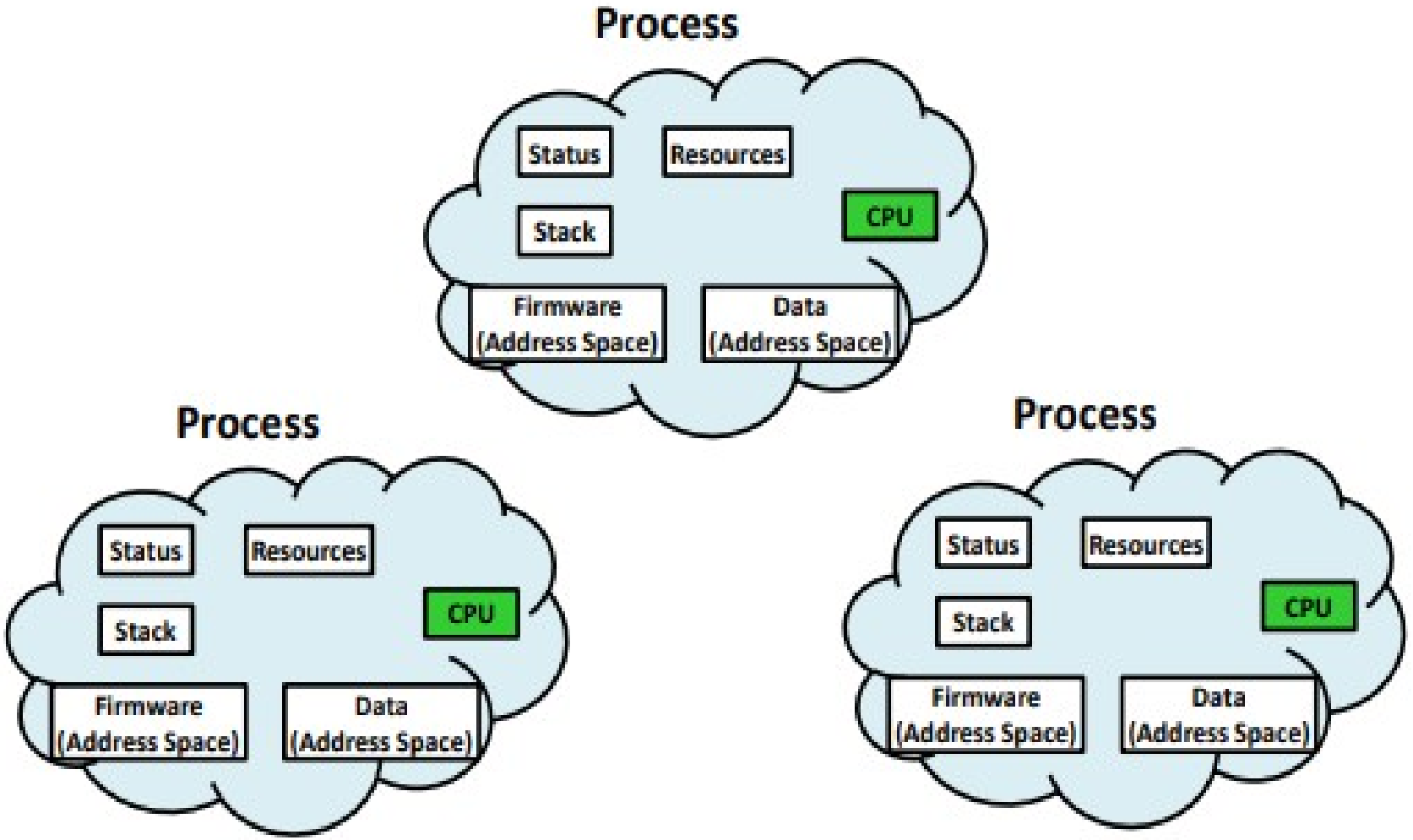
Process, threads and interrupts

Chapters 3, 7 (Monday class) and 8 in “Linux Kernel Development” by Robert Love

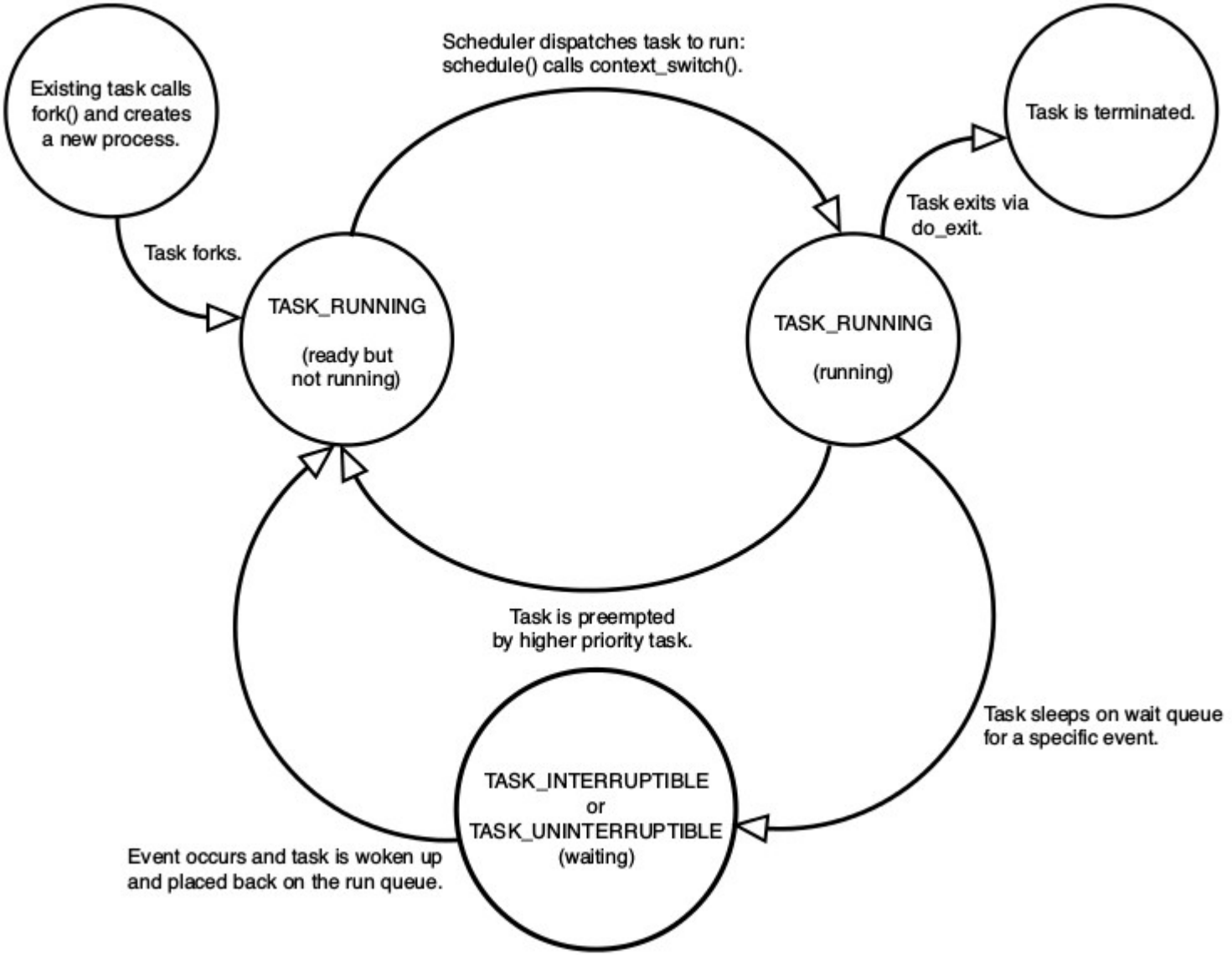
Process

- A program in execution, living result of running program code, also called task
  - executes program code (also called the text section)
  - data section containing global variables
  - set of resources like open files, pending signals, internal kernel data, processor state
  - memory address space with one or more memory mappings
  - one or more threads of execution

In pictures


Process
Status
Resources
Stack
CPU
Firmware (Address Space)
Data (Address Space)
Process
Status
Resources
Stack
CPU
Firmware (Address Space)
Data (Address Space)
Process
Status
Resources
Stack
CPU
Firmware (Address Space)
Data (Address Space)

Process State



Creating and terminating processes

- **fork()** system call creates a new process by duplicating an existing one
- The process that calls fork() is the parent, whereas the new process is the child
- The parent resumes execution and the child starts execution at the same place: where the call to fork() returns.
- Often, immediately after a fork it is desirable to execute a new, different program. **exec()** creates a new address space and loads a new program into it.
- Finally, a program exits via the **exit()** system call.This function terminates the process and frees all its resources.

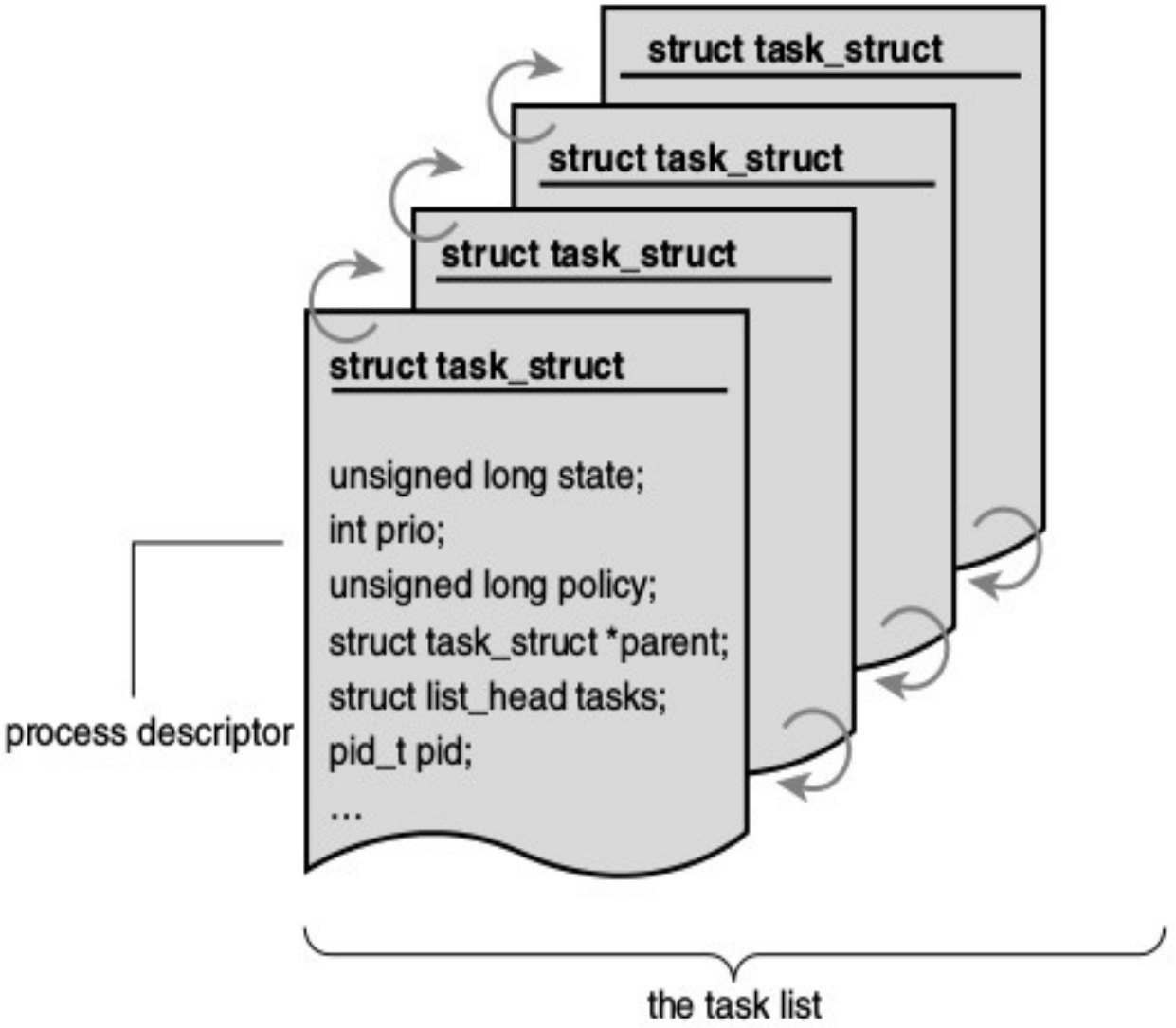
fork() implemented with clone(), clone() -> do_fork() -> copy_process() <kernel/fork.c>

- calls dup_task_struct() , which creates a new kernel stack, thread_info structure, and task_struct for the new process. At this point, the child and parent process descriptors are identical.
- checks new child will not exceed the limits on the number of processes for current user.
- various members of the process descriptor are cleared or set to initial values. The bulk of the values in task_struct remain unchanged.
- The child's state is set to TASK_UNINTERRUPTIBLE to ensure that it does not yet run
- copy_process() calls copy_flags() to update the flags member of the task_struct .The PF_SUPERPRIV flag, which denotes whether a task used superuser privileges, is cleared.The PF_FORKNOEXEC flag, which denotes a process that has not called exec() , is set.
- calls alloc_pid() to assign an available PID to the new task.
- Depending on the flags passed to clone() , copy_process() either duplicates or shares open files, filesystem information, signal handlers, process address space, and namespace.These resources are typically shared between threads in a given process; otherwise they are unique and thus copied here.
- copy_process() cleans up and returns to the caller a pointer to the new child.
- back in do_fork() , if copy_process() returns successfully, the new child is woken up and run.

Process Descriptor and Task List

- Circular doubly linked list called ***task list***
- Each element in the task list is a ***process descriptor***
- Process descriptor is of the type ***struct task_struct***, defined in <linux/sched.h>
- task_struct is ~1.7 KB on a 32 bit machine
- Process descriptor contains all information about a specific process
  - process identification value or PID (unique)
  - open files
  - the process's address space
  - pending signals
  - the process's state
  - and much more.

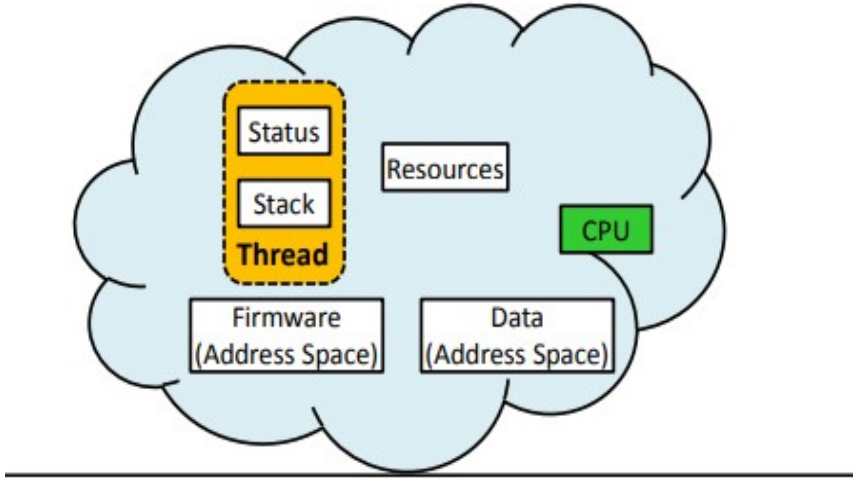
Process Descriptor and Task List

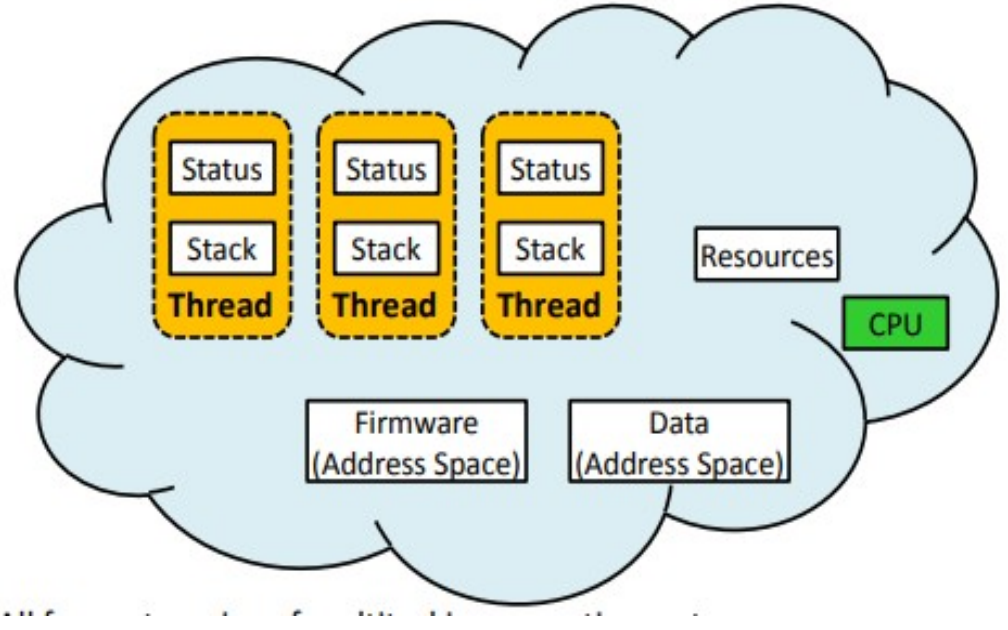


Threads

- Threads are a popular modern programming abstraction.
- Threads enable ***concurrent*** programming and on multi-processor systems ***parallelism***
- Obejct of activity within process
- Each thread includes
  - unique program counter
  - process stack
  - set of processor registers
- The kernel schedules individual threads, not processes

In pictures


Status
Stack
Thread
Resources
CPU
Firmware
(Address Space)
Data
(Address Space)


Status
Stack
Thread
Status
Stack
Thread
Status
Stack
Thread
Resources
CPU
Firmware
(Address Space)
Data
(Address Space)

Linux implementation of threads

- Linux implements all threads as standard processes. A thread is merely a process, that shares resources with other processes.
- No special scheduling semantics or data structures to represent threads.

Creating Threads

- Threads are created as normal tasks
- **clone() system call** is passed flags corresponding to the specific resources to be shared

```
clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);
```

- Identical to normal fork(), except that address space, file system resources, file descriptors and signal handlers are shared
- The new task and its parent are what are popularly called *threads*

What to share with child process/ thread

<linux/sched.h>

Flag	Meaning
CLONE_FILES	Parent and child share open files.
CLONE_FS	Parent and child share filesystem information.
CLONE_IDLETASK	Set PID to zero (used only by the idle tasks).
CLONE_NEWNS	Create a new namespace for the child.
CLONE_PARENT	Child is to have same parent as its parent.
CLONE_PTRACE	Continue tracing child.
CLONE_SETTID	Write the TID back to user-space.
CLONE_SETTLS	Create a new TLS for the child.
CLONE_SIGHAND	Parent and child share signal handlers and blocked signals.
CLONE_SYSVSEM	Parent and child share System V SEM_UNDO semantics.
CLONE_THREAD	Parent and child are in the same thread group.
CLONE_VFORK	vfork() was used and the parent will sleep until the child wakes it.
CLONE_UNTRACED	Do not let the tracing process force CLONE_PTRACE on the child.
CLONE_STOP	Start process in the TASK_STOPPED state.
CLONE_SETTLS	Create a new TLS (thread-local storage) for the child.
CLONE_CHILD_CLEARTID	Clear the TID in the child.
CLONE_CHILD_SETTID	Set the TID in the child.
CLONE_PARENT_SETTID	Set the TID in the parent.
CLONE_VM	Parent and child share address space.

Kernel Threads

<linux/kthread.h>

- Standard processes existing only in the kernel space
- Do not have an address space, mm pointer that points to address space is null
- Opearte only in the kernel space and do not context switch into the user space
- Schedulable and preemptible, just like normal processes
- Linux delegates several tasks to kernel threads, most notably the flush tasks and the **ksoftirqd** task.
- We can see the kernel threads by running the command

  ps -ef

Process, threads and interrupts

Chapters 3, 7 (Monday class) and 8 in “Linux Kernel Development” by Robert Love

Top Half vs. Bottom Half

These two goals of an interrupt handler conflict with one another

- Execute quickly
- Perform a large amount of work

So the processing of interrupts is split into two parts, or halves

Top half

The interrupt handler is the top half. The top half is run immediately upon receipt of the interrupt and performs only the work that is time-critical, such as acknowledging receipt of the interrupt or resetting the hardware.

Bottom half

Work that can be performed later is deferred until the bottom half. The bottom half runs in the future, at a more convenient time, with all interrupts enabled.

Example using network card

- When network cards receive packets from the network, the network cards immediately issue an interrupt. This optimizes network throughput and latency and avoids timeouts.
- The kernel responds by executing the network card's registered interrupt.
- The interrupt runs, acknowledges the hardware, copies the new networking packets into main memory, and readies the network card for more packets. These jobs are the important, time-critical, and hardware-specific work.
- The kernel generally needs to quickly copy the networking packet into main memory because the network data buffer on the networking card is fixed and miniscule in size, particularly compared to main memory. Delays in copying the packets can result in a buffer overrun, with incoming packets overwhelming the networking card's buffer and thus packets being dropped.
- After the networking data is safely in the main memory, the interrupt's job is done, and it can return control of the system to whatever code was interrupted when the interrupt was generated.
- The rest of the processing and handling of the packets occurs later, in the bottom half

Bottom halves and deferred work

- Interrupts are enabled in bottom half (unlike interrupt handler)
- Softirq and tasklets cannot sleep
- Work queues can sleep
- Softirq needs proper locking (as different processors can execute same softirq code)

Bottom Half	Status	
BH	Removed in 2.5	
Task queues	Removed in 2.5	
Softirq	Available since 2.3	Networking and block devices
Tasklet	Available since 2.3	
Work queues	Available since 2.5	

Softirq <kernel/softirq.c>

- static struct softirq_action softirq_vec[NR_SOFTIRQS];
- 32 is the limit, only nine exist

Tasklet	Priority	Softirq Description
HI_SOFTIRQ	0	High-priority tasklets
TIMER_SOFTIRQ	1	Timers
NET_TX_SOFTIRQ	2	Send network packets
NET_RX_SOFTIRQ	3	Receive network packets
BLOCK_SOFTIRQ	4	Block devices
TASKLET_SOFTIRQ	5	Normal priority tasklets
SCHED_SOFTIRQ	6	Scheduler
HRTIMER_SOFTIRQ	7	High-resolution timers
RCU_SOFTIRQ	8	RCU locking

Handling softirq

- Assign a sofirq index
- Register handler: the softirq handler is registered at run-time via open_softirq(), which takes two parameters: the softirq's index and its handler function.The networking subsystem, for example, registers its softirqs like this, in net/core/dev.c :
  - open_softirq(NET_TX_SOFTIRQ, net_tx_action);
  - open_softirq(NET_RX_SOFTIRQ, net_rx_action);
- A registered softirq must be marked before it will execute.This is called **raising the softirq**. Usually, an interrupt handler marks its softirq for execution before returning. For example, the networking subsystem would call,
  - raise_softirq(NET_TX_SOFTIRQ);
- Then, at a suitable time, the softirq runs. Pending softirqs are checked for and executed in the following places:
  - In the return from hardware interrupt code path
  - In the ksoftirqd kernel thread
  - In any code that explicitly checks for and executes pending softirqs, such as the net-working subsystem

Kernel threads run softirq, tasklets

- In designing softirqs, the kernel developers realized that some sort of compromise was needed.
- The solution ultimately implemented in the kernel is to not immediately process

  reactivated softirqs. Instead, if the number of softirqs grows excessive, the kernel wakes up

  a family of kernel threads to handle the load.
- The kernel threads run with the lowest possible priority (nice value of 19), which ensures they do not run in lieu of anything important.
- This concession prevents heavy softirq activity from completely starving user-spaceof processor time. Conversely, it also ensures that “excess” softirqs do run eventually.
- Finally, this solution has the added property that on an idle system the softirqs are handled rather quickly because the kernel threads will schedule immediately.
- There is one thread per processor.The threads are each named ksoftirqd/n where n is the processor number. On a two-processor system, you would have ksoftirqd/0 and ksoftirqd/1 . Having a thread on each processor ensures an idle processor, if available, can always service softirqs.

Checking for softirq-s

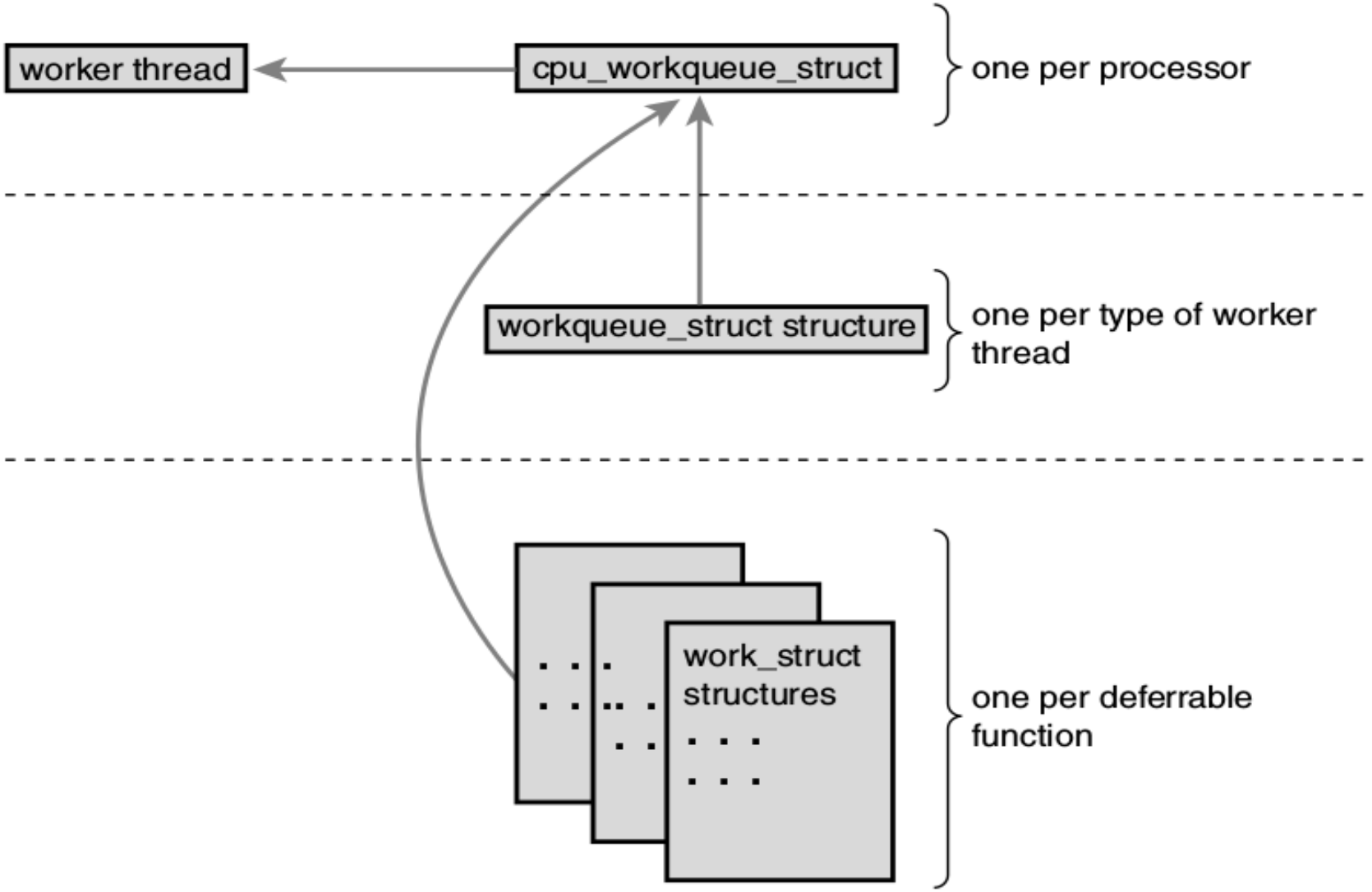
- Regardless of the method of invocation, softirq execution occurs in __do_softirq(), which is invoked by do_softirq()

```
u32 pending;

pending = local_softirq_pending();
if (pending) {
    struct softirq_action *h;

    /* reset the pending bitmask */
    set_softirq_pending(0);

    h = softirq_vec;
    do {
        if (pending & 1)
            h->action(h);
        h++;
        pending >>= 1;
    } while (pending);
}
```

Work Queues

- If we need a schedulable entity to perform bottom-half processing, we need work queues.
- They are the only bottom-half mechanisms that run in process context, so the only ones that can sleep.
- Useful for situations in which we
  - need to allocate a lot of memory
  - obtain a semaphore
  - perform block I/O
- In its most basic form, the work queue subsystem is an interface for creating kernel threads to handle work queued from elsewhere.These kernel threads are called **worker threads**.

Work Queues


worker thread
cpu_workqueue_struct
one per processor
workqueue_struct structure
one per type of worker thread
work_struct structures
one per deferrable function