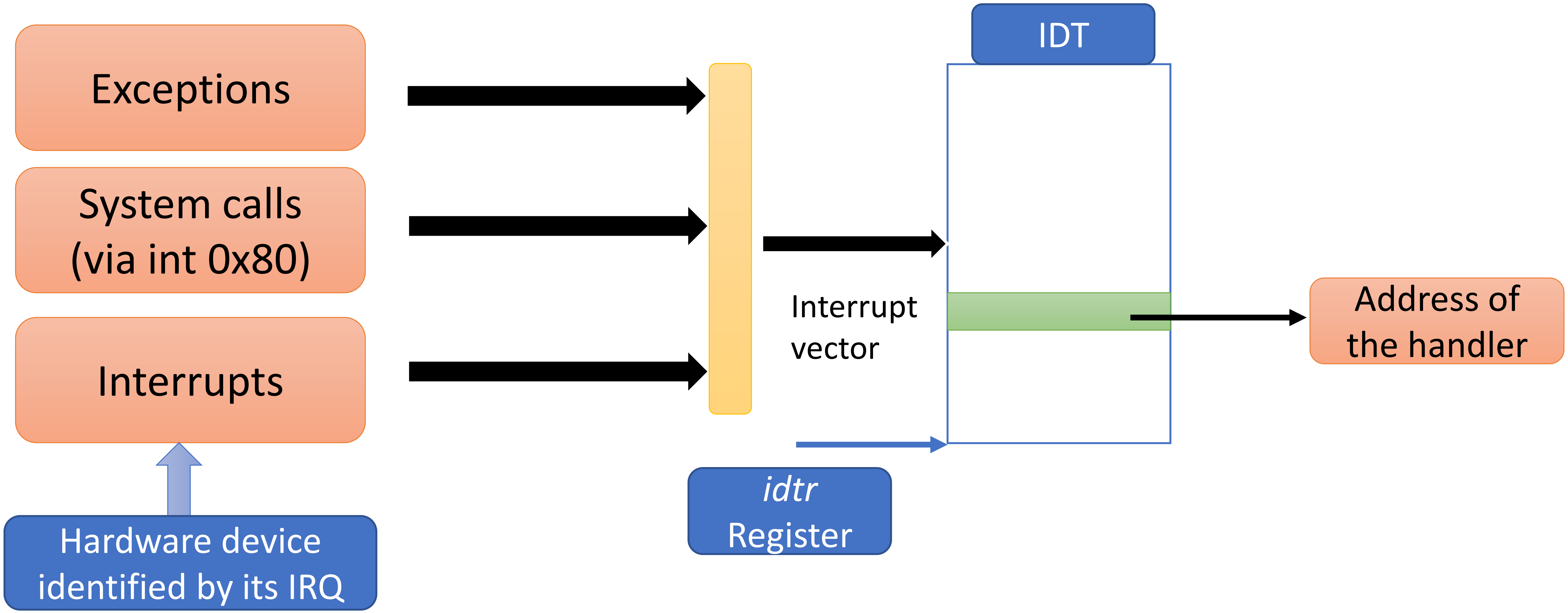
Chapter 4: System Calls, Interrupts and Signals

Smruti R Sarangi

IIT Delhi

Interrupt Descriptor Table (IDT)



Digression: Linked Lists in Linux

/include/linux/types.h

```
struct list_head {
        struct list_head *next, *prev;
}
```

Where is the data that each node should contain?

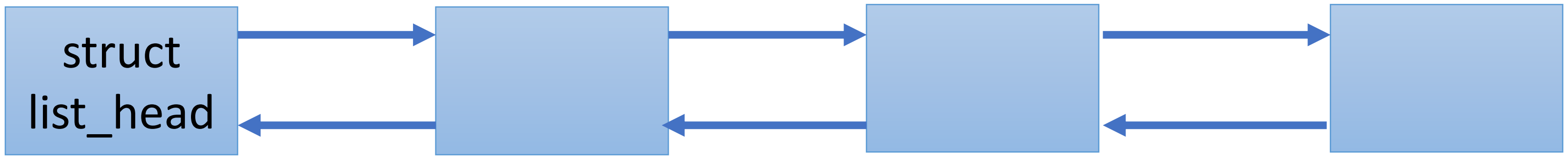
/include/linux/list.h

```
#define list_entry (ptr, type, member)  container_of (ptr, type, member)
```

/include/linux/container_of.h

```
#define container_of(ptr, type, member) ({
        void *__mptr = (void *)(ptr);
        ((type *)(__mptr - offsetof(type, member))); })
```

What is happening here?



- Given a list_head find the pointer to the structure that it is a part of.

```
struct abc {
        int x;
        struct list_head list;
}
```

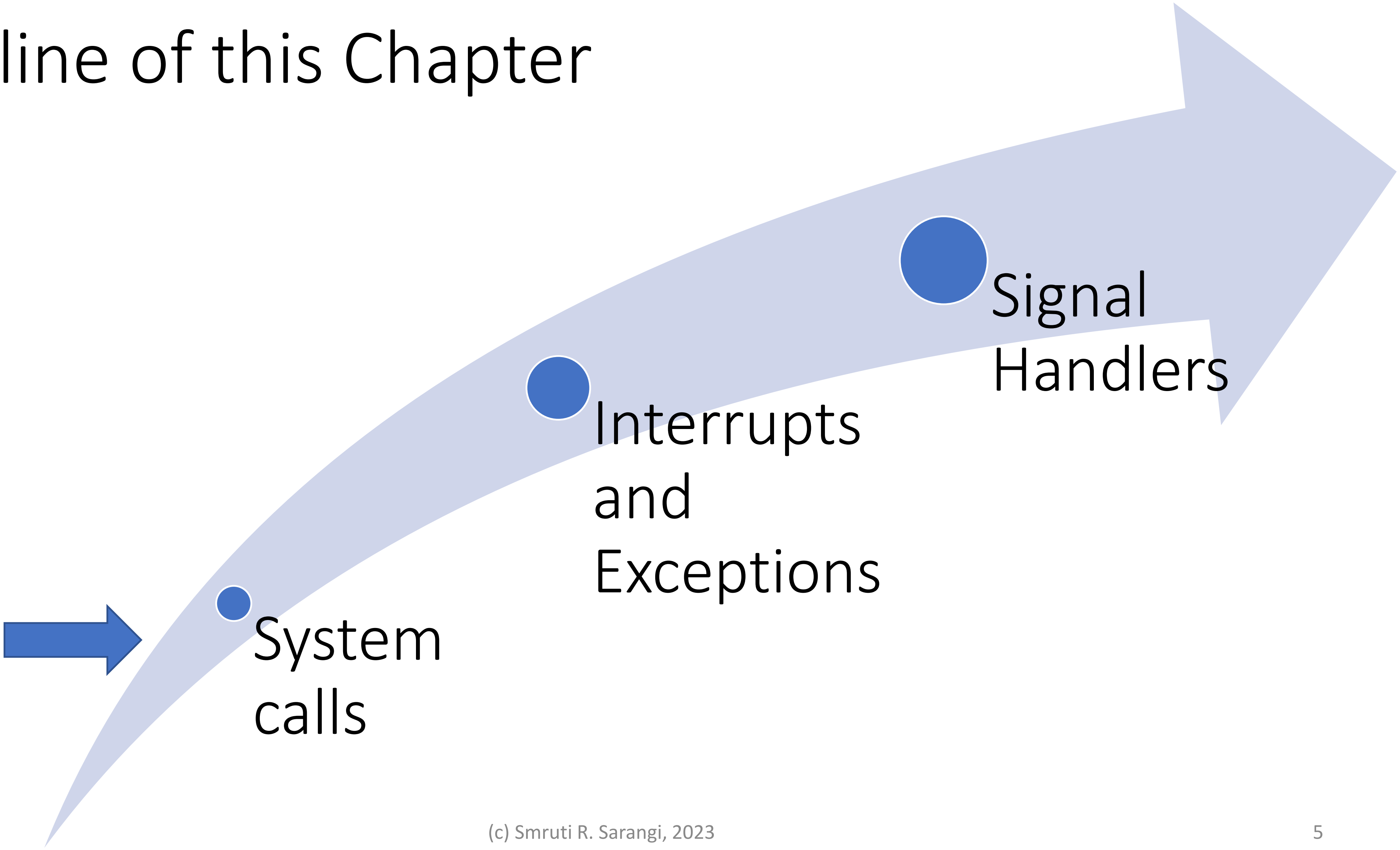
```
struct abc* current = ....     ;
struct abc* next = list_entry (current->list.next,
                   struct abc, list);
```

```
struct def {
        int x;
        float y;
        struct list_head list;
}
```

```
struct def* current = ....     ;
struct abc* next = list_entry (current->list.next,
                   struct abc, list);
```

Linked list of a different type

Outline of this Chapter



Consider a simple piece of C code

```
#include <stdio.h>

int main() {
        printf ("Hello World \n");
}
```

glibc version 2.5

- Where is *printf* defined?
- It is defined deep inside the glibc library
- Let us trace its path.

stdio.h

vfprintf-internal.c

```
int printf (const char* format, ...)
```

- The format string is a constant string (cannot be modified)
- The … indicates a variable number of arguments
- This is the flow of control inside the glibc code
  - printf → __printf (alias) → vfprintf → printf_positional → outstring → PUT
- Very soon, the signature changes. The functions become more and more generic. An additional argument FILE *s is introduced. It defaults to *stdout* (terminal)

```
int vfprintf (FILE *s, const CHAR_T *format, va_list ap, unsigned int mode_flags);
```

Chain of calls

- __xsputn function → new_do_write in fileops.c (glibc)
- Finally make the *write* system call using the extensive *syscall* support of glibc.

https://codebrowser.dev/glibc/glibc/sysdeps/unix/sysv/linux/x86_64/sysdep.h.html

Let us discuss the general concepts that underlie standard library design.

General Concepts about Library Design

outstring

printf_positional

vfprintf

printf

Send the output to the file

Creates the string that needs to be printed

Function that is visible to the programmer

Generic version that prints to a file

Access a table of function pointers

Prepare the arguments for the *write* system call

Make the system call

Abstracted out as a file

- stdout → the terminal
- stdin → from the terminal (scanf)
- stderr → the error stream (defaults to the terminal)

System Call Format in Linux

Attribute	Register
System call number	rax
Arg 1	rdi
Arg 2	rsi
Arg 3	rdx
Arg 4	r10
Arg 5	r8
Arg 6	r9

Rest on the stack

rcx and *r11* are internally used

Just setup the arguments and call the *syscall* instruction

The *rax* register contains the return value.

Let us now look at the OS side

/arch/x86/entry/entry_64.S

- What happens after a syscall?

We have already seen the part that stores the state of the process in the previous set of slides.

Also, turn off interrupts

Call *do_syscall_64*

do_syscall_64

/arch/x86/entry/common.c

- First enable interrupts on the local CPU
- Use the contents of the *rax* register to access a system call table
- This will provide the function pointer of the system call
- Invoke the system call handler.

https://elixir.bootlin.com/linux/latest/source/arch/x86/entry/syscalls/syscall_64.tbl

/include/uapi/asm-generic/unistd.h

Shows the list of system calls in Linux x86-64

First 12 entries

			Function name
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

Let us look at *sys_write*

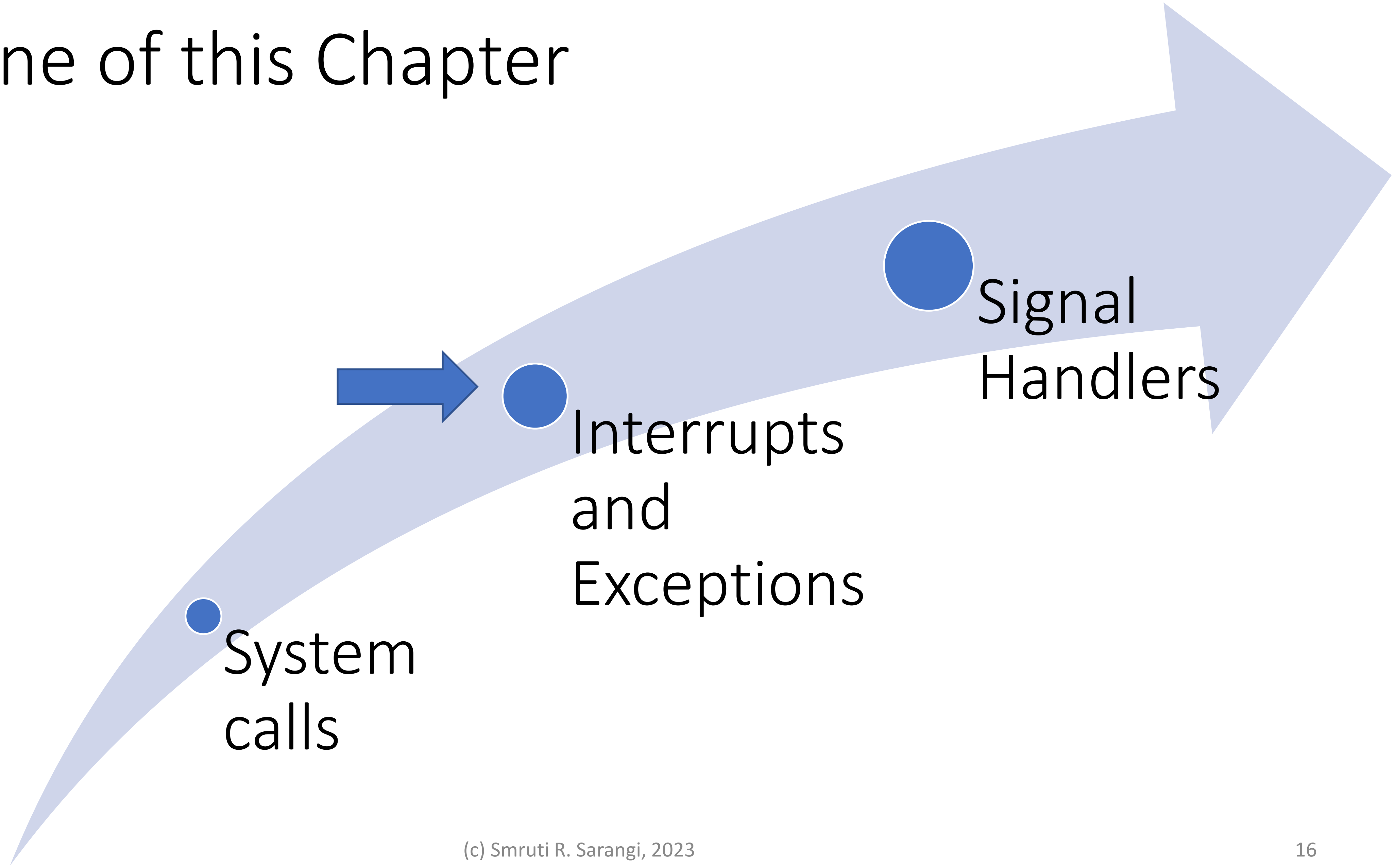
- There is a complex system of macros that requires a genius to decipher
- It finally takes you to *fs/read_write.c => ksys_write.*
- This is where the system call is processed and the value is returned.

Steal this opportunity

Check TIF_NEED_RESCHED

- If the current task has eaten up a lot of CPU time or there is a higher priority thread waiting, then the scheduler sets the *TIF_NEED_RESCHED* flag
- It is a flag in *thread_info*
- The kernel checks this flag and if the current task has exceeded its quota
  - Invokes the *schedule* function
  - Possibly schedules another task on the same core or continues with the current task
- A reverse process is followed to restore the state of the previously swapped out task
- Use *sysret* to return

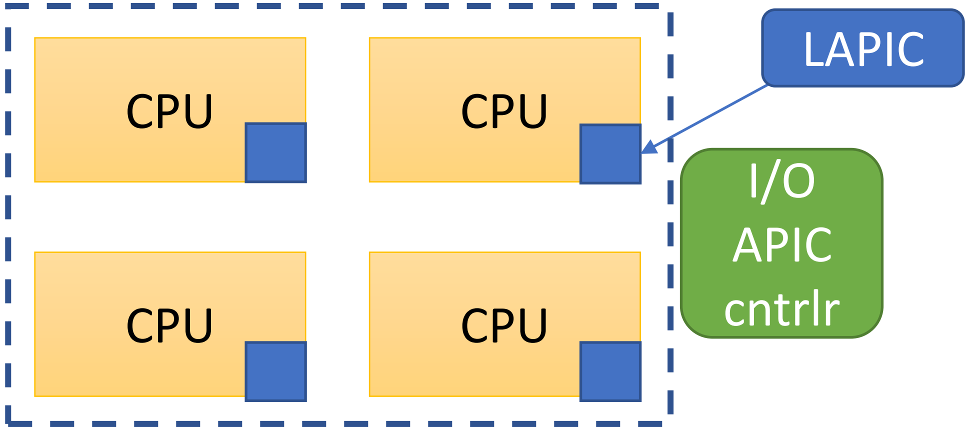
Outline of this Chapter



APIC Architecture

Advanced Programmable Interrupt Controller (APIC)

- Intel processors have a dedicated circuit to manage interrupts: interrupt controllers
- There are two kinds of interrupt controllers
  - LAPIC (Local APIC): This is a per-CPU interrupt controller
  - I/O APIC: One per the entire system. Manages external I/O interrupts.



I/O APIC

3. Buffer the interrupt vector (number) and data
4. Don't deliver if the interrupt is masked
5. Acknowledge receipt to the APIC

1. Enable/disable interrupts
2. Choose the highest priority interrupt

IRQ (interrupt request): Kernel identifier for a HW interrupt source

An interrupt vector (INT) identifies any kind of an interrupting event: interrupt, system call, exception, fault, etc.

The LAPIC sends an interrupt vector to the CPU (not an IRQ)

Roles of the Interrupt Controllers

- I/O APIC
  - Mainly contains a redirection table
  - Receives interrupts from peripheral buses and routes them to LAPICs
  - It typically has 24 interrupt lines (referred to as an IRQ)
  - Every device is assigned its IRQ number
    - Lower the number → Higher the priority
    - The timer interrupt's IRQ number is 0
- Local APIC
  - Receives the interrupt from the I/O APIC
  - Can also receive inter-processor interrupts (IPIs) from other LAPICs. The OS uses these to run the kernel on other cores.
  - Provides timing services (periodic interrupts and a timer)

Interrupt Distribution for all Types of Interrupts

I/O | Timer | Inter-Processor

Static Distribution
- One specific core
- Set of cores
- Broadcast

Dynamic Distribution
- The core specifies the priority in a task priority register
- The interrupt is delivered to the core that is running the task with the least priority

/proc/interrupts

IRQ #	CPU0	CPU1	CPU2	CPU3	Chip	HW IRQ -- type	Handler
	Count/CPU						
0:	7	0	0	0	IR-IO-APIC	2-edge	timer
1:	0	0	0	0	IR-IO-APIC	1-edge	i8042
8:	0	0	0	0	IR-IO-APIC	8-edge	rtc0
9:	0	4	0	0	IR-IO-APIC	9-fasteoi	acpi
12:	0	0	0	0	IR-IO-APIC	12-edge	i8042
16:	0	0	252	0	IR-IO-APIC	16-fasteoi	ehci_hcd:usb1
23:	0	0	0	33	IR-IO-APIC	23-fasteoi	ehci_hcd:usb2

Level triggered. Wait till acknowledged by the CPU

IRQ Sharing and Dynamic Allocation

- Many a time the same HW IRQ number is shared between devices. Given that it is not possible to know which device has raised the interrupt, we often need to run multiple handlers and check.
- Dynamic allocation: Many times an IRQ is allocated to a device when it is accessed for the first time. This minimizes the number of IRQs that are needed.

Interrupt Vector Range	Meaning
0-19	Non-maskable interrupts and exceptions
20-31	Reserved by Intel
32-127	External interrupts
128	System calls
239	Local APIC timer interrupt
251-253	Inter-processor interrupts

IRQ Assignment

Example

IRQ	Interrupt Vector	HW Device
0	32	Timer
1	33	Keyboard
8	40	System clock
10	42	Network interface
11	43	USB port

This mapping is important. The OS must be aware of this.

How does the hardware handle interrupts?

1. Determines the vector associated with the interrupt/exception.
2. This is between 0 and 255. Read the corresponding IDT entry.
3. Read the segment descriptor and get its base address.
4. Ensure that we have adequate privileges to access the segment.
5. Change the privilege level (if required) and perform an appropriate context switch.
6. Based on the exception/interrupt, we may need to execute the next instruction or the same instruction once again.
7. Load the interrupt handler

Kernel Code

Interrupt Descriptors: *struct irq_desc*

/include/linux/irqdesc.h

```
struct irq_desc {
        struct irq_common_data    irq_common_data;
        struct irq_data           irq_data;
        irq_flow_handler_t        handle_irq;
        struct irqaction          *action;
        ....
}
```

- `irq_common_data`: CPU affinity and per-IRQ data
- `irq_data`: interrupt number, HW interrupt number, pointer to IRQ domain (set of IRQs), per-chip data
- `handle_irq`: interrupt handler
- `action`: handler, device, flags, IRQ number, name of the IRQ

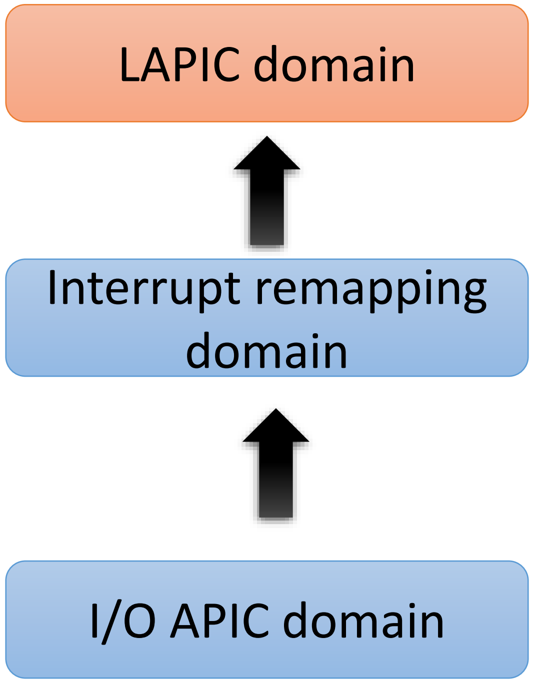
IRQ Domains

- Modern processors have a lot of interrupt controllers
- The interrupt number needs to retain its meaning – it somehow needs to be unique
- Solution: Assign each interrupt controller its domain. Let the IRQ numbers be unique in their domain. Basically, define virtual IRQ numbers.
- Each interrupt controller calls *irq_domain_add_<domain type>*
- A domain's job is to map HW IRQ numbers to irq_desc data structures [this is called reverse mapping]

Less than 256 HW IRQs → Linear list

More than 256 HW IRQs → Radix tree

Hierarchy of IRQ Domains



Setting up the Interrupt Descriptor Table (IDT)

- The IDT maps the interrupt number as seen by the CPU (interrupt vector) to the memory address of the handler
- It is setup by the BIOS
- The kernel re-initializes it and maps it to the idt_table data structure
    - Each entry of the table is indexed by the interrupt vector
    - It points to the function that needs to handle the interrupt
    - Segment + offset within the segment

Setting up the Interrupt Descriptor Table (IDT) – II

- The *start_kernel* function invokes a few functions
  - *early_irq_init*
    - Probes the default PCI devices and initializes an array of *irq_desc* structures
  - *init_IRQ*
    - Setup the per-CPU interrupt stacks
    - Sets up the basic IDT
  - *apic_bsp_setup*
    - Initializes the local APIC and I/O APIC

Setting up the Interrupt Table at Boot Time

- For standard x86 hardware, all the platform-specific *init* functions are defined in /arch/x86/kernel/x86_init.c

```
.irqs = {
        .pre_vector_init     = init_ISA_irqs,
        .intr_init           = native_init_IRQ,
        .intr_mode_select    = apic_intr_mode_select,
        .intr_mode_init      = apic_intr_mode_init,
        .create_pci_msi_domain = native_create_pci_msi_domain,
}
```

setup_local_APIC

setup_IO_APIC

Setting up the LAPIC

Set these first

/arch/x86/kernel/apic/apic.c

- Intel defines a bunch of APIC registers (MSRs)
- They are accessible via the WRMSR and RDMSR instructions
- A few of the important ones
  - LDR → Logical Destination Register
    Intel processors can be clustered (2-level hierarchy). The 32-bit register has a 16-bit cluster id and a 16-bit id within the cluster
  - DFR → Destination format register
    Indicates whether we are following clustering or not
  - TPR → Task priority register

Setting up the APICs

LAPIC

- Initialize the state
  - Initialize the timers
  - Set the performance counters to 0
  - Set the state to active

I/O APIC

/arch/x86/kernel/apic/io_apic.c

- For each I/O pin setup an IRQ
- Create a *domain* for each I/O-APIC
  - Group the set of IRQs in it

The Interrupt Call Path

- Now all the data structures have been set up

Push the interrupt vector to the stack

Jump to the IDT entry point

/arch/x86/entry/entry_64.S

```
DEFINE_IDTENTRY_IRQ(common_interrupt)
{
....
        struct irq_desc *desc;
        desc = __this_cpu_read(vector_irq[vector]);
        if (likely(!IS_ERR_OR_NULL(desc))) {
                handle_irq(desc, regs);
        } else {
        .....
}
```

vector number

all CPU registers

/arch/x86/kernel/irq.c

desc→ handle_irq(desc)

What does an IRQ handler look like?

/kernel/irq/chip.c

/kernel/irq/handle.c

- Consider the function *handle_level_irq* for level-sensitive interrupts
- It ultimately ends up invoking the IRQ handling function *__handle_irq_event_percpu*
  - Returns either NONE (interrupt not handled), HANDLED, or WAKE_THREAD (wake the handler thread and let it take care)
  - Because an IRQ can be shared, sequentially call all the handlers associated with it (refer to the linked list *struct irqaction** elements in *irq_desc*)
  - One of them will be associated with a device that should handle the interrupt.
  - The *irq_handler_t* function that actually handles the interrupt is defined in the code of the corresponding device driver (/drivers) directory.

Limitations on the Interrupt Handler

- The interrupt handler is typically known as the top half
- Its primary job is to acknowledge the receipt of the interrupt to the APIC
- AND collect data from the device or send some data to it.
- It is not allowed to make any form of a blocking call that uses locks.

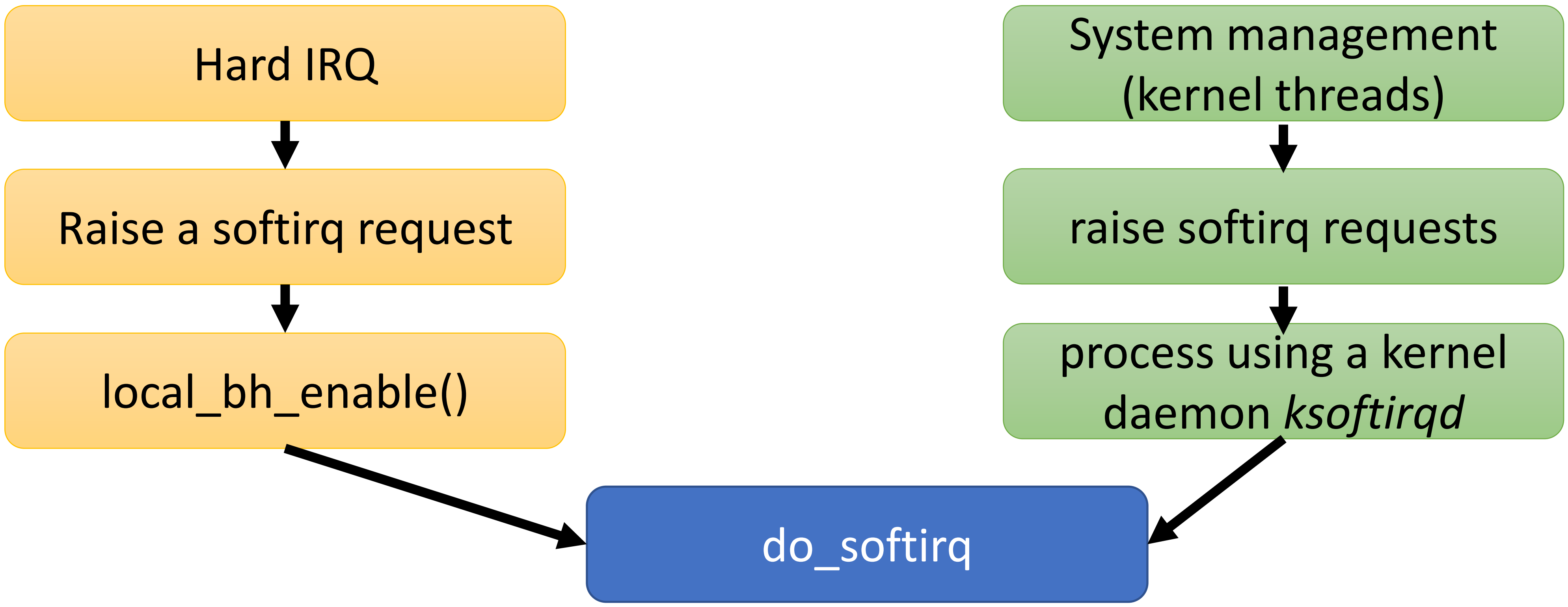
Schedule the work for later.
Make a deferred function call for finishing the process of servicing the interrupt.

softirqs, Threaded IRQs and Work Queues

- There used to be a concept called a bottom half
  - All the interrupts used to be enabled while it was executing
  - It could make blocking calls
- It has now morphed into softirqs and threaded IRQs
  - The key aim is to minimize the time that is spent with interrupts disabled
  - We would also not like to run ultra-high priority threads in the interrupt context
  - Hence, there is a need to create a low-priority deferred call mechanism
- There is also a more generic mechanism called *Work Queues*
  - We can run any generic function as a deferred function call in the process context (runs in the kernel space).

softirqs

- There are two ways that softirqs can be invoked: queue work after processing a regular IRQ (hard IRQ) or periodically process them



Raising a softirq

/kernel/softirq.c

- Different interrupt handlers call the function *raise_softirq* after they are done
- This sets the corresponding bit of a memory word stored in a per-CPU region

/include/linux/interrupt.h

Types of softirqs

HI_SOFTIRQ, TIMER_SOFTIRQ, NET_TX_SOFTIRQ, NET_RX_SOFTIRQ, BLOCK_SOFTIRQ, … SCHED_SOFTIRQ, HRTIMER_SOFTIRQ, …

Support a limited number of interrupts. Not flexible.

Invoking a softirq Handler

- Check all the softirq bits that are set to 1 in the memory word
- Invoke the corresponding softirq handlers
  - They run in the softirq interrupt context (less restrictive that the top half)
  - Cannot make blocking calls
  - They are reserved for kernel work, not device drivers
- There is a way to schedule work for a later time
  - Use threaded IRQs (this has taken the place of erstwhile *tasklets*)
  - Run a function on a separate thread
  - They run in process context (priority = 50)

Let us look at irqaction again

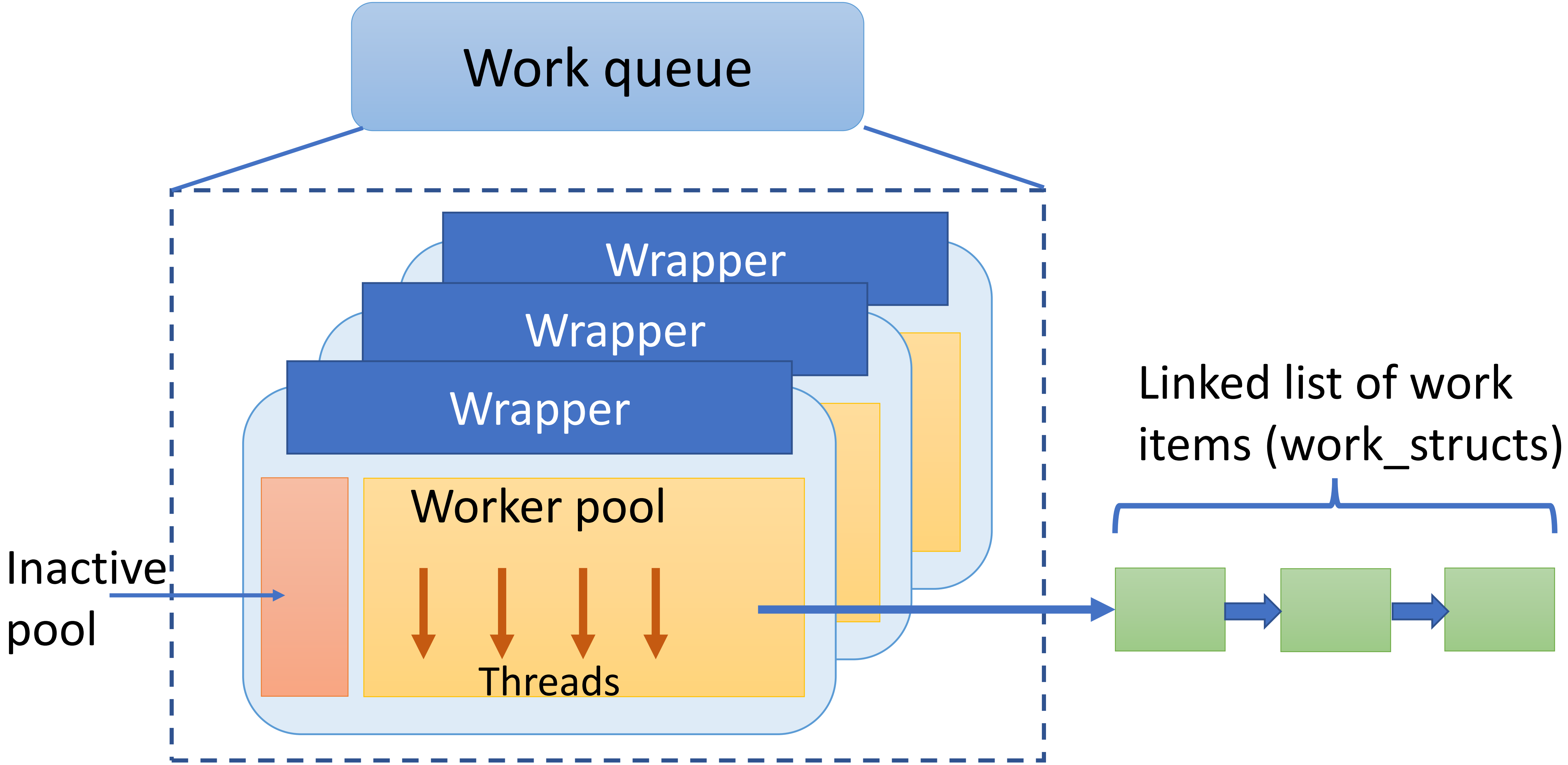
```
struct irqaction {
    …
    struct task_struct        *thread;
    irq_handler_t              thread_fn;
    ….
    }
```

Defines the function that needs to be called by the thread handling the IRQ.
The kernel creates a new thread and runs *thread_fn.*

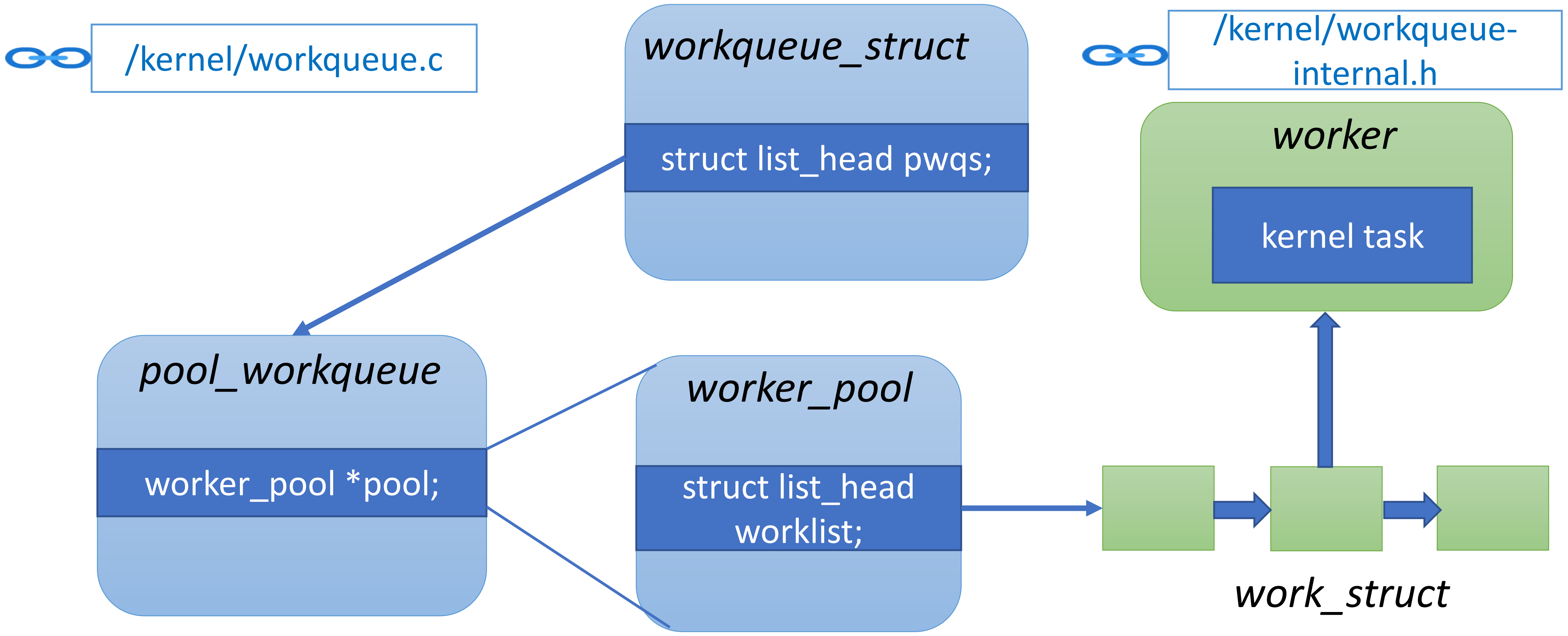
Work Queues

- The IRQ mechanism is not very generic
- It runs very high priority threads (often in interrupt context)
- We need a method to run low-priority kernel threads in the process context
- They should be generic and flexible
- This is where work queues come in
- Their basic element is a *work_struct* data structure – encapsulates a single work item

Conceptual Diagram



Important Structures and their Relationships



Work Struct

/include/linux/workqueue.h

```
struct work_struct {
        atomic_long_t data;
        struct list_head entry;
        work_func_t func;
};
```

- We maintain a pointer to a function and some data (it can be a pointer as well)
- It represents a pending function that needs to be executed
- This is the basic atom of work in a workqueue
- A driver creates a *struct work_struct* and inserts it in a given workqueue
- This is executed later on.

struct worker_pool

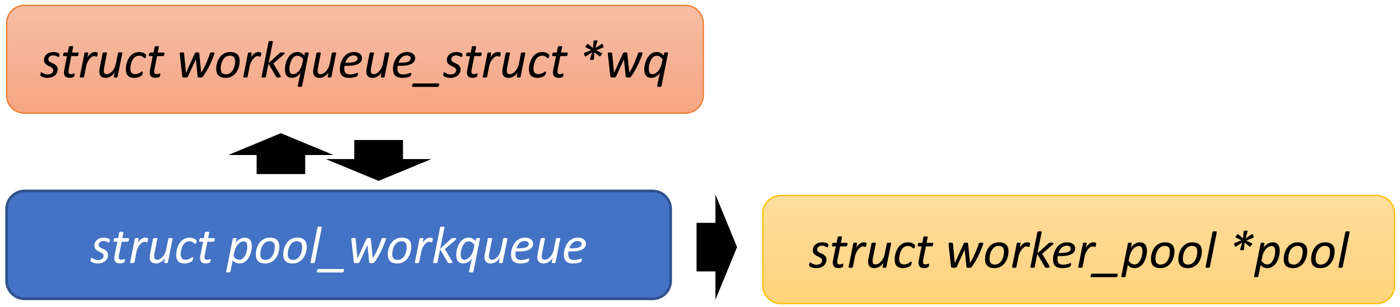
Maintain a pool of workers that can do the job for you. No need to allocate a new one.

struct worker_pool

cpu	the associated cpu
struct list_head worklist;	List of *work_structs*
struct list_head idle_list;	List of idle *workers*
hashtable busy_hash;	struct work struct * → worker

When new work comes assign a worker

struct pool_workqueue



- This is just a wrapper class on a *pool*. It restricts the size of the work in the pool such that the latter runs efficiently.
- If the number of active work items (nr_active) exceeds the maximum number of work items (max_active) →
  - Put them in the linked list *inactive_works*
  - Activate them later on when the work in the *pool* reduces

struct workqueue_struct

- This is the apex data structure.
- Contains a linked list of pool_workqueue structures

Key functions

/include/linux/workqueue.h

- create_workqueue (char * name)
- queue_work_on (int cpu, workqueue_struct *, work_struct *work)
- flush_scheduled work()
- bool flush_work (struct work_struct *work)

System wide Work Queues

System wide workqueues that are always present

```
extern struct workqueue_struct *system_wq;
extern struct workqueue_struct *system_highpri_wq;
extern struct workqueue_struct *system_long_wq;
extern struct workqueue_struct *system_unbound_wq;      /* not bound to a specific CPU */
extern struct workqueue_struct *system_freezable_wq;    /* can be suspended and resumed */
extern struct workqueue_struct *system_power_efficient_wq;    /* power efficient jobs */
extern struct workqueue_struct *system_freezable_power_efficient_wq;
```

Additionally, there are two workqueues per CPU: normal priority and high priority

Exceptions

/arch/x86/include/asm/trapnr.h

Exceptions

- Intel processors define up to 24 different types of exceptions (current version of the kernel)

Name of the Trap/Exception	Number	Description
X86_TRAP_DE	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

• • •

Consider the *divide* error

/arch/x86/include/asm/identry.h

```
DECLARE_IDTENTRY(X86_TRAP_DE,    exc_divide_error);
```

/arch/x86/kernel/traps.c

```
DEFINE_IDTENTRY(exc_divide_error)
{
        do_error_trap(regs, 0, "divide error", X86_TRAP_DE, SIGFPE,
                FPE_INTDIV, error_get_trap_addr(regs));
}
```

Send the SIGFPE signal to the process

DEFINE_IDTENTRY_*

/arch/x86/kernel/traps.c

DEFINE_IDTENTRY_*

- Send a signal to the process
- Write to the kernel logs using *printk*
- Go to kernel panic mode in the case of a double fault (fault within fault handler). Halt the system
- If it is a math error like invoking an instruction that the processor does not support, then try to emulate it or ignore it.
- Use the notify_die mechanism

/kernel/panic.c

The *notify_die* mechanism

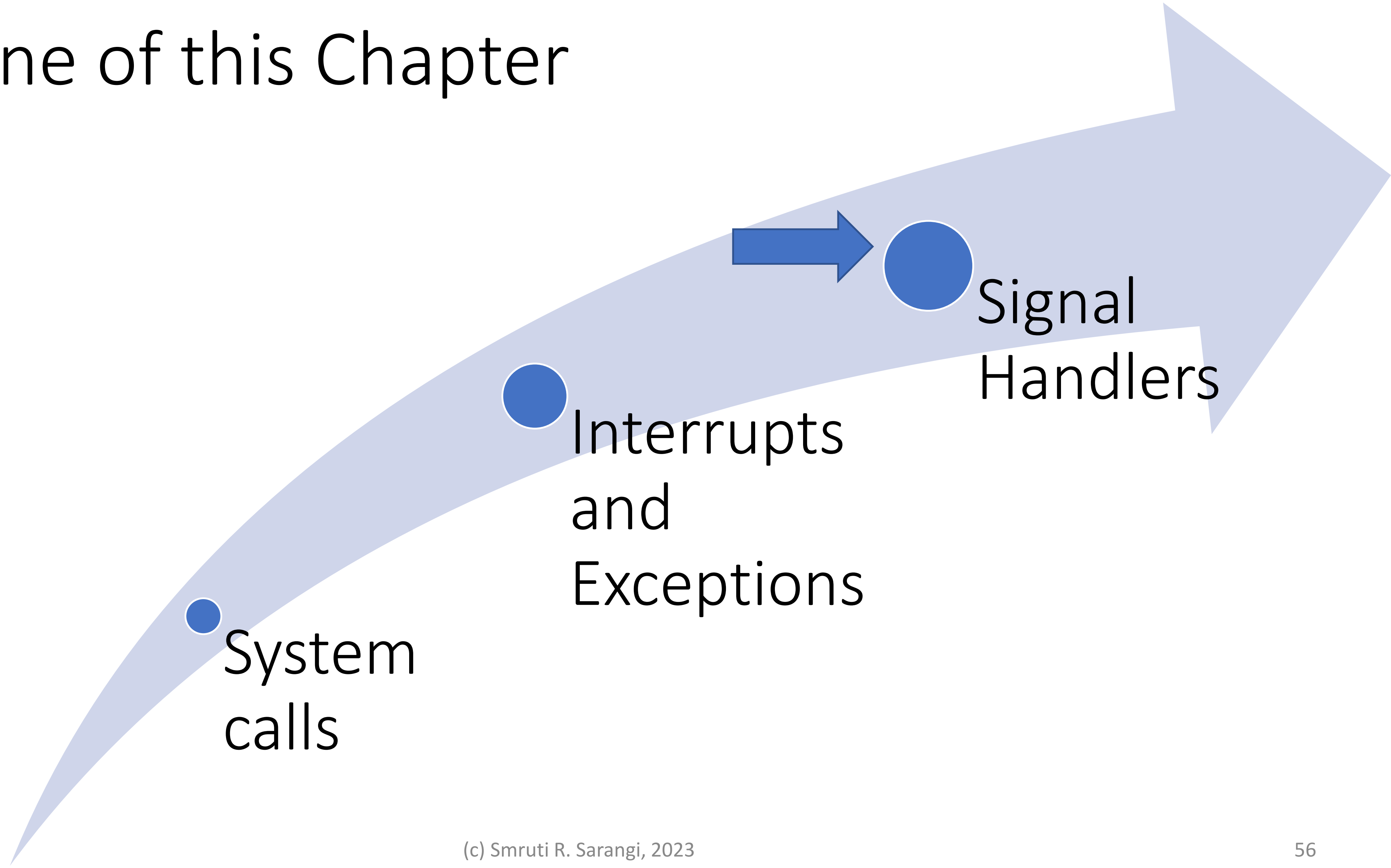
/include/linux/notifier.h

- Every event has a set of functions that are interested in it
- The *notify_die* function traverses a linked list can calls each function in sequence.
- They take appropriate action.
- For example for a HW breakpoint event, the HW breakpoint handler is invoked.

Return value of each function

Value	Meaning
NOTIFY_DONE	Don’t care about this event
NOTIFY_OK	Event handled. Don’t call any more functions
NOTIFY_STOP_MASK	Don’t call any more functions
NOTIFY_BAD	Something went wrong. Stop calling functions.

Outline of this Chapter



Example Code with Signal Handling

```
void handler (int sig){
   printf ("In the signal handler of process %d \n", getpid());
   exit(0);
}

int main(){
   pid_t child_pid, wpid; int status;

   signal (SIGUSR1, handler);
   child_pid = fork();

   if (child_pid == 0) {
      printf ("I am the child and I am stuck \n");
      while (1) {}
   } else {
      sleep (2);
      kill (child_pid, SIGUSR1);
      wpid = wait (&status);
      printf ("Parent exiting, child = %d, wpid = %d,  status = %d \n",child_pid, wpid, status);
   }
}
```

Child stuck in an infinite loop

Parent signals the child and waits for it to exit

The output

```
I am the child and I am stuck
In the signal handler of process 1078
Parent exiting, child = 1078, wpid = 1078,  status = 0
```

The *sleep* statement ensures that the child is setup and it prints that it is stuck.

- The signal handler correctly prints the child process's id.

Parent's side

- In normal circumstances you expect the child to exit correctly (status = 0) and wait() to return the *pid* of the child whose exit status it is returning

Common Signals

Signal	Number	Signal	Number
SIGHUP	1	SIGSEGV	11
SIGINT	2	SIGUSR2	12
SIGQUIT	3	SIGPIPE	13
SIGILL	4	SIGALRM	14
SIGTRAP	5	SIGTERM	15
SIGABRT	6	SIGSTKFLT	16
SIGBUS	7	SIGCHLD	17
SIGFPE	8	SIGCONT	18
SIGKILL	9	SIGSTOP	19
SIGUSR1	10	SIGTSTP	20

Sending a Signal to a Process

kill(...)
- Send a signal to a thread group

tkill(...)
- Send a signal to a specific thread

tgkill
- Send a signal to a specific thread in a thread group (protects against thread ids being reused)

- If the process is not executing when a signal is sent to it, then it is queued by the kernel and delivered later
- Signals can be blocked and unblocked
- A pending signal is generated but not delivered
- We cannot have two pending signals of the same type pending for a process.
- When a signal handler is executing, it blocks the corresponding signal
- SIGKILL signals are sent to all the threads in a group. Other signals that are sent to a group are handled by any one thread. Signal handlers are shared in a group.

Delivery of a Signal to a Process

- Available options: either handle a signal or perform the default action

Action	Remarks
Ignore	It is an unimportant signal, and no handler is registered for it.
Terminate	Kill the process. Signals such as SIGINT
Dump	Along with termination create a core dump file that can be read by a debugger
Stop	Stop the process (example: SIGSTOP)
Resume	Resume a stopped process
Handle	Handle the signal if a handler is registered for it

SIGKILL an SIGSTOP cannot be ignored, handled, or blocked

Relevant Entries in *task_struct*

```
struct signal_struct      *signal;
struct sighand_struct     *sighand;
sigset_t                   blocked;
sigset_t                   real_blocked;
struct sigpending          pending;
unsigned long              sass_ss_sp;
size_t                     sass_ss_size;
```

`*signal`	Description of the signal
`*sighand`	Signal handlers
`blocked`, `real_blocked`	Blocked signals.
`pending`	Pending signals
`sass_ss_sp`, `sass_ss_size`	Stack size

Signal handlers by default use the process's stack. However, they can be made to execute on an alternative stack. The variables *sass_sp_size* and *sass_ss_size* are used to specify the alternative stack.

struct signal_struct

/include/linux/sched/signal.h

```
struct signal_struct {
    atomic_t               live;             /* number of active processes in the group */
    struct list_head       thread_head;      /* all the threads in the thread group */
    wait_queue_head_t      wait_chldexit;    /* processes waiting on the wait system call */
    struct task_struct     *curr_target;     /* last thread that received a signal */
    struct sigpending      shared_pending;   /* shared list of pending signals in the group */
} ;
```

- This is shared by all the processes that are a part of the same thread group
- Linux treats a thread group as a whole unit insofar as signals as concerned
- A lot of signals are sent to a thread group; a thread within the group handles it.

Signal Handler

```
struct sighand_struct {
        refcount_t              count;
        wait_queue_head_t       signalfd_wqh;
        struct k_sigaction      action[_NSIG];
};
```

- This represents a signal handler.
- All the threads in the thread group share the same sighand_struct
  - count maintains the number of task_structs that point to this handler
  - One mechanism of handling a signal is by a file
- _NSIG = 64 in x86 machines
- We store an array of k_sigaction structures

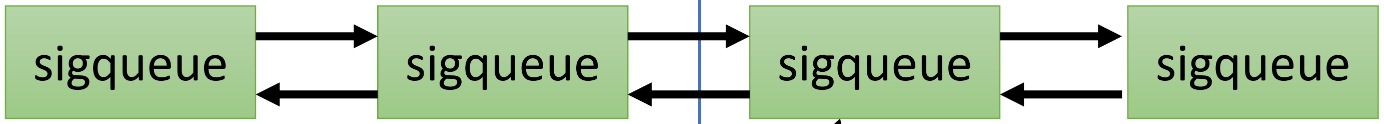
struct sigaction

```
struct sigaction {
        __sighandler_t sa_handler;
        unsigned long sa_flags;
        sigset_t sa_mask;
};
```

- sa_handler is the function pointer
- sa_mask is the signal mask (for the currently masked signals)

struct sigpending

```
struct sigpending {
        struct list_head list;
        sigset_t signal;
}
```



- It contains a list of sigqueue data structures

```
struct sigqueue {
        struct list_head list;        /* Pointing to its current position in the
                                                 queue of sigqueues */
        kernel_siginfo_t info;     /* signal number, signal source, etc. */
}
```

kernel_siginfo_t

```
struct {
        int si_signo;                   /* signal number */
        int si_errno;                   /* error number */
        int si_code;                    /* source of the signal */
        union __sifields _sifields;     /* sender's pid + more */
}
```

Storing the User’s Context (in the User’s Stack)

Stored on the user’s stack

```
struct rt_sigframe {
        char __user *  pretcode;        /* return address: __restore_rt glibc function */
        struct ucontext uc;             /* context */
        struct siginfo info;            /* kernel_siginfo_t */
};
```

User process’s context

```
struct ucontext {
        unsigned long        uc_flags;
        stack_t              uc_stack;       /* user’s stack pointer */
        struct sigcontext    uc_mcontext;
};
```

Snapshot of all the registers and the user process’s state

Returning from a Signal Handler

- We return to the __restore_rt glibc function
  - Recall: its address was pushed to the user's stack
- The __restore_rt function makes a system call sigreturn

1. Follows the reverse set of steps
2. Uses the *copy_from_user* function to copy the signal context to kernel space.
3. Restores the context of the user process. The kernel can only populate the registers.

srsarangi@cse.iitd.ac.in