creating abstractions offer a unified way to deal with abnormal control flow?

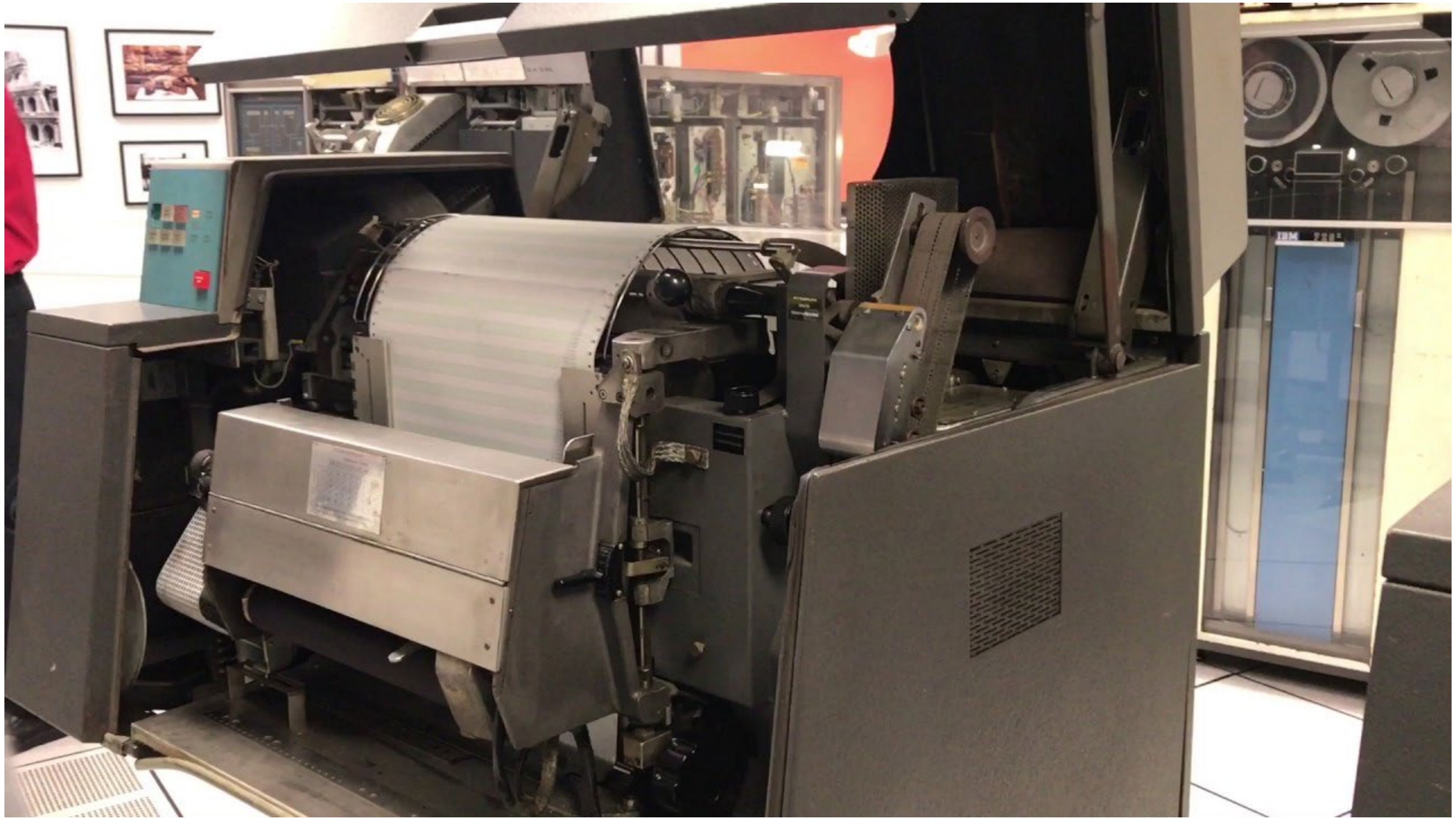
The hardware has this issue: an I/O event might finish more or less at any instant. Interrupts are like procedure calls that occur "when needed".

Linux offers programmable signal handling mechanisms that mimic interrupts.

C++ offers a similar concept via its throw statement, and the try/catch control structure.

All forms of exceptions can disrupt computation, making it very hard to write a "safe" handler!

PRINTERS APPARENTLY USED TO CATCH FIRE FAIRLY OFTEN!



HIGHLY EXCEPTIONAL CONTROL FLOW

```c
static int lp_check_status(int minor)
{
        int error = 0;
        unsigned int last = lp_table[minor].last_error;
        unsigned char status = r_str(minor);
        if ((status & LP_PERRORP) && !(LP_F(minor) & LP_CAREFUL))
                /* No error. */
                last = 0;
        else if ((status & LP_POUTPA)) {
                if (last != LP_POUTPA) {
                        last = LP_POUTPA;
                        printk(KERN_INFO "lp%d out of paper\n", minor);
                }
                error = -ENOSPC;
        } else if (!(status & LP_PSELECD)) {
                if (last != LP_PSELECD) {
                        last = LP_PSELECD;
                        printk(KERN_INFO "lp%d off-line\n", minor);
                }
                error = -EIO;
        } else if (!(status & LP_PERRORP)) {
                if (last != LP_PERRORP) {
                        last = LP_PERRORP;
                        printk(KERN_INFO "lp%d on fire\n", minor);
                }
                error = -EIO;
        } else {
                last = 0; /* Come here if LP_CAREFUL is set and no
                             errors are reported. */
        }

        lp_table[minor].last_error = last;

        if (last != 0)
                lp_error(minor);

        return error;
}
```

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/char/lp.c?h=v5.0-rc3

TODAY

Exceptional Control Flow

Linux signals

Programming language-level exceptions

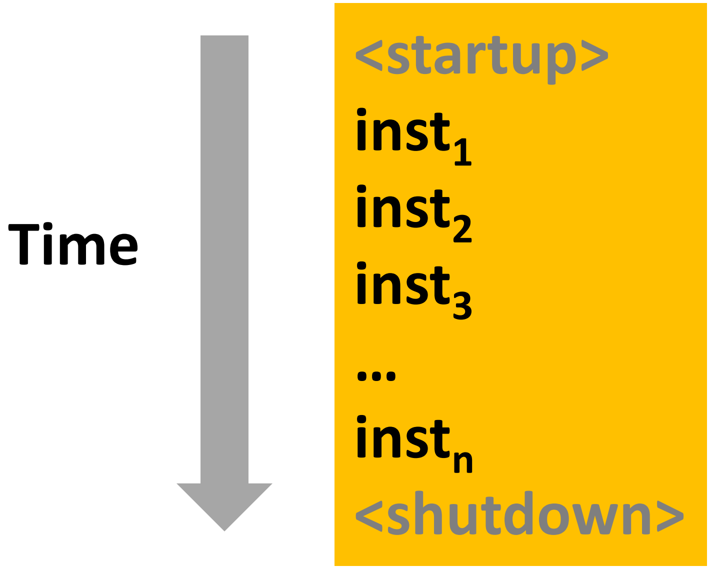
C++ features for handling exceptions

CONTROL FLOW

Processors do only one thing:

- From startup to shutdown, a CPU simply reads and executes (interprets) a sequence of instructions, one at a time
- This sequence is the CPU's *control flow* (or *flow of control*)

Physical control flow



ALTERING THE CONTROL FLOW

Up to now: two mechanisms for changing control flow:

- Jumps and branches… Call and return
- In effect, we change control flow to react to changes in program state

Insufficient: We also need to react to changes in <u>system</u> state

- Data arrives from a disk or a network adapter
- Instruction divides by zero
- User hits Ctrl-C at the keyboard… Timer expires…

EXCEPTIONS: SEVERAL "FLAVORS" BUT MANY COMMONALITIES

All exceptions "seize control," generally by forcing the immediate execution of a handler procedure, no matter what your process was doing.

When a hardware device wants to signal that something needs attention, or has gone wrong, we say that the device triggers an interrupt. Linux generalizes this and views all forms of exceptions as being like interrupts.

Once this occurs, we can "handle" the exception in ways that might hide it, or we may need to stop some task entirely (like with ^C).

BIGGEST CONCERN

An exception can occur in the middle of some sort of expression evaluation, or data structure update.

For example, if your code manages a linked list, the exception could occur in the middle of adding a node!

So… the handler cannot assume that data structures are intact!

HOW WE HANDLE THIS

We think in terms of “recoverable” exceptions and “non-recoverable” ones.

A recoverable exception occurs if the kernel or the program can handle the exception, then resume normal execution.

A non-recoverable exception terminates the task (or perhaps just part of some task).

LET'S LOOK FIRST AT MECHANISMS, BUT THEN WE WILL SEE AN ABSTRACTION EMERGE

A mechanistic perspective looks at how each class of event arises. Each form of abnormal control flow has a concrete cause

Because the hardware features are diverse, we could end up with a diverse set of language features to deal with them.

In practice, there is a surprisingly degree of uniformity representing one abstraction that is applies in various ways

THIS ILLUSTRATES CONCEPTUAL ABSTRACTION

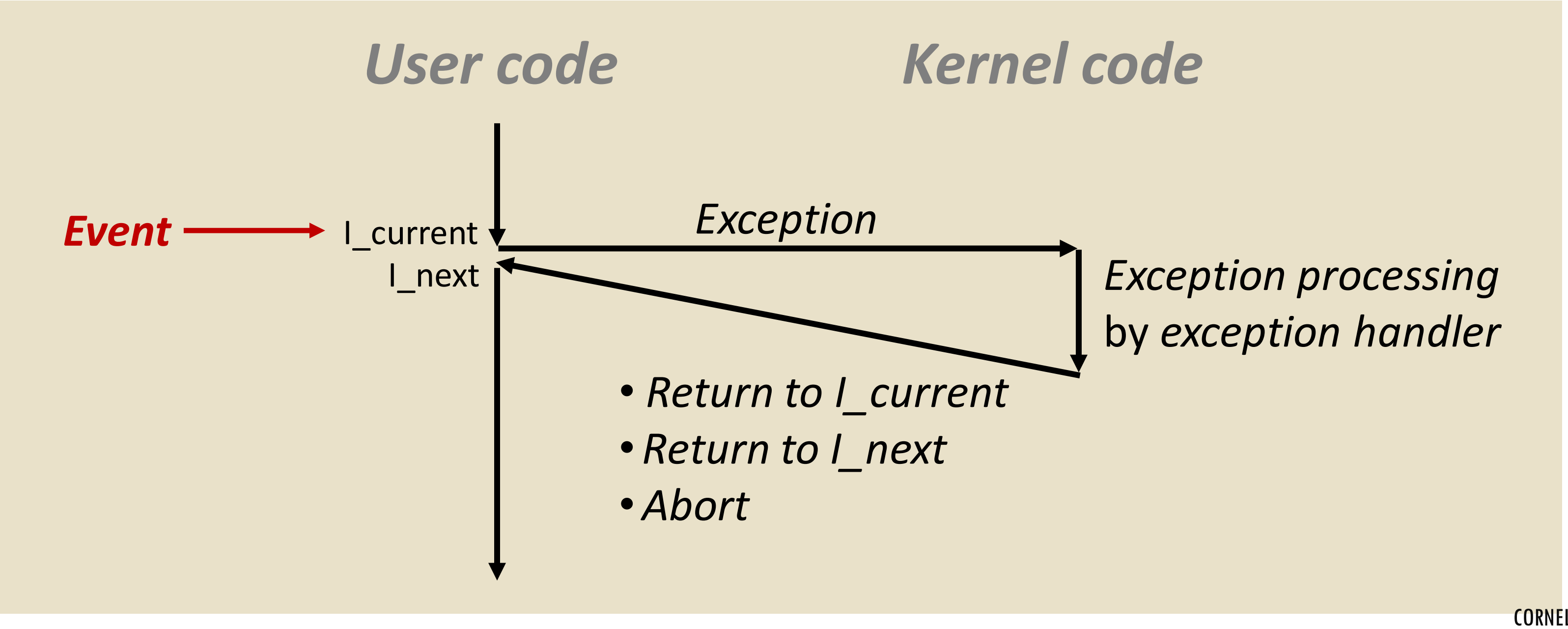
Rather than abstracting storage, the way a file system abstracts the storage blocks on a device, control flow abstractions have a conceptual flavor.

They illustrate a *reused design pattern* and a *way of thinking about abnormal control flow*. This concept is universal, yet the embodiment varies.

THIS DESIGN PATTERN IS A LINUX FEATURE

An exception often causes a transfer of control to the OS kernel in response to some event (i.e., change in processor state)

- Examples: Divide by 0, arithmetic overflow, page fault, I/O request completes, typing Ctrl-C

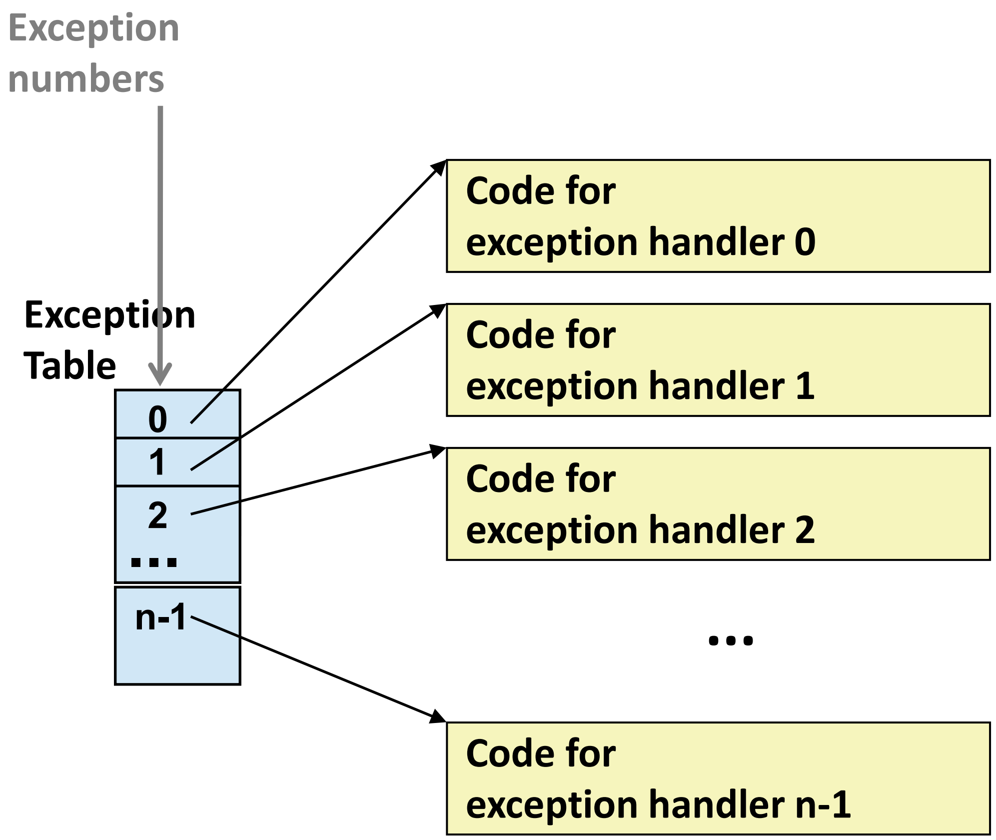


EXCEPTION TABLES

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



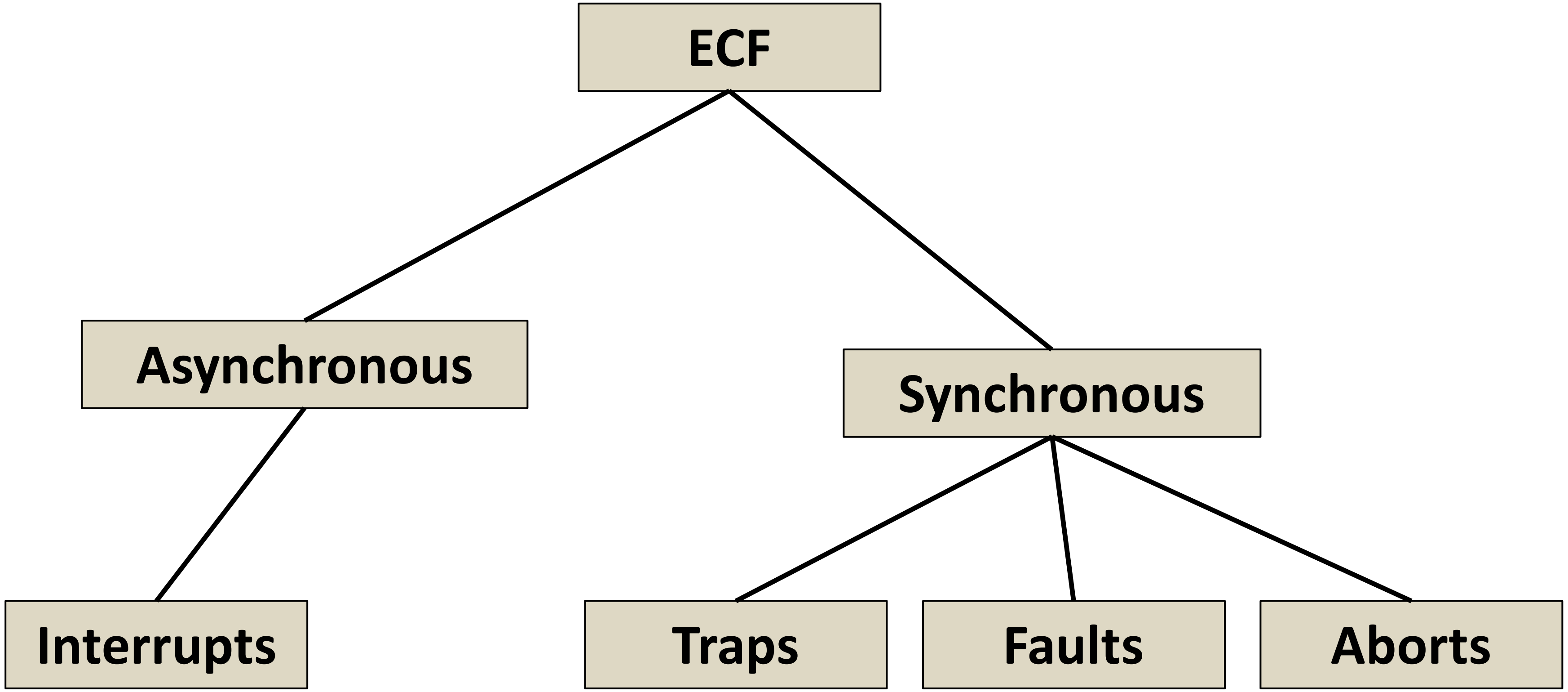
EXCEPTION TABLES

The kernel has one for interrupts.

Each process has one for signals.

The entries are simply the addresses of the handler methods. A special exception handler turns the exception into a kind of procedure call, at which the handler runs like normal code.

(PARTIAL) TAXONOMY



ASYNCHRONOUS EXCEPTIONS (INTERRUPTS)

Caused by events external to the processor

- Indicated by setting the processor's interrupt pin
- Handler returns to the instruction that was about to execute

Examples:

- Timer interrupt
  - Every few ms, an external timer chip triggers an interrupt.
  - Used by the kernel to take back control from user programs
- I/O interrupt from external device
  - Typing a character or hitting Ctrl-C at the keyboard
  - Arrival of a packet from a network, or data from a disk

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. Control resumes at "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… Aborts current program
  - Examples: illegal instruction, parity error, machine check

SYSTEM CALLS

- Each Linux system call has a unique ID number
- Examples:

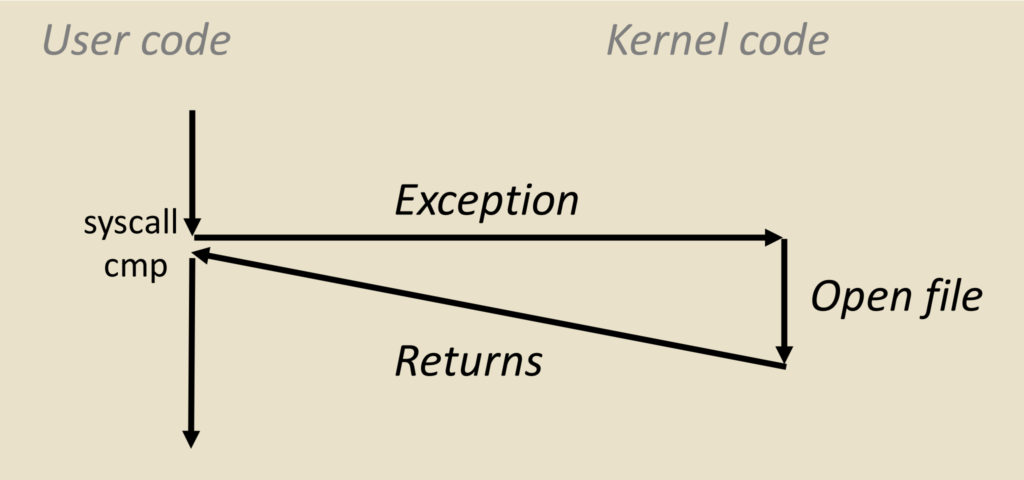
Number	*Name*	*Description*
0	`read`	Read file
1	`write`	Write file
2	`open`	Open file
3	`close`	Close file
4	`stat`	Get info about file
57	`fork`	Create process
59	`execve`	Execute a program
60	`_exit`	Terminate process
62	`kill`	Send signal to process

SYSTEM CALL EXAMPLE: OPENING FILE

User calls: `open(filename, options)`

Calls `__open` function, which invokes system call instruction `syscall`

```
00000000000e5d70 <__open>:
...
e5d79:   b8 02 00 00 00      mov  $0x2,%eax  # open is syscall #2
e5d7e:   0f 05               syscall         # Return value in %rax
e5d80:   48 3d 01 f0 ff ff   cmp  $0xfffffffffffff001,%rax
...
e5dfa:   c3                  retq
```



- `%rax` contains syscall number
- Other arguments in `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9`
- Return value in `%rax`
- Negative value is an error corresponding to negative `errno`

SYSTEM CA

User calls: `open(fil`

Calls `__open` function,

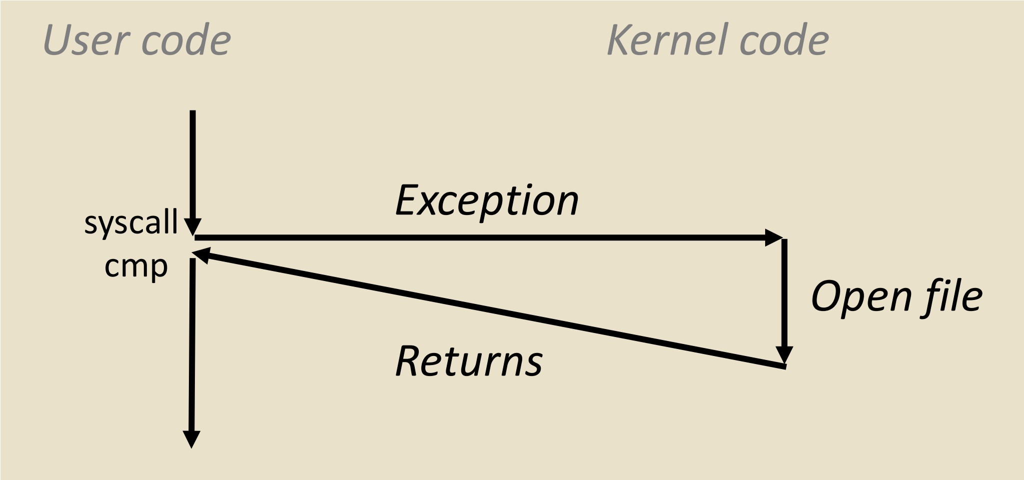
```
00000000000e5d70 <__op
...
e5d79:   b8 02 00 00 00
e5d7e:   0f 05              sysca
e5d80:   48 3d 01 f0 ff ff   c
...
e5dfa:   c3                retq
```

Almost like a function call

- Transfer of control
- On return, executes next instruction
- Passes arguments using calling convention
- Gets result in `%rax`

One Important exception!

- Executed by Kernel
- Different set of privileges
- And other differences:
  - e.g., “address” of “function” is in `%rax`
  - Uses `errno`
  - Etc.



- `%rax` contains syscall number
- Other arguments in `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9`
- Return value in `%rax`
- Negative value is an error corresponding to negative `errno`

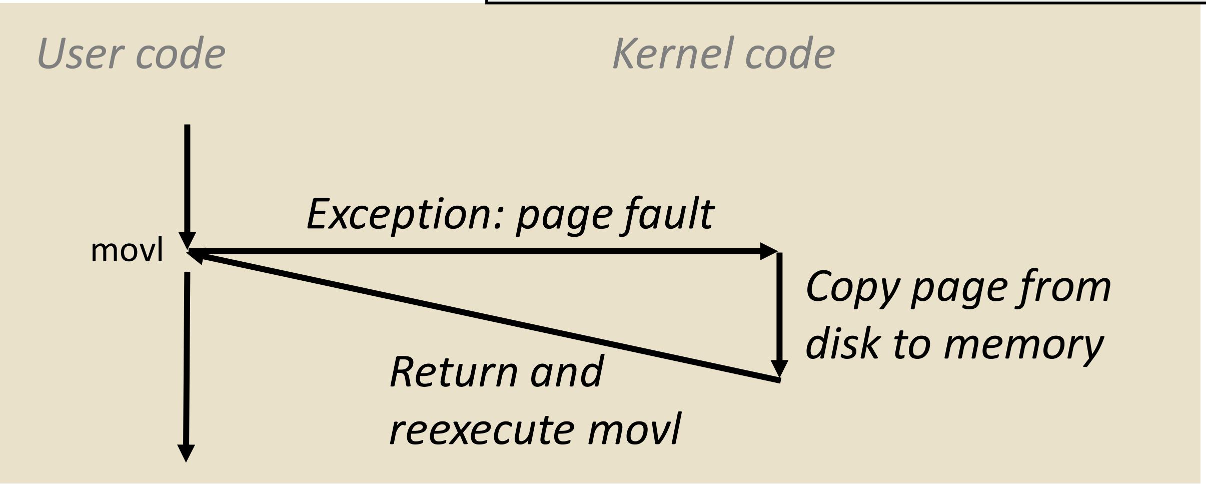
FAULT EXAMPLE: PAGE FAULT

User writes to memory location

That portion (page) of user's memory is currently paged out (on disk)

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

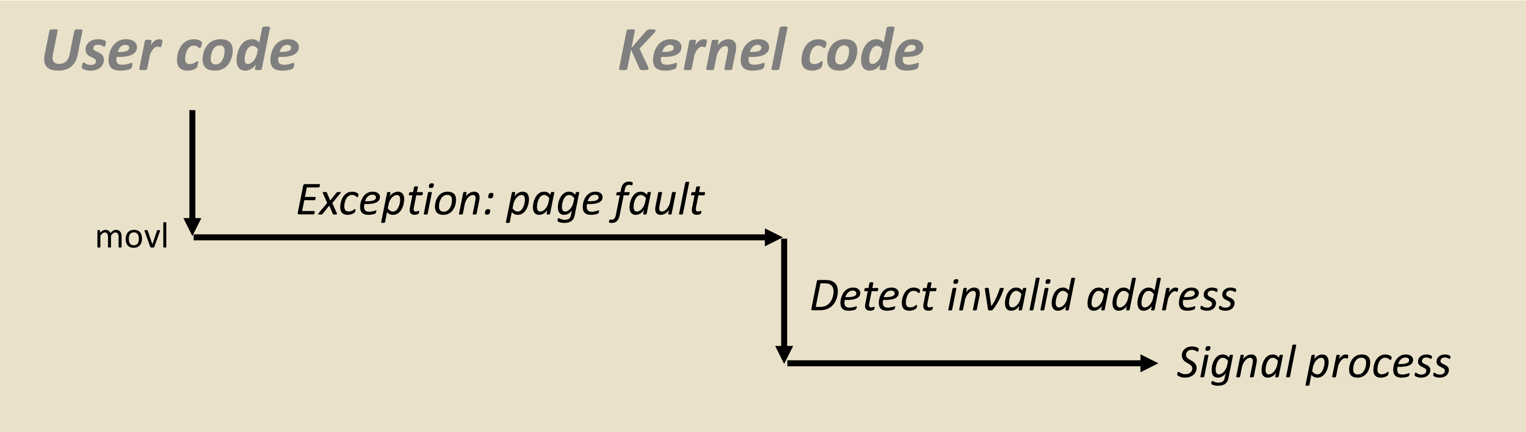
```
80483b7:   c7 05 10 9d 04 08 0d   movl   $0xd,0x8049d10
```



FAULT EXAMPLE: INVALID MEMORY REFERENCE

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

```
80483b7:   c7 05 10 9d 04 08 0d   movl   $0xd,0x8049d10
```



Sends SIGSEGV signal to user process

User process exits with "segmentation fault"

SOME FLAVORS OF SEGMENT FAULTS

Trying to read or write into memory that isn't part of your address space.

Trying to modify a write-protected data or code segment.

Trying to jump into (execute) a data segment (this is actually possible, but you have to do something special).

YET EXCEPTIONS ALSO ALLOW US TO EMULATE “INFINITE NUMBER OF CORES”

Basic idea: if we have more threads than cores, we can use timer exceptions to switch from thread to thread (or process to process)

This is called a “context switch” and involves saving the state of the interrupted thread: the contents of the registers.

Then we can load the state of the thread we wish to switch to.

CONTEXT SWITCHES BETWEEN PROCESSES

For the hardware, a process is simply a set of threads plus a memory map that tells which memory pages belong to the process, and what protection rules to apply.

As part of the context switch, the kernel simply tells the hardware which "page table" to use for this process.

TODAY

Exceptional Control Flow

Linux signals

Programming language-level exceptions

C++ features for handling exceptions

LINUX SIGNALS

Linux uses a variety of signals to "tell" an active process about exceptions relevant to it. The approach mimics what the hardware does for interrupts.

The signal must be caught or ignored. Some signals are ignored by default. Others must be caught and will terminate the process if not.

To catch a signal, a process (or some library it uses) must register a "signal handler" procedure. Linux will pause normal execution and call the handler. When the handler returns, the interrupted logic resumes.

LIST OF LINUX SIGNALS

SIGABRT	Abort signal from abort(3)
SIGALRM	Timer signal from alarm(2)
SIGBUS	Bus error (bad memory access)
SIGCHLD	Child stopped or terminated
SIGCONT	Continue if stopped
SIGEMT	Emulator trap
SIGFPE	Floating-point exception
SIGHUP	User logged out or controlling process terminated
SIGILL	Illegal Instruction
SIGINFO	A synonym for SIGPWR
SIGINT	Interrupt from keyboard
SIGIO	I/O now possible (4.2BSD)
SIGIOT	IOT trap. A synonym for SIGABRT
SIGKILL	Kill signal (cannot be caught or ignored)
SIGLOST	File lock lost (unused)
SIGPIPE	Broken pipe: write to pipe with no readers
SIGPOLL	Pollable event (Sys V); synonym for SIGIO
SIGPROF	Profiling timer expired
SIGPWR	Power failure (System V)
SIGQUIT	Quit from keyboard
SIGSEGV	Invalid memory reference
SIGSTOP	Stop process
SIGTSTP	Stop typed at terminal
SIGSYS	Bad system call (SVr4)
SIGTERM	Termination signal
SIGTRAP	Trace/breakpoint trap
SIGTTIN	Terminal input for background process
SIGTTOU	Terminal output for background process
SIGURG	Urgent condition on socket (4.2BSD)
SIGUSR1	User-defined signal 1
SIGUSR2	User-defined signal 2
SIGVTALRM	Virtual alarm clock (4.2BSD)
SIGXCPU	CPU time limit exceeded (4.2BSD)
SIGXFSZ	File size limit exceeded (4.2BSD)
SIGWINCH	Window resize signal (4.3BSD, Sun)

GDB – LINUX DEBUGGER

Allows you to understand where an exception occurred.

You can set breakpoints, examine variables, see the call stack

You can even watch individual variables

Uses exception handlers for all of this!

PAUSE FOR A DEMO: LETS SEE WHAT HAPPENS IF A PROGRAM TRIES TO ACCESS MEMORY INAPPROPRIATELY, AND HOW GDB HELPS US TRACK SUCH AN ISSUE DOWN.

GDB cheatsheet - page 1

Running

gdb *<program> [core dump]*
Start GDB (with optional core dump).

gdb --args *<program> <args…>*
Start GDB and pass arguments

gdb --pid *<pid>*
Start GDB and attach to process.

set args *<args...>*
Set arguments to pass to program to be debugged.

run
Run the program to be debugged.

kill
Kill the running program.

Breakpoints

break *<where>*
Set a new breakpoint.

delete *<breakpoint#>*
Remove a breakpoint.

clear
Delete all breakpoints.

enable *<breakpoint#>*
Enable a disabled breakpoint.

disable *<breakpoint#>*
Disable a breakpoint.

Watchpoints

watch *<where>*
Set a new watchpoint.

delete/enable/disable *<watchpoint#>*
Like breakpoints.

<where>

function_name
Break/watch the named function.

line_number
Break/watch the line number in the current source file.

file:line_number
Break/watch the line number in the named source file.

Conditions

break/watch *<where>* if *<condition>*
Break/watch at the given location if the condition is met.
Conditions may be almost any C expression that evaluate to true or false.

condition *<breakpoint#> <condition>*
Set/change the condition of an existing break- or watchpoint.

Examining the stack

backtrace
where
Show call stack.

backtrace full
where full
Show call stack, also print the local variables in each frame.

frame *<frame#>*
Select the stack frame to operate on.

Stepping

step
Go to next instruction (source line), diving into function.

next
Go to next instruction (source line) but don‘t dive into functions.

finish
Continue until the current function returns.

continue
Continue normal execution.

Variables and memory

print/*format <what>*
Print content of variable/memory location/register.

display/*format <what>*
Like „print“, but print the information after each stepping instruction.

undisplay *<display#>*
Remove the „display“ with the given number.

enable display *<display#>*
disable display *<display#>*
En- or disable the „display“ with the given number.

x/*nfu <address>*
Print memory.
n: How many units to print (default 1).
f*:* Format character (like „print“).
u: Unit.

Unit is one of:

- *b*: Byte,
- *h*: Half-word (two bytes)
- *w*: Word (four bytes)
- *g*: Giant word (eight bytes)).

GDB cheatsheet - page 2

Format

a Pointer.
c Read as integer, print as character.
d Integer, signed decimal.
f Floating point number.
o Integer, print as octal.
s Try to treat as C string.
t Integer, print as binary (*t* = „two“).
u Integer, unsigned decimal.
x Integer, print as hexadecimal.

<what>

expression
: Almost any C expression, including function calls (must be prefixed with a cast to tell GDB the return value type).

file_name::*variable_name*
: Content of the variable defined in the named file (static variables).

function::*variable_name*
: Content of the variable defined in the named function (if on the stack).

{type}address
: Content at *address*, interpreted as being of the C type *type*.

$register
: Content of named register. Interesting registers are $esp (stack pointer), $ebp (frame pointer) and $eip (instruction pointer).

Threads

thread *<thread#>*
: Chose thread to operate on.

Manipulating the program

set var *<variable_name>=<value>*
: Change the content of a variable to the given value.

return *<expression>*
: Force the current function to return immediately, passing the given value.

Sources

directory *<directory>*
: Add *directory* to the list of directories that is searched for sources.

list
list *<filename>*:<function>
list *<filename>:<line_number>*
list *<first>,<last>*
: Shows the current or given source context. The *filename* may be omitted. If *last* is omitted the context starting at *start* is printed instead of centered around it.

set listsize *<count>*
: Set how many lines to show in „list“.

Signals

handle *<signal> <options>*
: Set how to handle signles. Options are:

(no)print: (Don‘t) print a message when signals occurs.

(no)stop: (Don‘t) stop the program when signals occurs.

(no)pass: (Don‘t) pass the signal to the program.

Informations

disassemble
disassemble *<where>*
: Disassemble the current function or given location.

info args
: Print the arguments to the function of the current stack frame.

info breakpoints
: Print informations about the break- and watchpoints.

info display
: Print informations about the „displays“.

info locals
: Print the local variables in the currently selected stack frame.

info sharedlibrary
: List loaded shared libraries.

info signals
: List all signals and how they are currently handled.

info threads
: List all threads.

show directories
: Print all directories in which GDB searches for source files.

show listsize
: Print how many are shown in the „list“ command.

whatis *variable_name*
: Print type of named variable.

TODAY

Exceptional Control Flow

Linux signals

Programming language-level exceptions

C++ features for handling exceptions

UNHANDLED SEGMENTATION FAULTS

Our program dereferenced a null pointer, causing a segmentation fault. gdb showed us the line and variable responsible for the crash.

Notice the contrast with the cases where Linux was able to handle the fault: page faults and stack faults… in those, the program hadn't done anything wrong... The instruction that caused the fault can be retried (and will succeed) once the new page is mapped in.

With a segmentation fault, there is no way to "repair" the issue.

WHAT CAN WE DO?

Segmentation faults terminate the process.

But you could also “imagine” catching them and just terminating some thread that triggered the fault.

Other kinds of exceptions might be user-designed ones intended to reflect program logic, like “divide by 0” in Bignum

... LEADING TO

The C++ concept of a “thrown” exception, and try/catch

We use this feature to manage many kinds of exceptions that we anticipated and want to handle in code

But it can be a bit tricky to get this right without leaking memory or other kinds of resources, as we will see next

TODAY

Exceptional Control Flow

Linux signals

Programming language-level exceptions

C++ features for handling exceptions

EXCEPTIONS AT THE LANGUAGE LEVEL

Many programming languages have features to help you manage exceptions.

For Linux signals, this is done purely through library procedures that register the desired handler method.

But for program exceptions, a program might halt, or there may be a way to manage the exception and resume execution.

One big difference: Linux can restart a program at the exact instruction and in the exact state it was in prior to an interrupt or signal. But a programming language generally can't resume the same instruction after an event like a zero divide, so we need a way to transfer control to "alternative logic"

WHAT CAN WE DO IF A FAULT MIGHT OCCUR, BUT CAN BE HANDLED?

Most languages, including C++, offer a way to attempt some action, but then "catch" exceptions that might occur.

As part of these mechanisms the application is given a way to "throw" an exception if the logic detects a problem.

C++ CONSTRUCT

```
try
{
        do_something…
}
catch (exception-type)              //  Something went wrong!
{
        handler for exception        //  "Fix" the issue (or report it)
}
```

C++ CONSTRUCT

```
try
{
        salaries[employee] *= 1.05;// Give a raise…
}
catch (EmployeeUnknown)              // "Employee unknown"
{
        handler for exception         //  Print an error msg
}
```

"DO_SOMETHING" WON'T BE RETRIED

When Linux handled a page fault, it restarted the program on the same instruction and in the same state as it had at the fault.

When C++ catches this "not found" error and prints the error message, we just continue with the next line of code.

A COMMON ISSUE THIS CAN RAISE

Suppose that your program was working with a resource such as an open file, or was holding a lock (we'll discuss locks soon…)

The try/catch can jump to a caller, exiting from one or more code blocks and method calls that were active.

Thus the resource could be left "dangling", causing memory leaks or open files or other potential problems.

VISUALIZING THIS ISSUE

```
void annual_sip(float standard_raise)
{
	for(auto emp: emp_list)
	{
		try
		{
			give_raise(emp.name, .05);
		}
		catch(EmployeeNotFound)
		{
			cout << "Salary DB is missing an employee!" << endl;
		}
	}
```

```
void give_raise(char* name, float raise)
{
	FILE *fp = fopen("Paychecks.dat");
	salaries[name] *= 1.0 + raise;
	…. write a record in the paychecks file…
	fclose(fp);
}
```

VISUALIZING THIS ISSUE

```
void annual_sip(float standard_raise)
{
    for(auto emp: emp_list)
    {
        try
        {
            give_raise(emp.name, .05);
        }
        catch(EmployeeNotFound)
        {
            cout << "Salary DB is missing an employee!" << endl;
        }
    }
```

If this employee is not in the salaries database, exception is thrown here.

```
void give_raise(char* name, float raise)
{
    FILE *fp = fopen("Paychecks.dat");
    salaries[name] *= 1.0 + raise;
    …. write a record in the paychecks file…
    fclose(fp);
}
```

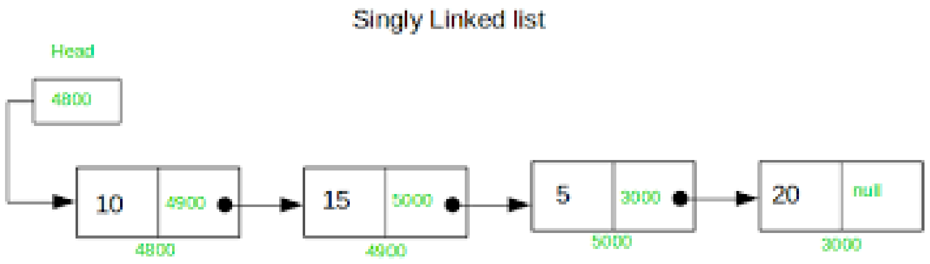
VISUALIZING THIS ISSUE

```
void annual_sip(float standard_raise)
{
        for(auto emp: emp_list)
        {
                catch(EmployeeNotFound)
                {
                        cout << “Salary DB is missing an employee!” << endl;
                }
        }
```

The exception transfers control to the catch block in annual_sip. The stack frame of give_raise is released. But this means that the line that calls fclose will never execute, so we “leak” open files!

```
void give_raise(char* name, float raise)
{
        FILE *fp = fopen(“Paychecks.dat”);
        salaries[name] *= 1.0 + raise;
        …. write a record in the paychecks file…
        fclose(fp);
}
```

LINKED LIST EXAMPLE



Suppose that your code is adding a node in a linked list. Now the exception handler tries to access that list data structure.

The list might sometimes "seem to be broken" because not all the pointers will have their correct values!

Any data that your program *updates* could be seen during the update, rather than just before or after!

EXCEPTIONS RUN A RISK OF BUGS!

If an exception handler were to look at this list while it was changing, it could crash! Similarly, an exception handler can't allocate new memory objects, or print a message – all of those could be unsafe at some random moment when the handler runs!

Solution? Sometimes you can temporarily disable exception handling. Additionally, it is always best for exceptional handlers to be short, self-contained, and to not invoke library methods!

THAT ISSUE WON'T ARISE WITH C++ CATCH

A throw/catch sequence won't resume the code that threw the exception.

Moreover, in C++ we will have run the destructors for all stack allocated objects that went out of scope before running catch.

This gives a very predicable, controlled behavior

COULD C++ THROW/CATCH REPLACE SIGNALS?

It may seem natural to think about using throw/catch as a signal replacement, but this won't work.

The problem is that a signal is asynchronous and unpredictable. With throw/catch the exception is synchronous and usually involves a software "choice" to throw the exception.

This is a shame, in fact, because it is so hard to write safe signal handlers.

SUMMARY

The exception pattern is very widely seen in Linux and C++. Broadly, exception handling mimics hardware interrupts. But hardware interrupts and signals can be "inhibited".

C++ try/catch control flow can't be inhibited and can easily disrupt updates and resource management: a potential source of serious bugs.

Per-resource wrappers offer an elegant solution.