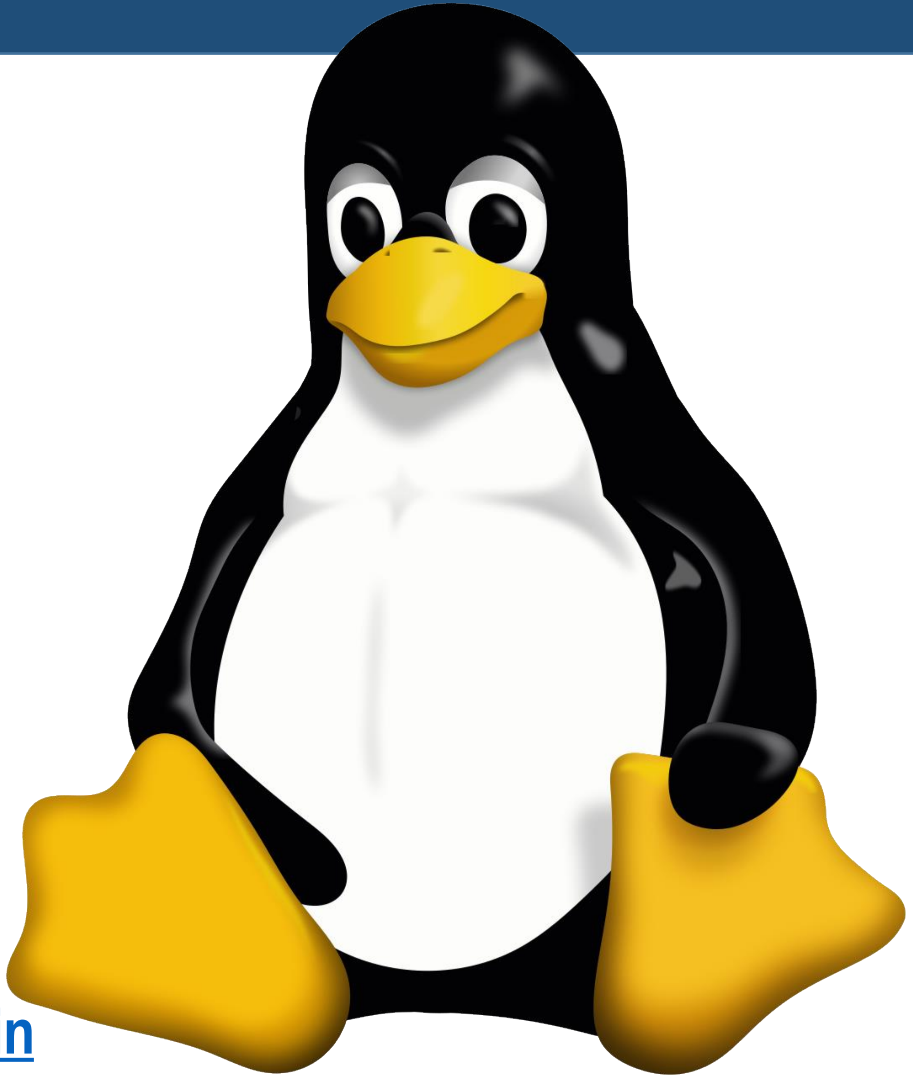
CS 60038: Advances in Operating Systems Design
Lecture 3: Linux Kernel Overview and Process Management

Department of Computer Science and Engineering

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

Sandip Chakraborty
sandipc@cse.iitkgp.ac.in



In Today's Class..

- General Kernel Responsibilities
  - Resource Abstraction
  - Sharing of Resources
  - OS functions
- Kernel Organization
  - Interrupts
  - Kernel Services
  - Serial Execution
  - Daemons
  - Booting Procedure
  - Logging in
  - Control flow
- Process Management

Majority of the slides have been taken from the CSNB334 Advanced Operating Systems course at UNITEN Malaysia. Thanks to Prof. Abdul Rahim Ahmad for making them available online.

Kernel Responsibilities

- OS based on UNIX always divides OS functionality into two components:
  - Kernel – executes in supervisor mode
  - OS components – executes in user mode
- Kernel responsibility
  - Abstract and manage hardware
  - Manage sharing of resources among executing programs
    - Support processes, files and other resources
    - Managed by system calls
- Linux differs from other systems that use UNIX in :
  - The data structures and
  - The algorithms used.

Resource Abstraction

- Creation of software to simplify operations on hardware
  - Hides the details of hardware operations needed to control it.
  - Other software can use the abstraction without knowing the details of how the hardware is managed.
  - E.g. device driver.

- In Linux, abstractions are into
  - Processes
    - A process is an abstraction of the CPU operation that executes an object program placed in the RAM.
  - Resources

- Linux is a multiprogrammed OS where:
  - CPU is time-multiplexed
  - Memory is space-multiplexed
    - Divided into blocks called memory partitions.

Resource Management

- Procedure for creating resource abstractions, and allocating and deallocating of system resources to and from processes as they execute

- Resource – any logical component that the OS manages and can be obtained only by having the process request the use of the resource from the OS.
    - E.g: memory (RAM), storage, CPU, files.

Resource Sharing

- Processes request, use and release resources
- Therefore, the OS needs to
  - manage competition for resources.
  - assure exclusive use of resources.
    - CPU – by disabling interrupt
    - RAM – by hardware memory protection mechanism
    - Devices – by preventing CPU from executing I/O instructions unless on behalf of processes allocated.
- Implementation of exclusive use.
  - Partly hardware, partly software.
  - Hardware knows when OS is getting CPU use – using mode bit (supervisor(or kernel)/user)

Allowing sharing of Resources

- Sometimes, processes need to share a resource
  - Eg. Process that reads a database with one that writes into it.
- Sharing violates exclusive use mechanism
- Mechanism for resource sharing adds complexity to resource managers design.

OS Function Partitioning

- 4 categories of functions:
  - Process and Resource management cpu.
    - gives illusion of multiple virtual machines.
  - Memory Management
    - Allocation to processes
    - Protection from unauthorized access.
    - swapping in virtual memory system.
  - Device Management
    - Provides controlling functions to many devices
  - File Management
    - Provides organization and storage abstractions

Kernel Organization

- Linux kernel is monolithic
  - Implemented in a single executable module
- Data structure for any aspect of the kernel is available to any other part of the kernel except normally
  - device drivers
  - Interrupt handlers

Philosophy:

- Main part of kernel should never change
- New, kernel space software can be added for device management as devices can change.

Linux Kernel

- To account for advanced devices (esp. bitmap displays and networks)
  - Linux uses **modules** as containers to implement extensions to the main part of the kernel.

- Module – an independent software unit that can be designed and implemented after the kernel and dynamically loaded into kernel space.

- Module interface is more general than a device driver interface
  - Giving more flexibility to the system programmer to extend the kernel functionality.

- Module can implement device drivers.

Interrupt

- To interrupt kernel for some service request
  - By system call from processes (Software interrupt)
    - `CLI, STI` - disable/enable interrupts
  - From hardware components (Hardware interrupt)

- Hardware Interrupt is an electronic signal from device to CPU
  - From devices that finished some work
  - From timer etc.

Processing an Interrupt

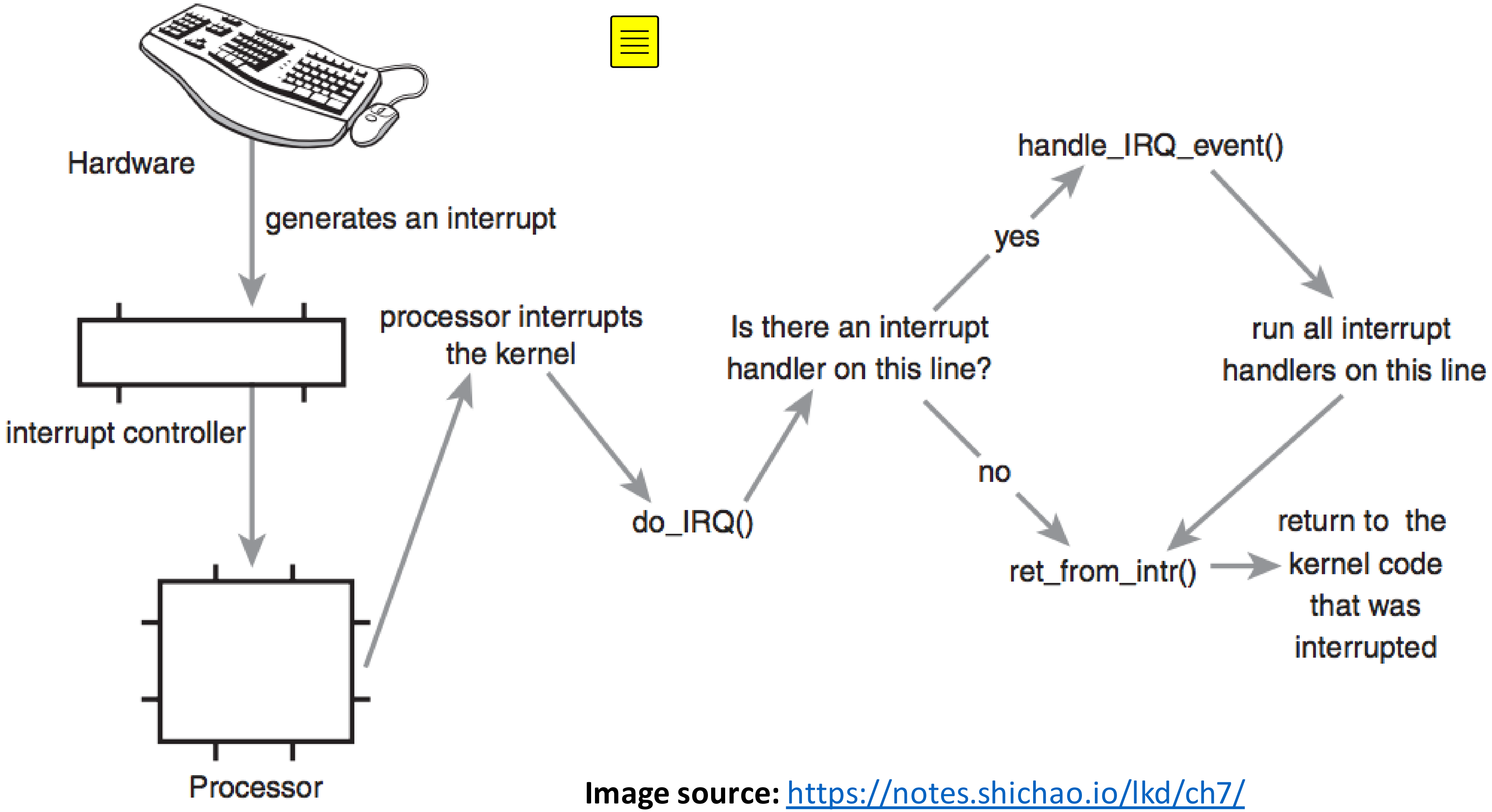

Hardware
generates an interrupt
interrupt controller
Processor
processor interrupts the kernel
do_IRQ()
Is there an interrupt handler on this line?
yes
handle_IRQ_event()
run all interrupt handlers on this line
no
ret_from_intr()
return to the kernel code that was interrupted

Image source: https://notes.shichao.io/lkd/ch7/

Kernel Services

- Kernel provides services by POSIX’s defined system call interface specification but implementation of this specification differs (b/w different versions of Linux).

- Implementation can be anywhere:
  - In kernel
  - In module
  - or, sometimes even in user-space programs
    - E.g. the thread package is implemented in a library code.

- System call scenario
  - A process (running in user mode) needing kernel services can do system call.
  - During call, it is running in kernel mode. Switch to kernel mode normally done by **a trap** instruction through a stub procedure
  - Upon completion, switch back to user mode

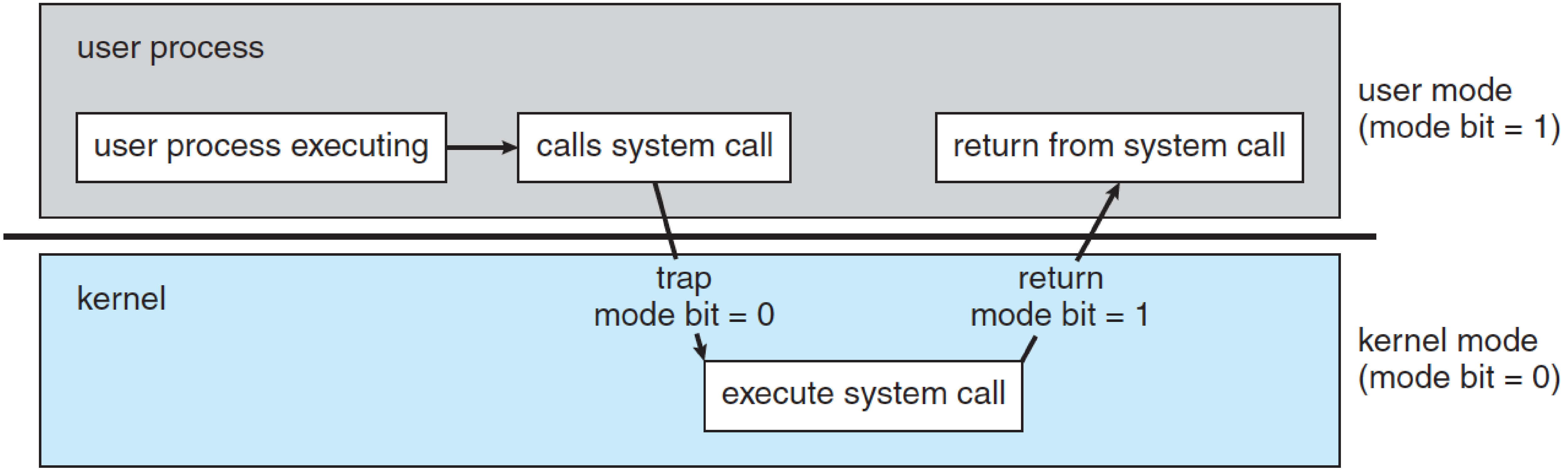
The dilemma?

- How can a user mode program switch the CPU to supervisor mode with the assurance that once the switch is done the CPU will be executing trusted kernel code and not untrusted user code?

The Path In and Out of the Kernel

- The *only* way to enter the operating kernel is to generate a processor interrupt. These interrupts come from several sources:
- *I/O devices:*
  - When a device, such as a disk or network interface, completes its current operation, it notifies the operating system by generating a processor interrupt.
- *Clocks and timers:*
  - Processors have timers that can be periodic (interrupting on a fixed interval) or count-down (set to complete at some specific time in the future). Periodic timers are often used to trigger scheduling decisions. For either of these types of timers, an interrupt is generated to get the operating system's attention.
- *Exceptions:*
  - When an instruction performs an invalid operation, such as divide-by-zero, invalid memory address, or floating point overflow, the processor can generate an interrupt.
- *Software Interrupts (Traps):*
  - Processors provide one or more instructions that will cause the processor to generate an interrupt.
  - Trap instructions are most often used to implement system calls and to be inserted into a process by a debugger to stop the process at a breakpoint.

The System Call Path



System Call Stub Functions

- The system call stub functions provide a high-level language interface to a function whose main job is to generate the software interrupt (trap) needed to get the kernel's attention.
- These functions are often called *wrappers*.
- The stub functions do the following:
  - set up the parameters,
  - trap to the kernel,
  - check the return value when the kernel returns, and
    - if no error: return immediately, else
    - if there is an error: set a global error number variable (called "errno") and return a value of -1.

The trap instruction

- A trap instruction is not a privileged instruction, so any program can execute a trap.

- However, the destination of the branch instruction is predetermined by a set of addresses that are kept in supervisory space and that are configured to point to kernel code

Sequence of steps............

1. Switch the CPU to supervisor mode.
2. Look up a branch address in a kernel space table.
3. Branch to a trusted OS function.

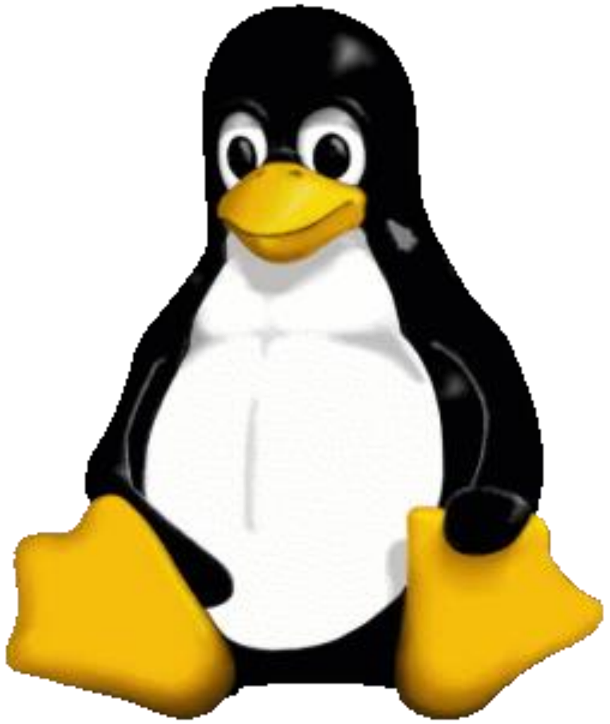
Because the trap instruction is compiled into the stub procedure, a user-space program cannot easily determine the trap's destination address.

Also, it cannot branch directly to the kernel function – only through the system-provided stub.

Daemons

- There is no special “kernel process” that executes the kernel code.
    - Kernel code is actually induced by normal user processes.
- Daemons are processes that do not belong to a user process.
- Started when machine is started
- Used for functioning in areas such as for
    - Processing network packets
    - Logging of system and error messages etc.
    - Normally, filename ends with d, such as
        - inetd, syslogd, crond, lpd etc

The Linux Kernel: Process Management



Process Descriptors

- The kernel maintains info about each process in a process descriptor, of type **`task_struct`**.
  - See **`include/linux/sched.h`**
  - Each process descriptor contains info such as run-state of process, address space, list of open files, process priority etc…

```
struct task_struct {
 volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
 unsigned long flags; /* per process flags */
 mm_segment_t addr_limit; /* thread address space:
                    0-0xBFFFFFFF for user-thead
                    0-0xFFFFFFFF for kernel-thread */
 struct exec_domain *exec_domain;
 long need_resched;
 long counter;
 long priority;
 /* SMP and runqueue state */
   struct task_struct *next_task, *prev_task;
   struct task_struct *next_run,  *prev_run;
   ...
 /* task state */
 /* limits */
 /* file system info */
 /* ipc stuff */
 /* tss for this task */
 /* filesystem information */
 /* open file information */
 /* memory management info */
 /* signal handlers */
   ...
};
```

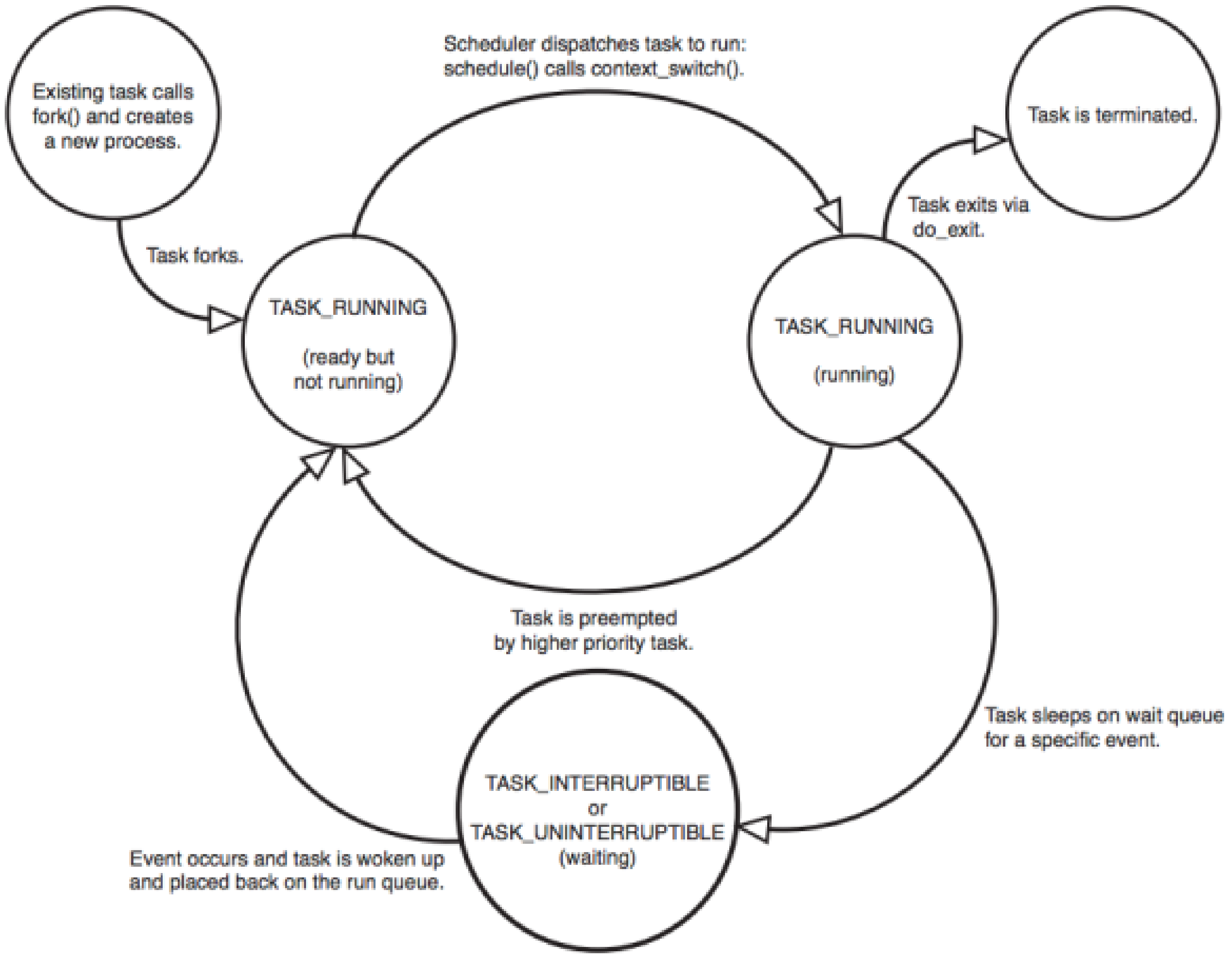
Contents of process descriptor

Process State

- Consists of an array of mutually exclusive flags
  - at least true for 2.2.x kernels.
  - implies exactly one **`state`** flag is set at any time.

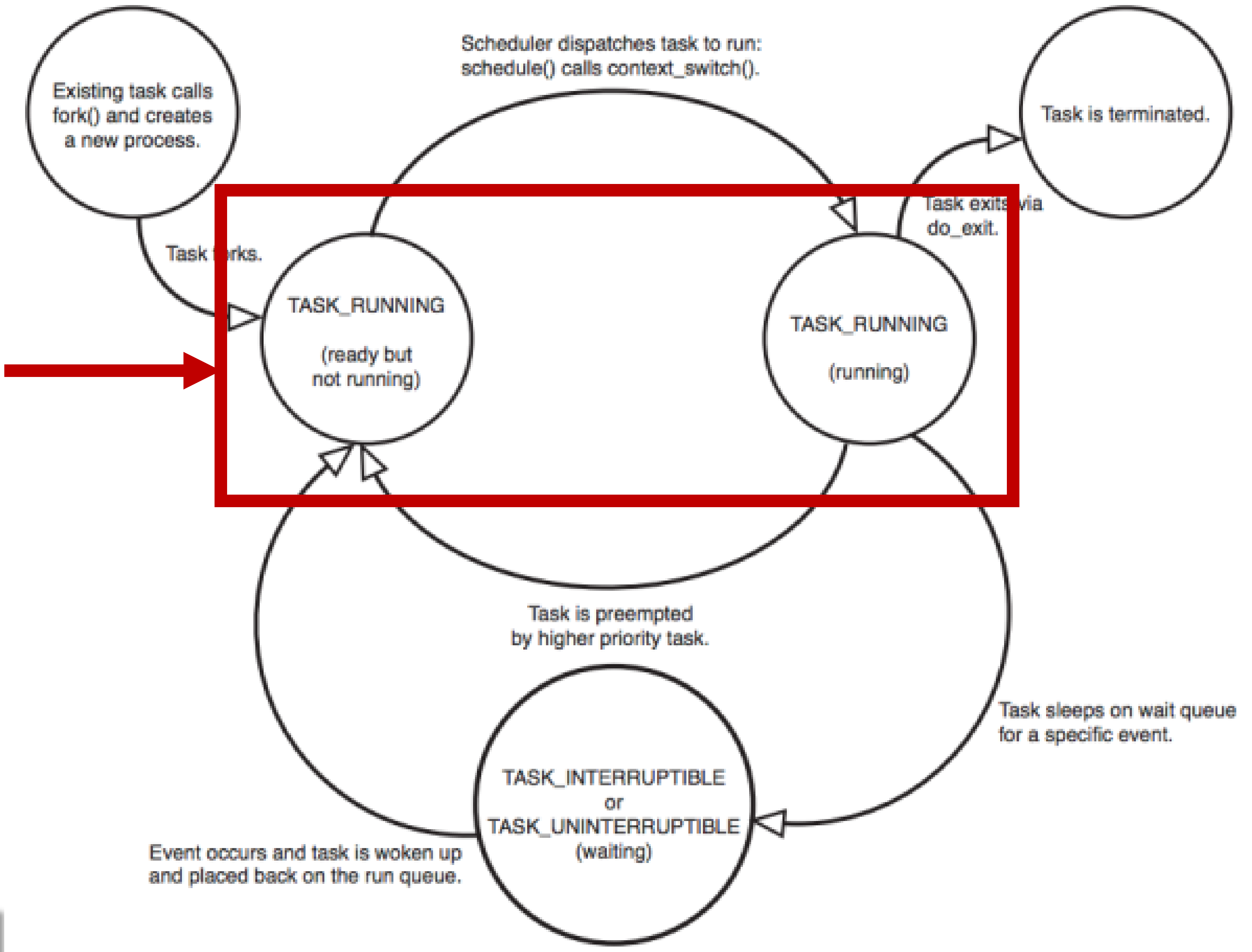
- **`state`** values:
  - **`TASK_RUNNING`** (executing on CPU or runnable).
  - **`TASK_INTERRUPTIBLE`** (waiting on a condition: interrupts, signals and releasing resources may "wake" process).
  - **`TASK_UNINTERRUPTIBLE`** (Sleeping process cannot be woken by a signal).
  - **`TASK_STOPPED`** (stopped process e.g., by a debugger).
  - **`TASK_ZOMBIE`** (terminated before waiting for parent).

Process States in Linux Kernel

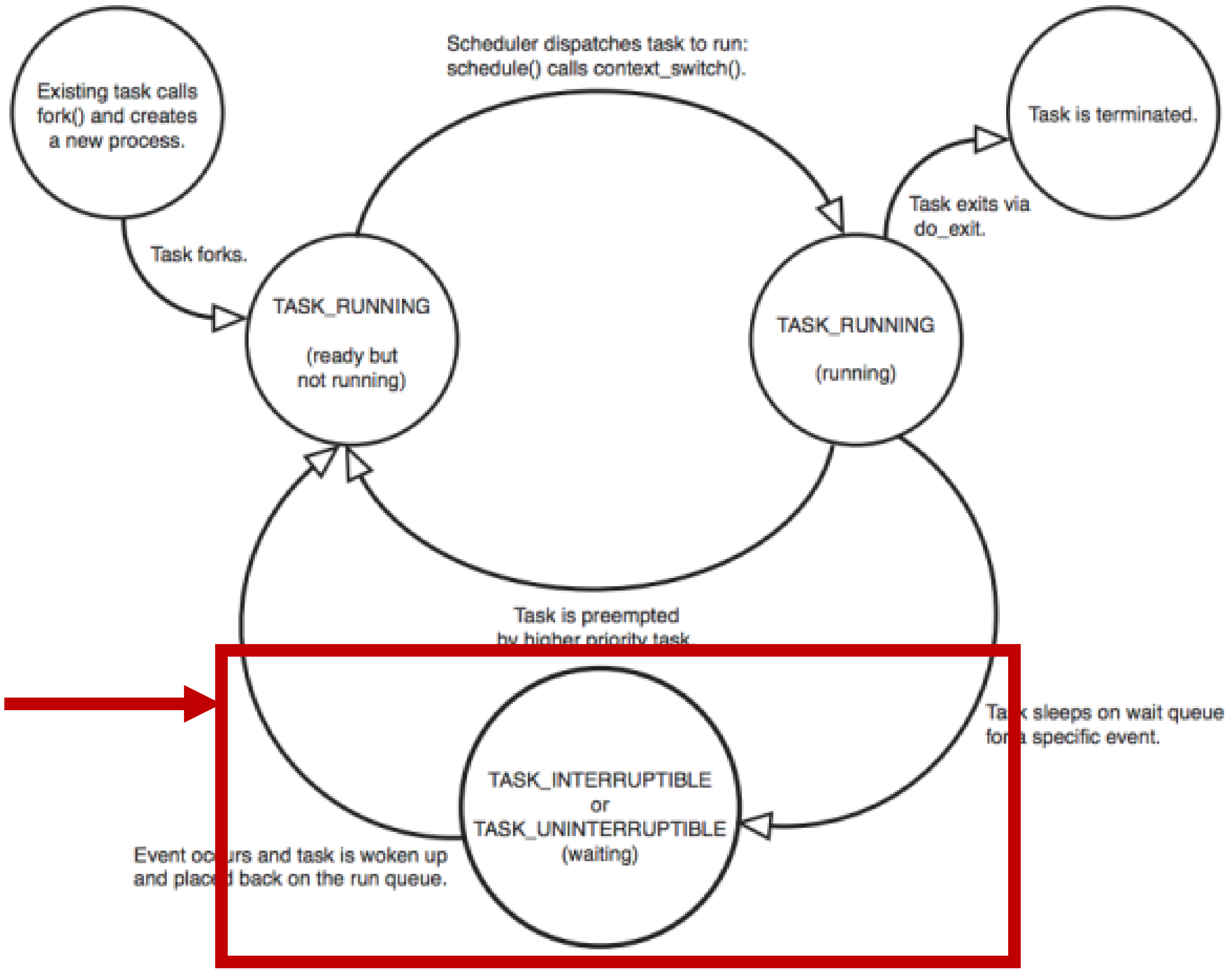

Existing task calls fork() and creates a new process.
Task forks.
TASK_RUNNING (ready but not running)
Scheduler dispatches task to run: schedule() calls context_switch().
TASK_RUNNING (running)
Task exits via do_exit.
Task is terminated.
Task is preempted by higher priority task.
Task sleeps on wait queue for a specific event.
TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE (waiting)
Event occurs and task is woken up and placed back on the run queue.

Process States in Linux Kernel

A process in user mode can only be in these states


Existing task calls fork() and creates a new process.
Task forks.
Scheduler dispatches task to run: schedule() calls context_switch().
Task is terminated.
Task exits via do_exit.
TASK_RUNNING (ready but not running)
TASK_RUNNING (running)
Task is preempted by higher priority task.
Task sleeps on wait queue for a specific event.
TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE (waiting)
Event occurs and task is woken up and placed back on the run queue.

Process States in Linux Kernel


Existing task calls fork() and creates a new process.
Task forks.
TASK_RUNNING (ready but not running)
Scheduler dispatches task to run: schedule() calls context_switch().
TASK_RUNNING (running)
Task exits via do_exit.
Task is terminated.
Task is preempted by higher priority task
Task sleeps on wait queue for a specific event.
TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE (waiting)
Event occurs and task is woken up and placed back on the run queue.

The user process moves to the kernel mode

Process Identification

- Each process, or independently scheduled execution context, has its own process descriptor.
- Process descriptor addresses are used to identify processes.
  - Process ids (or **PID**s) are 32-bit numbers, also used to identify processes.
  - For compatibility with traditional UNIX systems, LINUX uses PIDs in range 0..32767.
- Kernel maintains a **`task`** array of size **`NR_TASKS`**, with pointers to process descriptors. (Removed in 2.4.x to increase limit on number of processes in system).

Process Descriptor Storage

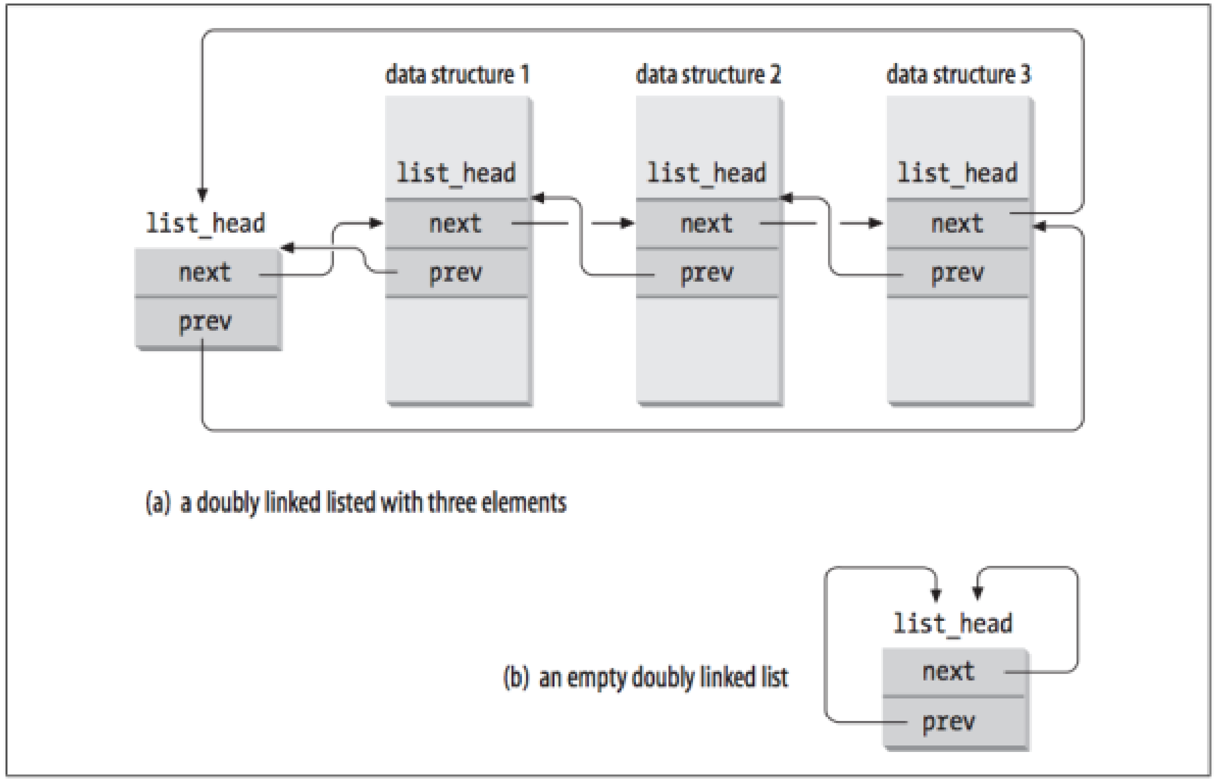
- Processes are *dynamic*, so descriptors are kept in dynamic memory.
- An 8KB memory area is allocated for each process, to hold process descriptor *and* kernel mode process stack.
  - **Advantage**: Process descriptor pointer of **`current`** (running) process can be accessed quickly from stack pointer.
  - 8KB memory area = 2^{13} bytes.
  - Process descriptor pointer = **`esp`** (register containing the stack pointer) with lower 13 bits masked.

Cached Memory Areas

- 8KB (**`EXTRA_TASK_STRUCT`**) memory areas are cached to bypass the kernel memory allocator when one process is destroyed and a new one is created.
- **`free_task_struct()`** and **`alloc_task_struct()`** are used to release / allocate 8KB memory areas to / from the cache.

The Process List

- The *process list* (of all processes in system) is a doubly-linked list.
  - **prev_task** & **next_task** fields of process descriptor are used to build list.
  - **init_task** (i.e., swapper) descriptor is at head of list.
    - **prev_task** field of **init_task** points to process descriptor inserted *last* in the list.
  - **for_each_task()** macro scans whole list.


data structure 1
data structure 2
data structure 3
list_head
next
prev
(a) a doubly linked listed with three elements
(b) an empty doubly linked list

The Run Queue

- Processes are scheduled for execution from a doubly-linked list of **`TASK_RUNNING`** processes, called the **`runqueue`**.
  - **`prev_run`** & **`next_run`** fields of process descriptor are used to build **`runqueue`**.
  - **`init_task`** heads the list.
  - **`add_to_runqueue()`**, **`del_from_runqueue()`**, **`move_first_runqueue()`**, **`move_last_runqueue()`** functions manipulate list of process descriptors.
  - **`NR_RUNNING`** macro stores number of runnable processes.
  - **`wake_up_process()`** makes a process runnable.

- **QUESTION:** Is a *doubly-linked list* the best data structure for a run queue?

Chained Hashing of PIDs

- PIDs are converted to matching process descriptors using a hash function.
  - A **`pidhash`** table maps PID to descriptor.
  - Collisions are resolved by chaining.
  - **`find_task_by_pid()`** searches hash table and returns a pointer to a matching process descriptor or **`NULL`**.

Managing the `task` Array

- The **`task`** array is updated every time a process is created or destroyed.
- A separate list (headed by **`tarray_freelist`**) keeps track of free elements in the **`task`** array.
  - When a process is destroyed its entry in the **`task`** array is added to the head of the freelist.

Wait Queues

- **`TASK_(UN)INTERRUPTIBLE`** processes are grouped into classes that correspond to specific events.
  - e.g., timer expiration, resource now available.
  - There is a separate wait queue for each class / event.
  - Processes are “woken up” when the specific event occurs.

Wait Queue Example

```
void sleep_on(struct wait_queue **wqptr) {
    struct wait_queue wait;
    current->state=TASK_UNINTERRUPTIBLE;
    wait.task=current;
    add_wait_queue(wqptr,&wait);
    schedule();
    remove_wait_queue(wqptr,&wait);
}
```

`sleep_on()` inserts the current process, **P**, into the specified wait queue and invokes the scheduler.

When **P** is awakened it is removed from the wait queue.

Process Switching

- Part of a process's execution context is its *hardware context* i.e., register contents.
  - The task state segment (**`tss`**) and kernel mode stack save hardware context.
    - **`tss`** holds hardware context not automatically saved by hardware (i.e., CPU).
- *Process switching* involves saving hardware context of **`prev`** process (descriptor) and replacing it with hardware context of **`next`** process (descriptor).
  - Needs to be fast!
  - Recent Linux versions override hardware context switching using software (sequence of **`mov`** instructions), to be able to validate saved data and for potential future optimizations.

The `switch_to` Macro

- **`switch_to()`** performs a process switch from the **`prev`** process (descriptor) to the **`next`** process (descriptor).
- **`switch_to`** is invoked by **`schedule()`** & is one of the most hardware-dependent kernel routines.
  - See **`kernel/sched.c`** and **`include/asm-*/system.h`** for more details.

Creating Processes

- Traditionally, resources owned by a parent process are duplicated when a child process is created.
  - *It is slow* to copy whole address space of parent.
    - *It is unnecessary*, if child (typically) immediately calls **`execve()`**, thereby replacing contents of duplicate address space.
- Cost savers:
  - *Copy on write* – parent and child share pages that are read; when either writes to a page, a new copy is made for the writing process.
  - *Lightweight processes* – parent & child share page tables (user-level address spaces), and open file descriptors.

Creating *Lightweight* Processes

- LWPs are created using `__clone()`, having 4 args:
  - **`fn`** – function to be executed by new LWP.
  - **`arg`** – pointer to data passed to **`fn`**.
  - **`flags`** – low byte=sig number sent to parent when child terminates; other 3 bytes=flags for resource sharing between parent & child.
    - **`CLONE_VM`**=share page tables (virtual memory).
    - **`CLONE_FILES`**, **`CLONE_SIGHAND`**, **`CLONE_VFORK`** etc…
  - **`child_stack`** – user mode stack pointer for child process.
- **`__clone()`** is a library routine to the **`clone()`** syscall.
  - **`clone()`** takes **`flags`** and **`child_stack`** args and determines, on return, the id of the child which executes the **`fn`** function, with the corresponding **`arg`** argument.

`fork()` and `vfork()`

- **`fork()`** is implemented as a **`clone()`** syscall with **`SIGCHLD`** sighandler set, all clone flags are cleared (no sharing) and **`child_stack`** is 0 (let kernel create stack for child on copy-on-write).
- **`vfork()`** is like **`fork()`** with **`CLONE_VM`** & **`CLONE_VFORK`** flags set.
  - With **`vfork()`** child & parent share address space; parent is blocked until child exits or executes a new program.

do_fork()

- **`do_fork()`** is called from **`clone()`**:
  - **`alloc_task_struct()`** is called to setup 8KB memory area for process descriptor & kernel mode stack.
  - Checks performed to see if user has resources to start a new process.
  - **`find_empty_process()`** calls **`get_free_taskslot()`** to find a slot in the **`task`** array for new process descriptor pointer.
  - **`copy_files/fs/sighand/mm()`** are called to create resource copies for child, depending on **`flags`** value specified to **`clone()`**.
  - **`copy_thread()`** initializes kernel stack of child process.
  - A new PID is obtained for child and returned to parent when **`do_fork()`** completes.

Kernel Threads

- Some (background) system processes run only in kernel mode.
  - e.g., flushing disk caches, swapping out unused page frames.
  - Can use *kernel threads* for these tasks.
- Kernel threads only execute kernel functions – normal processes execute these fns via syscalls.
- Kernel threads only execute in kernel mode as opposed to normal processes that switch between kernel and user modes.
- Kernel threads use linear addresses greater than PAGE_OFFSET – normal processes can access 4GB range of linear addresses.

Kernel Thread Creation

- Kernel threads created using:
  - `int kernel_thread(int (*fn)(void *), void *arg, unsigned long flags);`
  - `flags=CLONE_SIGHAND`, `CLONE_FILES` etc.

Process Termination

- Usually occurs when a process calls **`exit()`**.
  - Kernel can determine when to release resources owned by terminating process.
    - e.g., memory, open files etc.
- **`do_exit()`** called on termination, which in turn calls **`__exit_mm/files/fs/sighand()`** to free appropriate resources.
- Exit code is set for terminating process.
- **`exit_notify()`** updates parent/child relationships: all children of terminating processes become children of **`init`** process.
- **`schedule()`** is invoked to execute a new process.

Happy
Learning