```
typedef struct {
    int             mode;
    ScePspFVector3  pos;
    ScePspFVector3  sbuf[3];
    int             scnt;
    float           t;
} BALLOONDAT;

static BALLOONDAT      balloon;
static ScePspFVector3  sphere[20];
static ScePspFVector3  pole[20];

extern void DrawSphere(ScePspFVector3 *array,float r);
extern void DrawPole(ScePspFVector3 *array,float r);

void init_balloon(void)
{
        int     i;

        balloon.mode=MODE
        balloon.pos.x= 0.
        balloon.pos.y=-8.
        balloon.pos.z= 0.
        balloon.t=0.0f;
        balloon.scnt=2;

        for (i=0; i<3; i
                balloon.sbuf[i].x=RAN
                balloon.sbuf[i].y=RANGERAND(0.0f, 0.01f,
                balloon.sbuf[i].z=RANGERAND(0.0f,
        }
}
void draw_balloon(void)
{
        ScePspFVector3  vec;
```

Operating Systems and C

8. Exceptional Control Flow

Preface: Why am I teaching this lecture?

a) Willard is travelling.

b) Part of doing a PhD is learning how to teach.

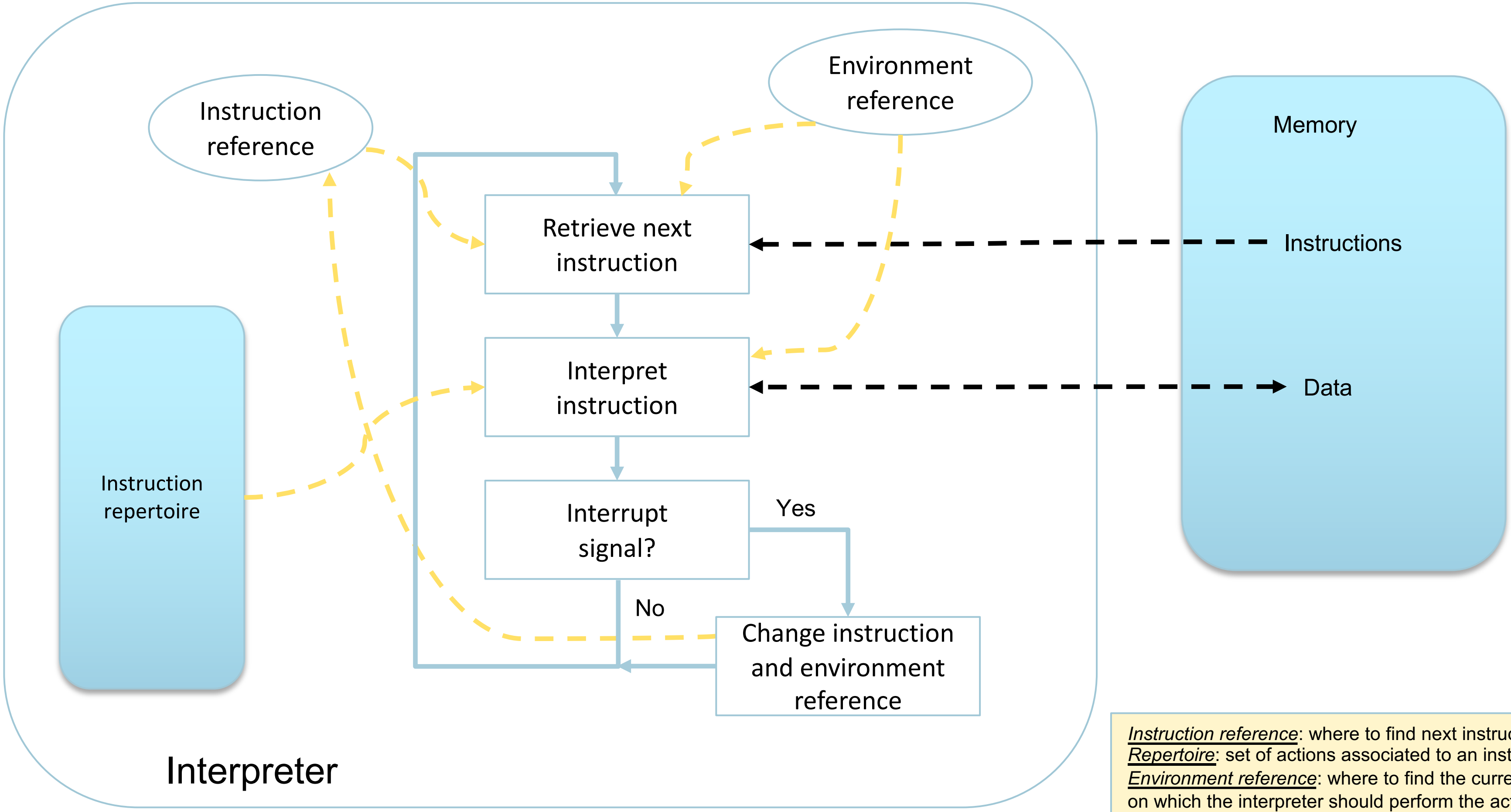
- My “teaching course” was during a lockdown, meaning that my first lecture was on Zoom ☹
- Zoom lectures are very different from physical lectures, so I requested to teach a physical course.
- Since this is my first physical lecture, feedback (both good and bad) is greatly appreciated.
  - You can approach me after the lecture, send me an e-mail or write on slack.

Outline

- Interrupt Handling
- Process Management
- Signals

Interpreter Abstraction

Source: Saltzer and Kaashoek



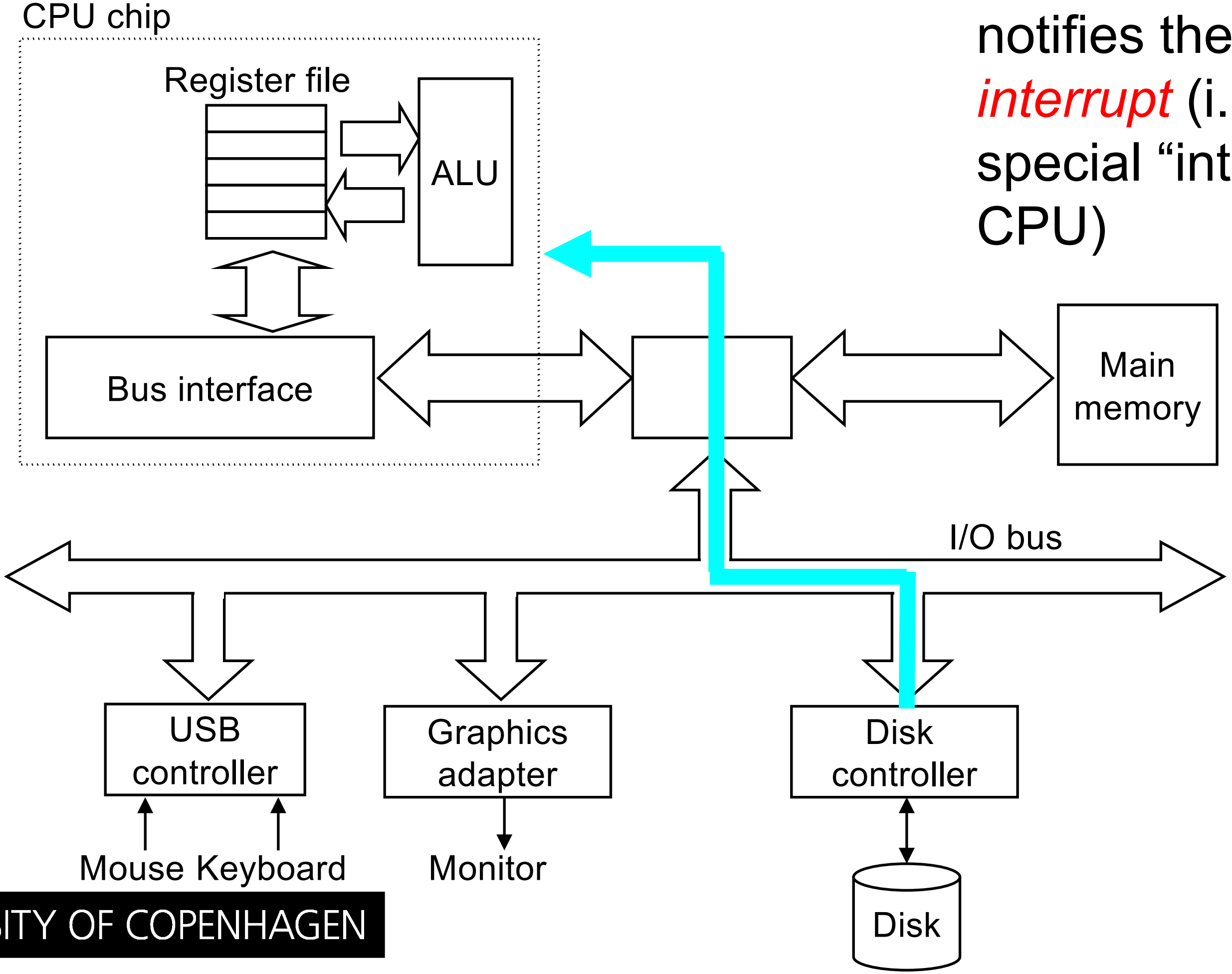
Instruction reference: where to find next instruction
Repertoire: set of actions associated to an instruction
Environment reference: where to find the current state on which the interpreter should perform the actions of the current instruction

Low vs. High Level Interrupts

- **Low level Interrupt mechanisms**
  - Exceptions: change in control flow in response to a system event (i.e., change in system state)
  - Combination of hardware and OS software
- **Higher level Interrupt mechanisms**
  - Process context switch
  - Signals
  - Nonlocal jumps: setjmp()/longjmp()
  - Implemented by either:
    - OS software (context switch and signals)
    - C language runtime library (nonlocal jumps)

Reading a Disk Sector (3)

When the DMA transfer completes, the disk controller notifies the CPU with an *interrupt* (i.e., asserts a special “interrupt” pin on the CPU)

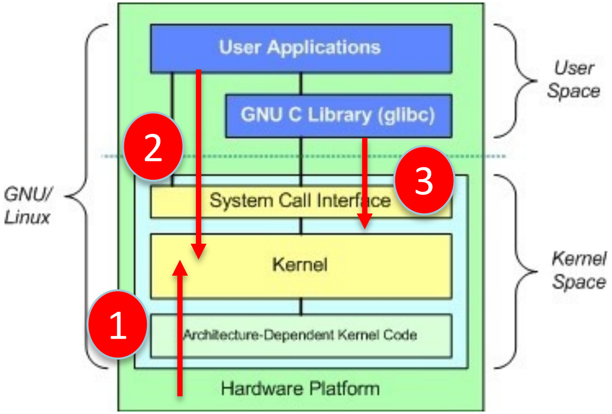


Exceptional Control Flow

Interruption event:

1. A device needs attention
2. The user program did something illegal
3. The user program asks the OS kernel for a service through a system call

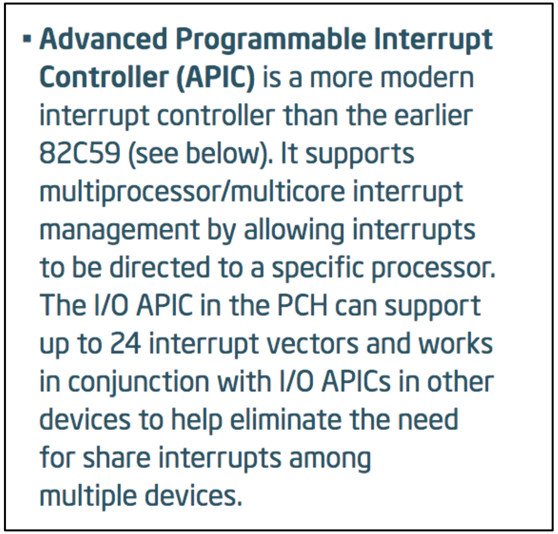
In these cases, **the flow of control is transferred from the user program to the OS kernel**.



Asynchronous Events

Managed in hardware

- Caused by events external to the processor
  - Indicated by setting the processor’s interrupt pin
  - **Handler returns to “next” instruction**
- Examples:
  - I/O interrupts
    - hitting Ctrl-C at the keyboard
    - arrival of a packet from a network
    - arrival of data from a disk
  - Hard reset interrupt
    - hitting the reset button
  - Soft reset interrupt
    - hitting Ctrl-Alt-Delete on a PC



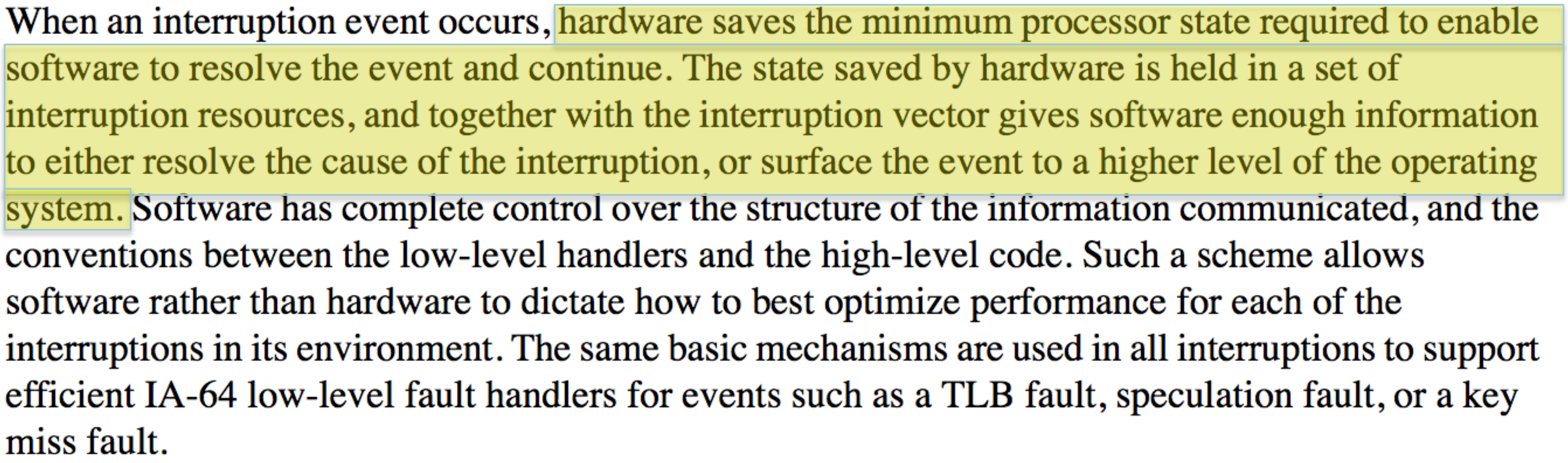
- **Advanced Programmable Interrupt Controller (APIC)** is a more modern interrupt controller than the earlier 82C59 (see below). It supports multiprocessor/multicore interrupt management by allowing interrupts to be directed to a specific processor. The I/O APIC in the PCH can support up to 24 interrupt vectors and works in conjunction with I/O APICs in other devices to help eliminate the need for share interrupts among multiple devices.

PCH: Platform Controller Hub

https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-introduction-basics-paper.pdf

Asynchronous Events

Hardware hands it out to software

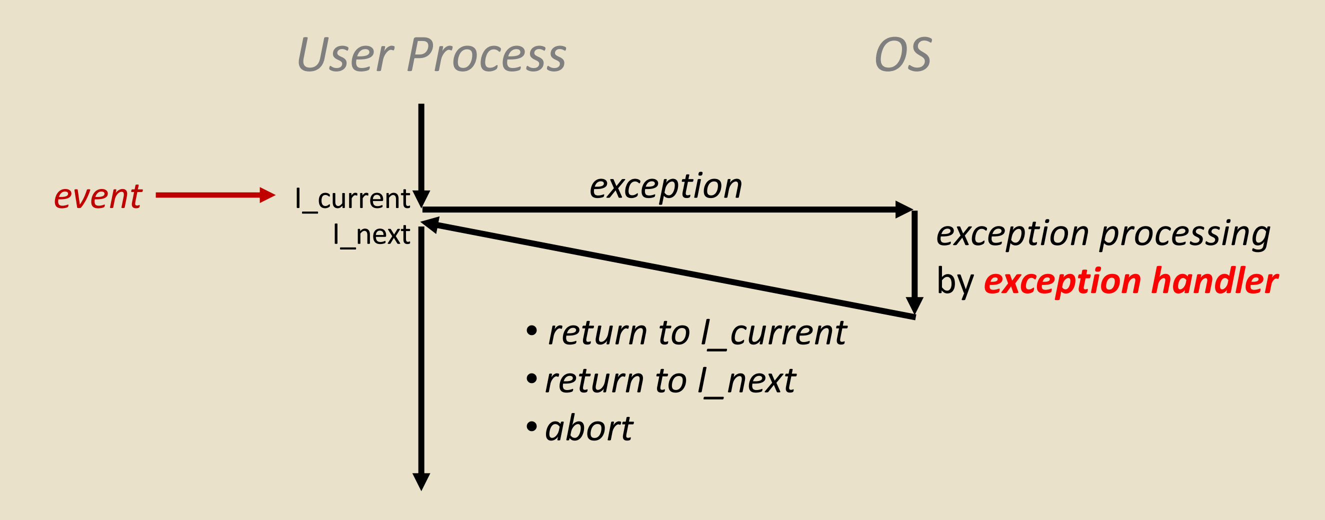


When an interruption event occurs, hardware saves the minimum processor state required to enable software to resolve the event and continue. The state saved by hardware is held in a set of interruption resources, and together with the interruption vector gives software enough information to either resolve the cause of the interruption, or surface the event to a higher level of the operating system. Software has complete control over the structure of the information communicated, and the conventions between the low-level handlers and the high-level code. Such a scheme allows software rather than hardware to dictate how to best optimize performance for each of the interruptions in its environment. The same basic mechanisms are used in all interruptions to support efficient IA-64 low-level fault handlers for events such as a TLB fault, speculation fault, or a key miss fault.

http://refspecs.linux-foundation.org/IA64-softdevman-vol2.pdf

Asynchronous Events: Exceptions

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

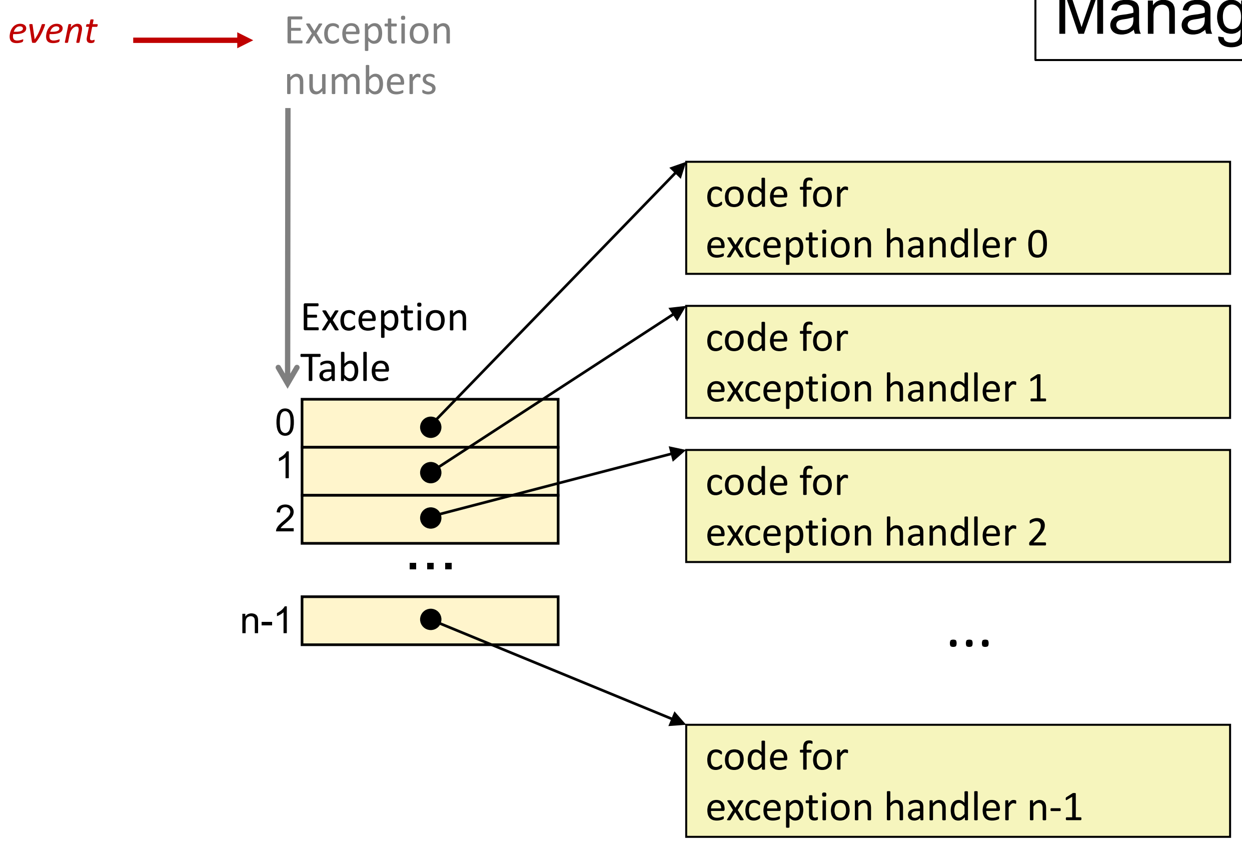


Examples:

div by 0, arithmetic overflow, page fault, I/O request completes, Ctrl-C

Exception Handler & Interrupt Vector

Managed in hardware



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

Interrupt Vector

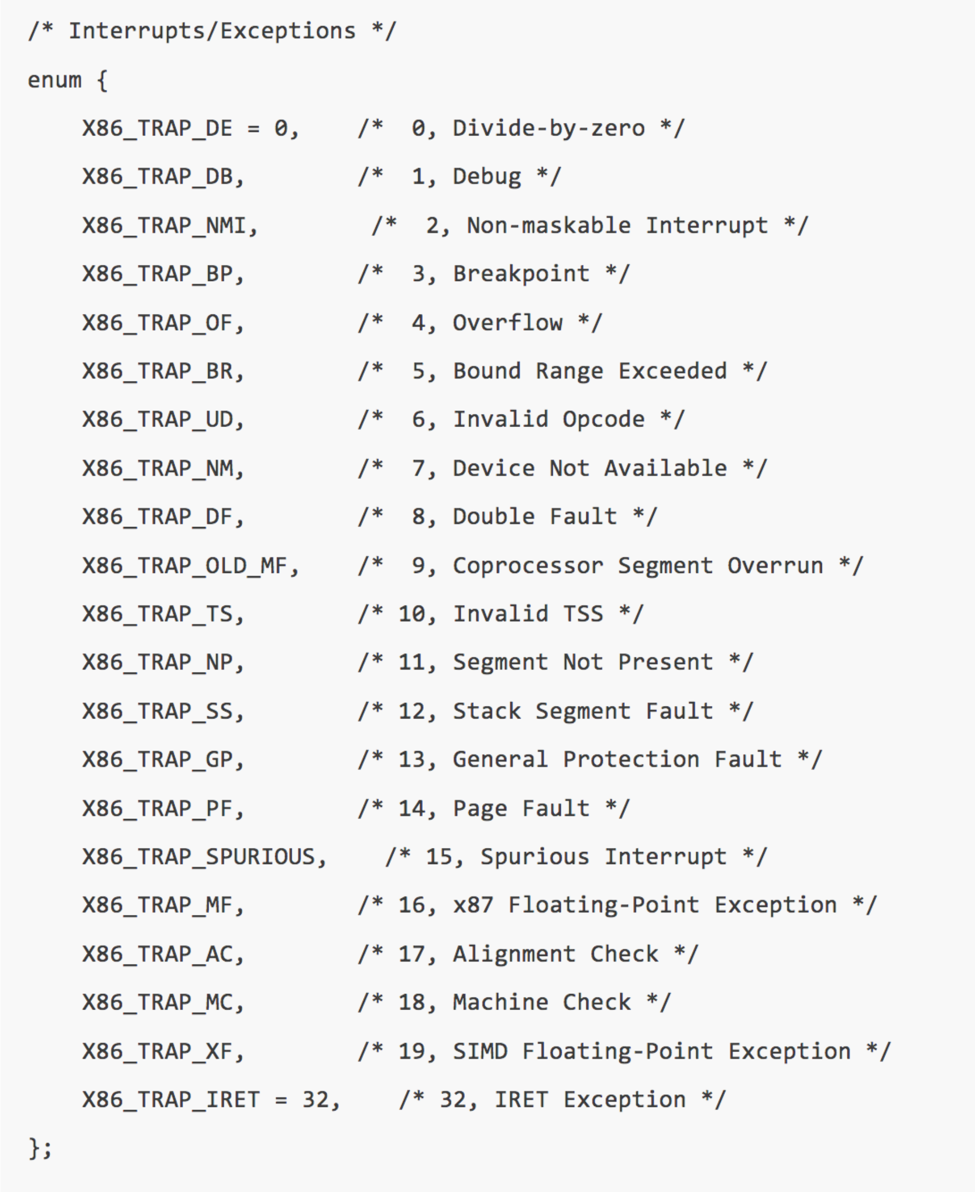
arch/x86/include/asm/traps.h

First 32 slots in interrupt vectors reserved (x86).

33-127: OS-defined

128 (0x80): system calls

129-255: OS-defined



```
/* Interrupts/Exceptions */
enum {
    X86_TRAP_DE = 0,        /*  0, Divide-by-zero */
    X86_TRAP_DB,            /*  1, Debug */
    X86_TRAP_NMI,            /*  2, Non-maskable Interrupt */
    X86_TRAP_BP,            /*  3, Breakpoint */
    X86_TRAP_OF,            /*  4, Overflow */
    X86_TRAP_BR,            /*  5, Bound Range Exceeded */
    X86_TRAP_UD,            /*  6, Invalid Opcode */
    X86_TRAP_NM,            /*  7, Device Not Available */
    X86_TRAP_DF,            /*  8, Double Fault */
    X86_TRAP_OLD_MF,        /*  9, Coprocessor Segment Overrun */
    X86_TRAP_TS,            /* 10, Invalid TSS */
    X86_TRAP_NP,            /* 11, Segment Not Present */
    X86_TRAP_SS,            /* 12, Stack Segment Fault */
    X86_TRAP_GP,            /* 13, General Protection Fault */
    X86_TRAP_PF,            /* 14, Page Fault */
    X86_TRAP_SPURIOUS,        /* 15, Spurious Interrupt */
    X86_TRAP_MF,            /* 16, x87 Floating-Point Exception */
    X86_TRAP_AC,            /* 17, Alignment Check */
    X86_TRAP_MC,            /* 18, Machine Check */
    X86_TRAP_XF,            /* 19, SIMD Floating-Point Exception */
    X86_TRAP_IRET = 32,        /* 32, IRET Exception */
};
```

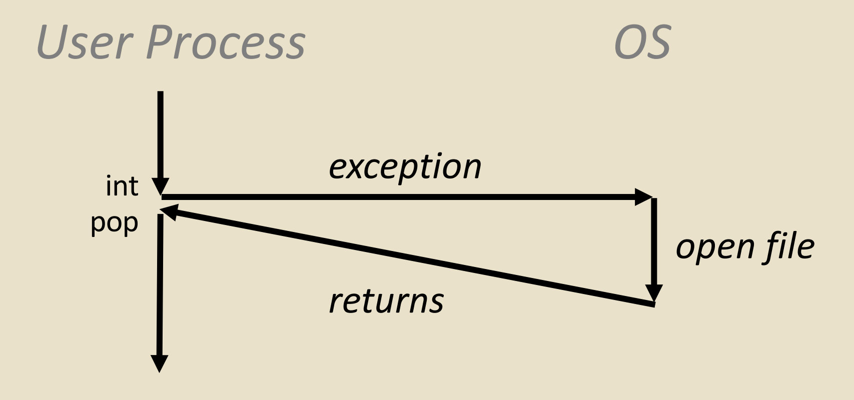
Synchronous Events

- Caused by events that occur as a result of executing an instruction:
  - ***Traps***
    - Intentional
    - Examples: ***system calls***, breakpoint traps, special instructions
    - **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: parity error, machine check
    - **Aborts current program**

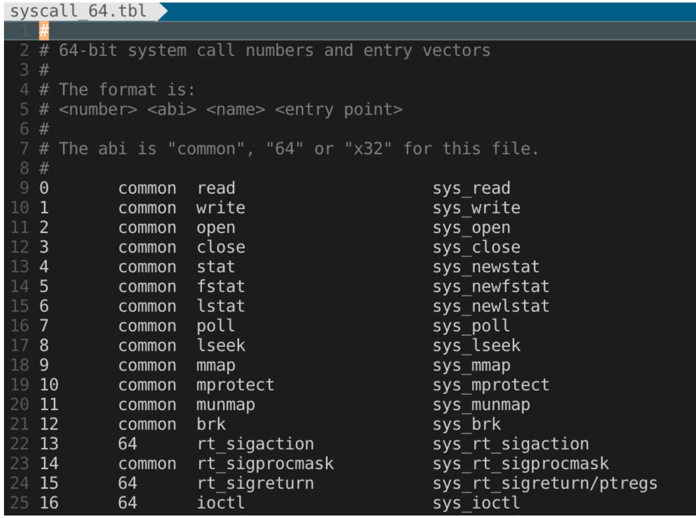
Trap Example: System call

- User calls: `open(filename, options)`
- Function `open` executes **system call instruction `int`**

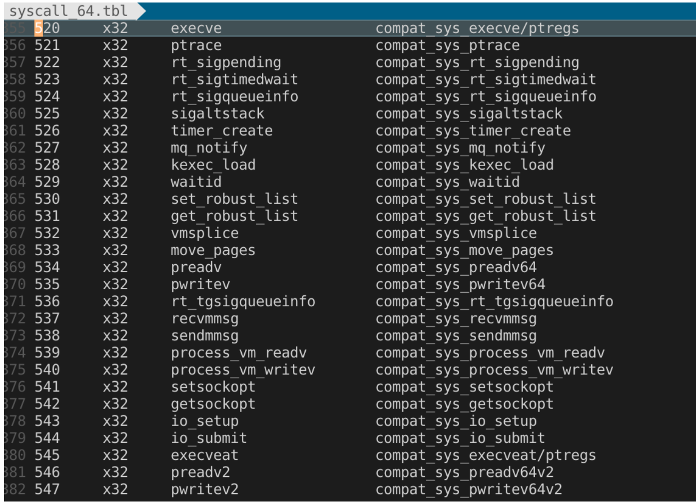
```
0804d070 <__libc_open>:
 . . .
 804d082:      cd 80                    int     $0x80
 804d084:      5b                       pop     %ebx
 . . .
```



- OS must find or create file, get it ready for reading or writing
- Returns integer file descriptor



```
#
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The abi is "common", "64" or "x32" for this file.
#
0       common  read                    sys_read
1       common  write                   sys_write
2       common  open                    sys_open
3       common  close                   sys_close
4       common  stat                    sys_newstat
5       common  fstat                   sys_newfstat
6       common  lstat                   sys_newlstat
7       common  poll                    sys_poll
8       common  lseek                   sys_lseek
9       common  mmap                    sys_mmap
10      common  mprotect                sys_mprotect
11      common  munmap                  sys_munmap
12      common  brk                     sys_brk
13      64      rt_sigaction            sys_rt_sigaction
14      common  rt_sigprocmask          sys_rt_sigprocmask
15      64      rt_sigreturn            sys_rt_sigreturn/ptregs
16      64      ioctl                   sys_ioctl
```



```
520     x32     execve                  compat_sys_execve/ptregs
521     x32     ptrace                  compat_sys_ptrace
522     x32     rt_sigpending           compat_sys_rt_sigpending
523     x32     rt_sigtimedwait         compat_sys_rt_sigtimedwait
524     x32     rt_sigqueueinfo         compat_sys_rt_sigqueueinfo
525     x32     sigaltstack             compat_sys_sigaltstack
526     x32     timer_create            compat_sys_timer_create
527     x32     mq_notify               compat_sys_mq_notify
528     x32     kexec_load              compat_sys_kexec_load
529     x32     waitid                  compat_sys_waitid
530     x32     set_robust_list         compat_sys_set_robust_list
531     x32     get_robust_list         compat_sys_get_robust_list
532     x32     vmsplice                compat_sys_vmsplice
533     x32     move_pages              compat_sys_move_pages
534     x32     preadv                  compat_sys_preadv64
535     x32     pwritev                 compat_sys_pwritev64
536     x32     rt_tgsigqueueinfo       compat_sys_rt_tgsigqueueinfo
537     x32     recvmmsg                compat_sys_recvmmsg
538     x32     sendmmsg                compat_sys_sendmmsg
539     x32     process_vm_readv        compat_sys_process_vm_readv
540     x32     process_vm_writev       compat_sys_process_vm_writev
541     x32     setsockopt              compat_sys_setsockopt
542     x32     getsockopt              compat_sys_getsockopt
543     x32     io_setup                compat_sys_io_setup
544     x32     io_submit               compat_sys_io_submit
545     x32     execveat                compat_sys_execveat/ptregs
546     x32     preadv2                 compat_sys_preadv64v2
547     x32     pwritev2                compat_sys_pwritev64v2
```

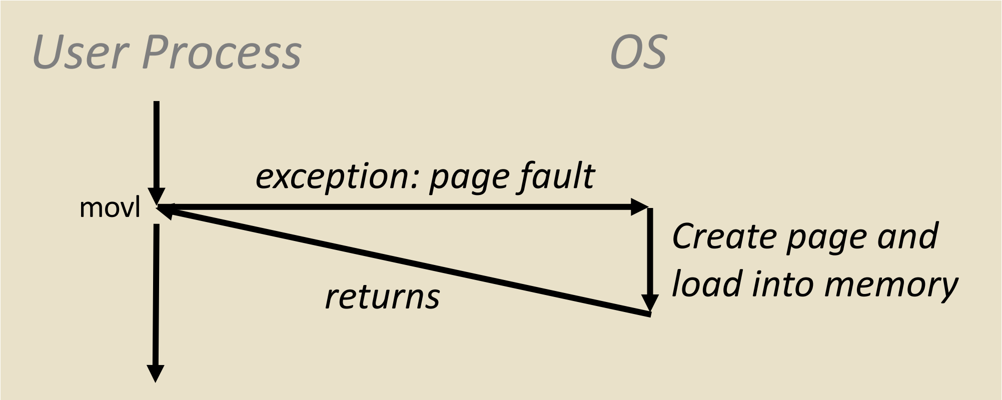
arch/x86/entry/syscalls/syscall_64.tbl

Fault Example: Page Fault

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

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

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



User Process

OS

movl

exception: page fault

Create page and load into memory

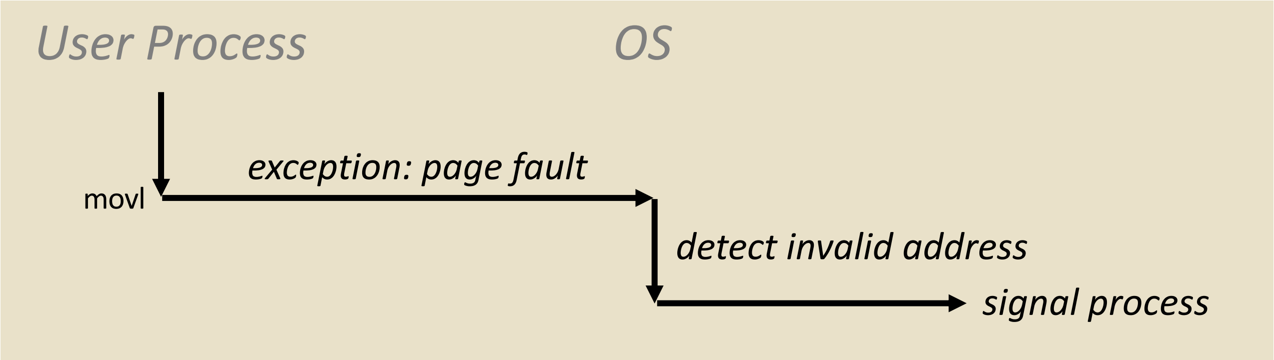
returns

- Page handler must load page into physical memory
- Returns to faulting instruction
- Successful on second try

Abort Example: Invalid Memory Reference

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

```
 80483b7:      c7 05 60 e3 04 08 0d   movl   $0xd,0x804e360
```



Page handler detects invalid address

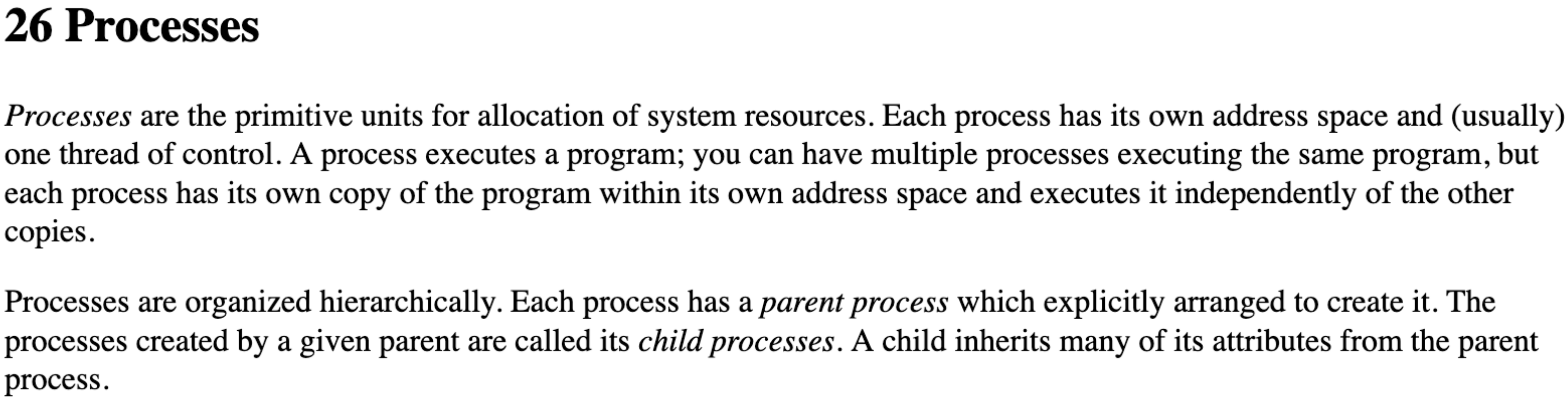
Sends `SIGSEGV` signal to user process

User process exits with “segmentation fault”

Outline

- Interrupt Handling
- Process Management
- Signals

Processes



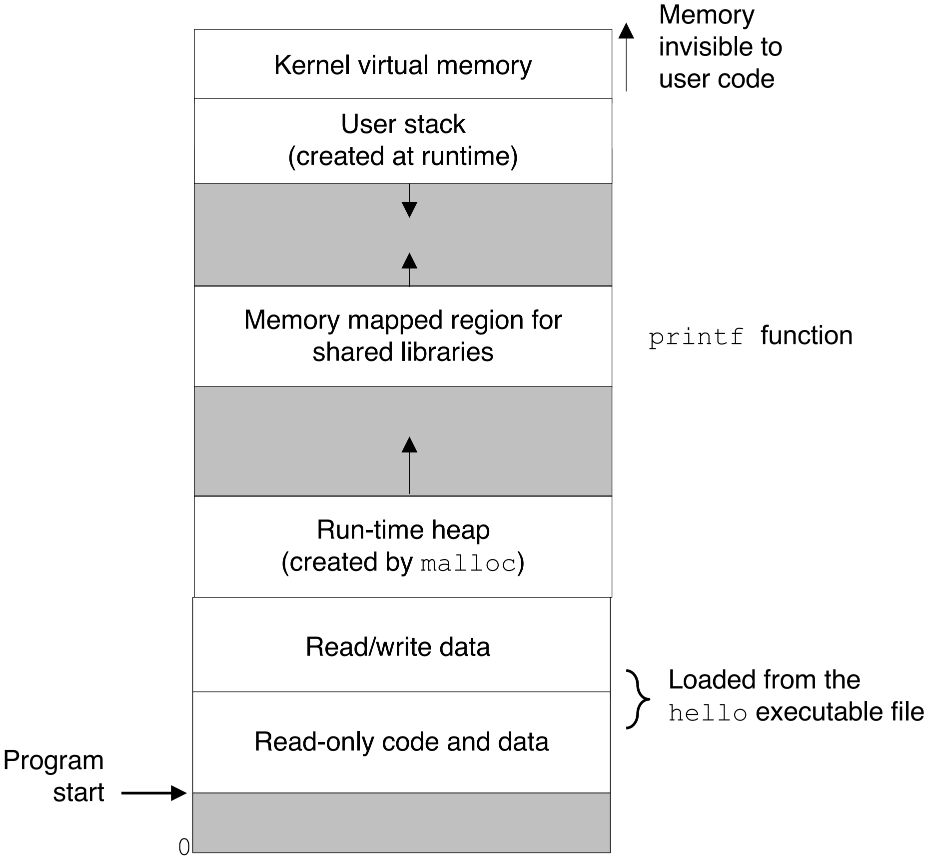
26 Processes

Processes are the primitive units for allocation of system resources. Each process has its own address space and (usually) one thread of control. A process executes a program; you can have multiple processes executing the same program, but each process has its own copy of the program within its own address space and executes it independently of the other copies.

Processes are organized hierarchically. Each process has a *parent process* which explicitly arranged to create it. The processes created by a given parent are called its *child processes*. A child inherits many of its attributes from the parent process.

https://www.gnu.org/software/libc/manual/html_node/Processes.html

Virtual Memory



Each process has its own address space.

All address spaces are structured in the same way.

Processes

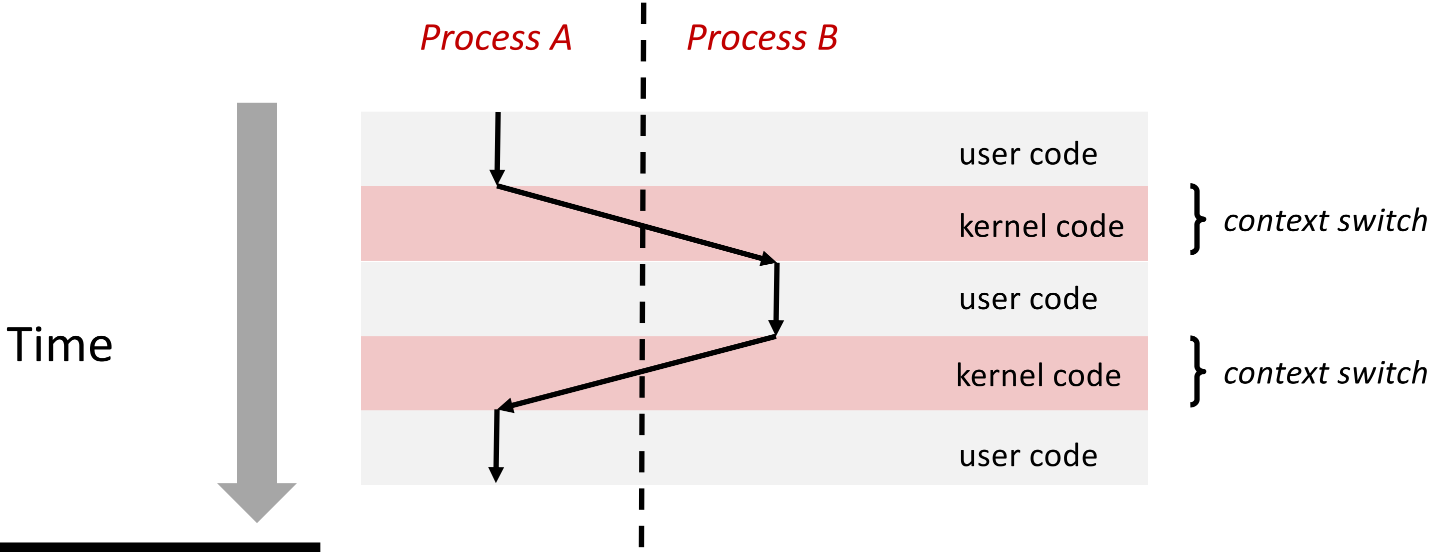
- Process creation/termination/control defined in standard C library (unistd.h)

- Transferring the thread of control from one process to another is called *context switching*. It is managed by the OS kernel.

Context Switching

Control flow passes from one process to another via a *context switch*

Important: the kernel is not a separate process, but rather runs as part of some user process



Context Switching

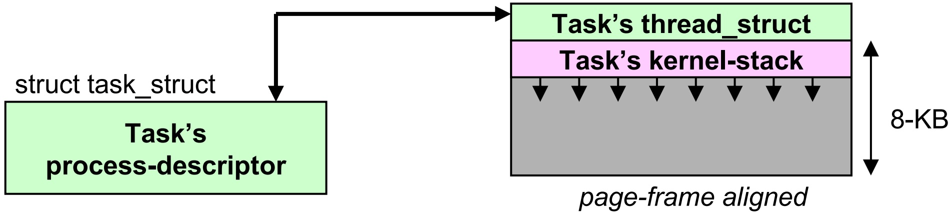
Processor modes:

- Supervisor vs. user modes: “Supervisor mode may provide access to different peripherals, to memory management hardware or to different memory address spaces. It is also capable of interrupt enabling, disabling, returning and loading of processor status.”
- Supervisor mode entered on system call (see slide 11)

Context Switching

When a context switch is made the scheduler marks the task as interruptible, saves the process's task_struct and replaces the current tasks pointer with a pointer to the new process's task_struct, marked as running, restoring its memory access and register context.

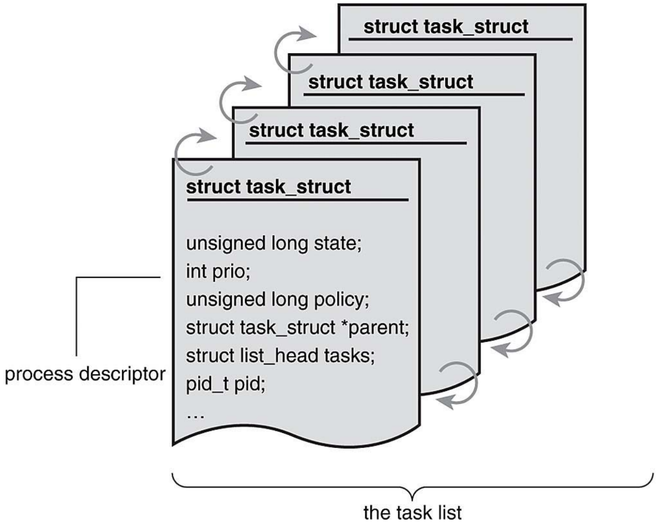
Process State (kernel)



Process context:

- 8KB / process in kernel space to store process descriptor task_struct (/linux/include/linux/sched.h).
  State:
    #define TASK_RUNNING 0
    #define TASK_INTERRUPTIBLE 1
    #define TASK_UNINTERRUPTIBLE 2
    #define TASK_ZOMBIE 4
    #define TASK_STOPPED 8

  Process ID
  + virtual memory info, file system info, open files, signal handlers, …
- The thread of execution thread_struct (/linux/arch/x86/include/asm/processor.h)
  PC, registers, Fault info,



Break!

- Break!
- We’ll continue in 15 minutes.

Process Management (libc)

- Spawning process: fork()
- Terminating process: exit()
- Waiting for process: wait()
- Executing a program within a process: execve()

On cos: /usr/include/unistd.h

`fork`: Creating New Processes

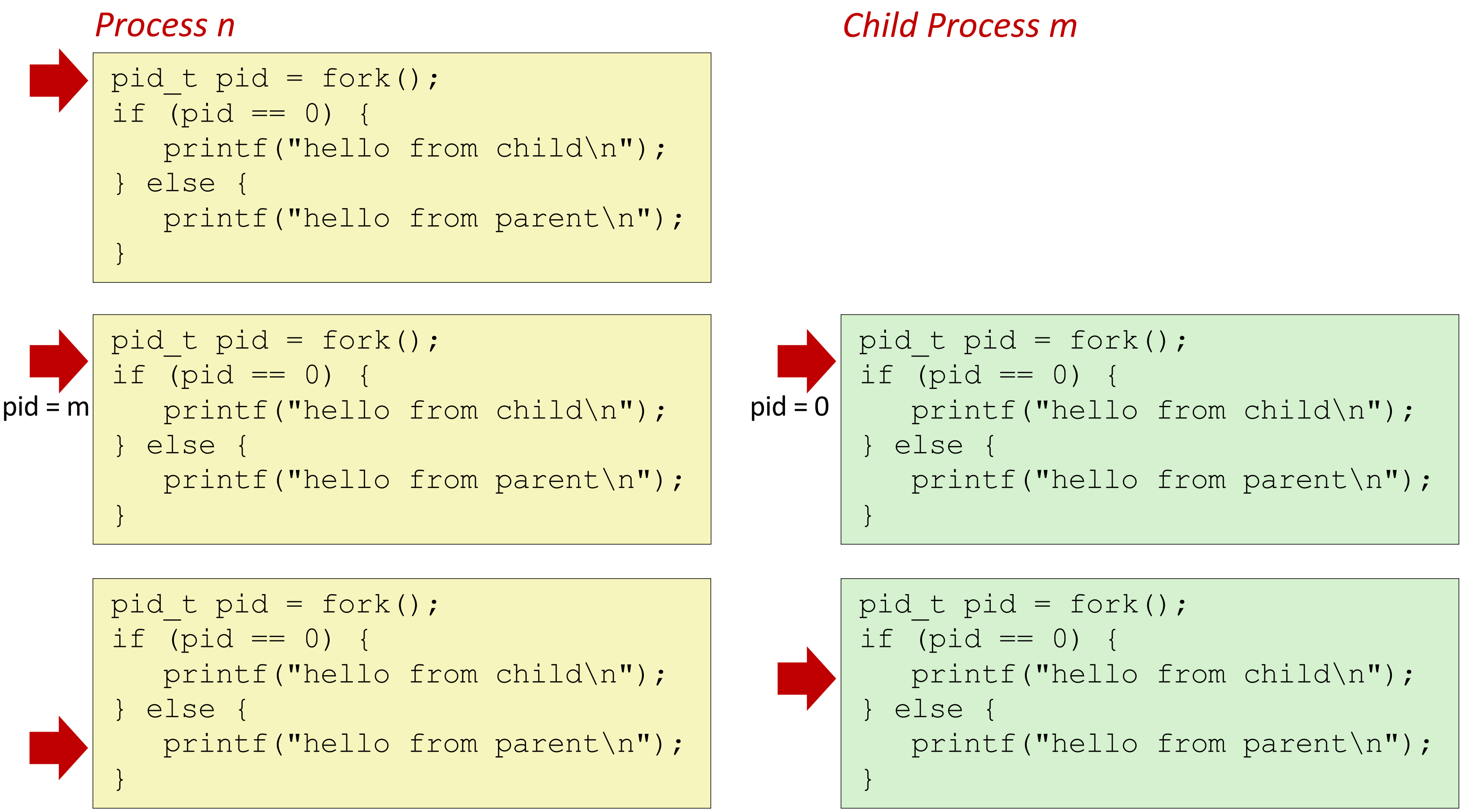
`int fork(void)`

- creates a new process (child process) that is identical to the calling process (parent process)
- returns 0 to the child process
- returns child's **`pid`** to the parent process

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Fork is interesting (and often confusing) because it is called *once* but returns *twice*

Understanding fork



Process n

Child Process m

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = m

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = 0

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
hello from parent
```

Which one is first?
No guarantee!

```
hello from child
```

Fork Example #1

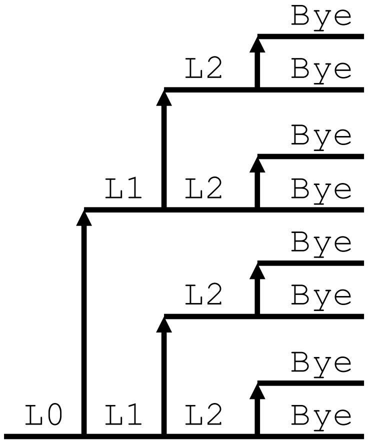
- Parent and child both run same code
  - Distinguish parent from child by return value from **`fork`**
- Start with same state, but each has private copy
  - Including shared output file descriptor
  - Relative ordering of their print statements undefined

```
void fork1()
{
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child has x = %d\n", ++x);
    } else {
        printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```

Fork Example #2

Both parent and child can continue forking

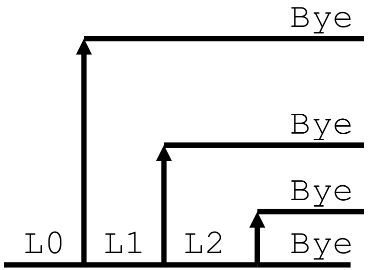
```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



Fork Example #3

Both parent and child can continue forking

```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```



`exit`: Ending a process

`void exit(int status)`

exits a process

- Normally return with status 0

`atexit()` registers functions to be executed upon exit

```
void cleanup(void) {
   printf("cleaning up\n");
}

void fork6() {
   atexit(cleanup);
   fork();
   exit(0);
}
```

Zombies

- Idea
  - When process terminates, still consumes system resources
    - Various tables maintained by OS
  - Called a “zombie”
    - Living corpse, half alive and half dead
- Reaping
  - Performed by parent on terminated child
  - Parent is given exit status information
  - Kernel discards process
- What if parent doesn’t reap?
  - If any parent terminates without reaping a child, then child will be reaped by **`init`** process
  - So, only need explicit reaping in long-running processes
    - e.g., shells and servers

Zombie Example

```
void fork7()
{
    if (fork() == 0) {
     /* Child */
     printf("Terminating Child, PID = %d\n",
            getpid());
     exit(0);
    } else {
     printf("Running Parent, PID = %d\n",
            getpid());
     while (1)
         ; /* Infinite loop */
    }
}
```

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 forks
 6640 ttyp9    00:00:00 forks <defunct>
 6641 ttyp9    00:00:00 ps
linux> kill 6639
[1]    Terminated
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6642 ttyp9    00:00:00 ps
```

`ps` shows child process as “defunct”

Killing parent allows child to be reaped by `init`

Nonterminating Child Example

```
void fork8()
{
    if (fork() == 0) {
     /* Child */
     printf("Running Child, PID = %d\n",
            getpid());
     while (1)
         ; /* Infinite loop */
    } else {
     printf("Terminating Parent, PID = %d\n",
            getpid());
     exit(0);
    }
}
```

```
linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6676 ttyp9    00:00:06 forks
 6677 ttyp9    00:00:00 ps
linux> kill 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6678 ttyp9    00:00:00 ps
```

- Child process still active even though parent has terminated

- Must kill explicitly, or else will keep running indefinitely

`wait`: Synchronizing with Children

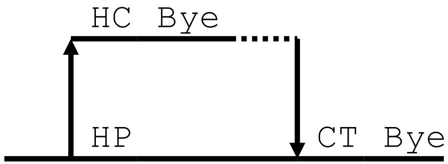
`int wait(int *child_status)`

- suspends current process until one of its children terminates
- return value is the **`pid`** of the child process that terminated
- if **`child_status != NULL`**, then the object it points to will be set to a status indicating why the child process terminated

`wait`: Synchronizing with Children

```
void fork9() {
   int child_status;

   if (fork() == 0) {
      printf("HC: hello from child\n");
   }
   else {
      printf("HP: hello from parent\n");
      wait(&child_status);
      printf("CT: child has terminated\n");
   }
   printf("Bye\n");
   exit();
}
```



`wait()` Example

If multiple children completed, will take in arbitrary order

Can use macros WIFEXITED and WEXITSTATUS to get information about exit status

```
void fork10()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                           wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", wpid);
    }
}
```

`waitpid()`: Waiting for a Specific Process

`waitpid(pid, &status, options)`

suspends current process until specific process terminates

```
void fork11()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
    if ((pid[i] = fork()) == 0)
        exit(100+i); /* Child */
    for (i = N-1; i >= 0; i--) {
    pid_t wpid = waitpid(pid[i], &child_status, 0);
    if (WIFEXITED(child_status))
        printf("Child %d terminated with exit status %d\n",
            wpid, WEXITSTATUS(child_status));
    else
        printf("Child %d terminated abnormally\n", wpid);
    }
}
```

`execve`: Loading and Running Programs

```
int execve(
     char *filename,
     char *argv[],
     char *envp[]
   )
```

Loads and runs in current process:

- Executable **`filename`**
- With argument list **`argv`**
- And environment variable list **`envp`**

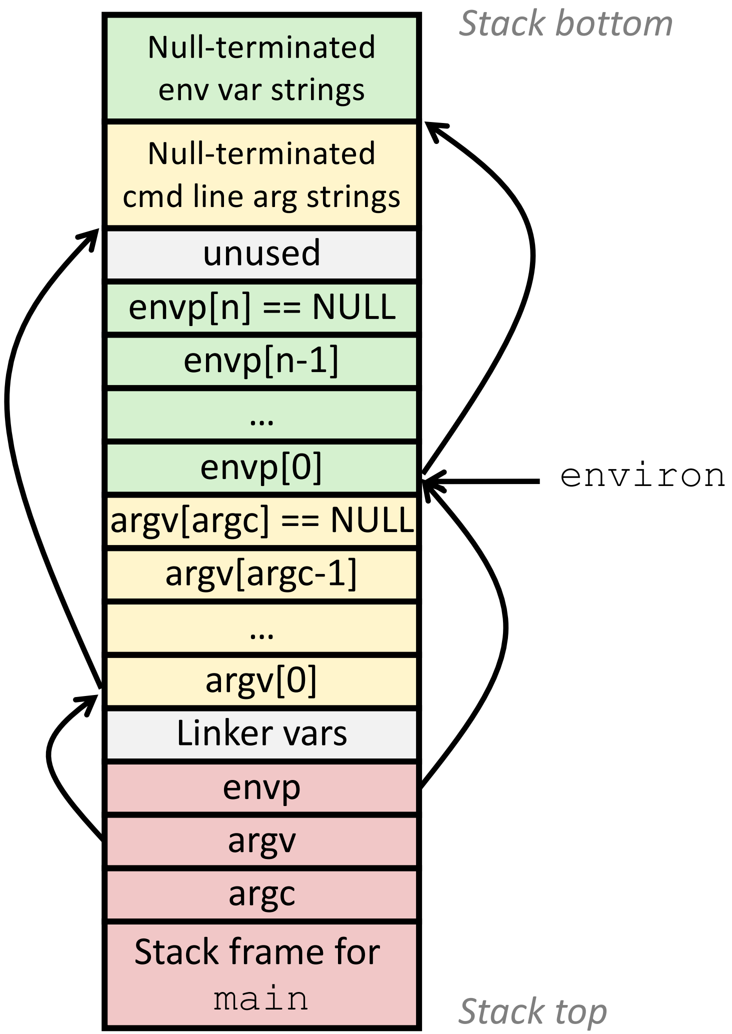
Does not return (unless error)

Overwrites code, data, and stack

- keeps pid, open files and signal context

Environment variables:

- “name=value” strings
- `getenv and putenv`



Stack bottom

Null-terminated env var strings

Null-terminated cmd line arg strings

unused

envp[n] == NULL

envp[n-1]

…

envp[0]

`environ`

argv[argc] == NULL

argv[argc-1]

…

argv[0]

Linker vars

envp

argv

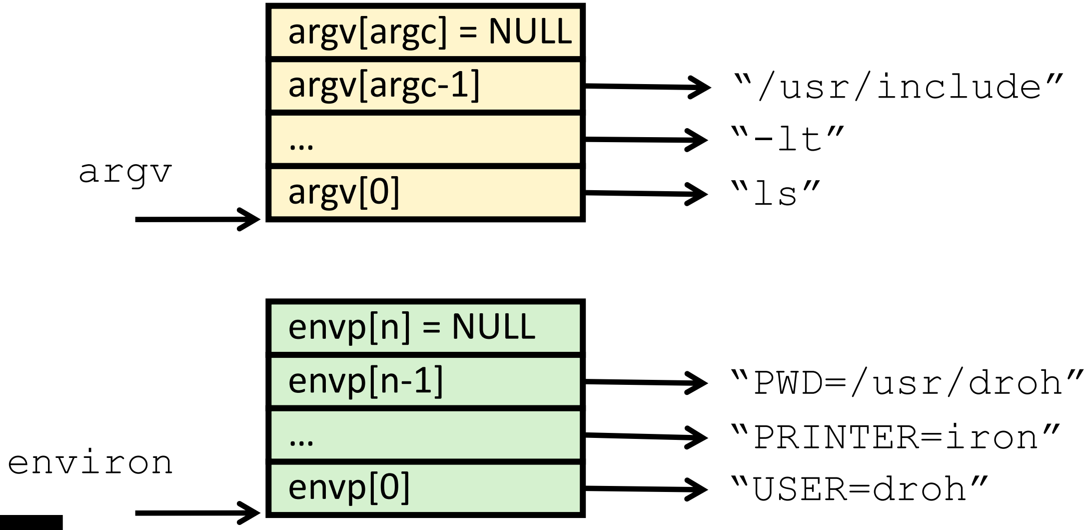
argc

Stack frame for `main`

Stack top

`execve` Example

```
if ((pid = Fork()) == 0) { /* Child runs user job */
    if (execve(argv[0], argv, environ) < 0) {
        printf("%s: Command not found.\n", argv[0]);
        exit(0);
    }
}
```



Outline

- Interrupt Handling
- Process Management
- Signals

Signals

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

- akin to 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	Interrupt (e.g., ctl-c from keyboard)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate & Dump	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

Sending a Signal

- Kernel *sends* (delivers) a signal to a *destination process* by updating some state in the context of the destination process

- Kernel sends a signal for one of the following reasons:
  - Kernel has detected a system event such as divide-by-zero (SIGFPE) or the termination of a child process (SIGCHLD)
  - Another process has invoked the **`kill`** system call to explicitly request the kernel to send a signal to the destination process

Receiving a Signal

A destination process *receives* a signal. It is forced by the kernel to react in some way to the delivery of the signal

Three 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***

Akin to a hardware exception handler being called in response to an asynchronous interrupt

Signal Concepts

Kernel maintains `pending` and `blocked` bit vectors in the context of each process

- **`pending`**: represents the set of pending signals
  - Kernel sets bit k in **`pending`** when a signal of type k is delivered
  - Kernel clears bit k in **`pending`** when a signal of type k is received
- **`blocked`**: represents the set of blocked signals
  - Can be set and cleared by using the **`sigprocmask`** function

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

`/bin/kill -9 -24817`

Send SIGKILL to every process in process group 24817

```
linux> ./forks 16
Child1: pid=24818 pgrp=24817
Child2: pid=24819 pgrp=24817

linux> ps
  PID TTY          TIME CMD
24788 pts/2    00:00:00 tcsh
24818 pts/2    00:00:02 forks
24819 pts/2    00:00:02 forks
24820 pts/2    00:00:00 ps
linux> /bin/kill -9 -24817
linux> ps
  PID TTY          TIME CMD
24788 pts/2    00:00:00 tcsh
24823 pts/2    00:00:00 ps
linux>
```

Receiving Signals

Suppose kernel is returning from an exception handler and is ready to pass control to process *p*

Kernel computes `pnb = pending & ~blocked`

The set of pending nonblocked signals for process *p*

If (`pnb == 0`)

Pass control to next instruction in the logical flow for *p*

Else

Choose least nonzero bit *k* in **`pnb`** and force process *p* to ***receive*** signal *k*

The receipt of the signal triggers some ***action*** by *p*

Repeat for all nonzero *k* in **`pnb`**

Pass control to next instruction in logical flow for *p*

Default Actions

Each signal type has a predefined *default action*, which is one of:

- The process terminates
- The process terminates and dumps core
- The process stops until restarted by a SIGCONT signal
- The process ignores the signal

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 ***signal handler***

- Called when process receives signal of type **`signum`**
- Referred to as ***"installing"*** the handler
- Executing handler is called ***"catching"*** or ***"handling"*** the signal
- When the handler executes its return statement, control passes back to instruction in the control flow of the process that was interrupted by receipt of the signal

Signal Handling Example

```
void int_handler(int sig) {
    safe_printf("Process %d received signal %d\n", getpid(), sig);
    exit(0);
}

void fork13() {
    pid_t pid[N];
    int i, child_status;
    signal(SIGINT, int_handler);
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()
            while(1); /* chi
        }
    for (i = 0; i < N; i++)
        printf("Killing proc
        kill(pid[i], SIGINT)
    }
    for (i = 0; i < N; i++)
        pid_t wpid = wait(&c
        if (WIFEXITED(child_
            printf("Child %c
                   wpid, WEX
        else
            printf("Child %c
    }
}
```

```
linux> ./forks 13
Killing process 25417
Killing process 25418
Killing process 25419
Killing process 25420
Killing process 25421
Process 25417 received signal 2
Process 25418 received signal 2
Process 25420 received signal 2
Process 25421 received signal 2
Process 25419 received signal 2
Child 25417 terminated with exit status 0
Child 25418 terminated with exit status 0
Child 25420 terminated with exit status 0
Child 25419 terminated with exit status 0
Child 25421 terminated with exit status 0
linux>
```

Motivation for processes

a) **Security**

- You may want to isolate execution of parts of the program.
  - If child process crashes, the control process is still running.

b) **Performance**

- With multiple processes you can split parts of the execution into multiple CPUs.
  - Important if you are working with large amount of data or long-running processes.

Take-Aways

An *exception* is a transfer of control to the OS in response to some *event (asynchronous vs. synchronous (trap, fault, abort)).*

Process has own address space and thread of control. Libs defines primitives for spawning/terminating processes, waiting for processes and executing programs within a process. The kernel takes care of context switching.

Hardware interrupts first handled in hardware then in software (kernel) through interrupt vector. Signals as process-level interrupt handling.