Chapter 3. Processes

- We discuss some **properties of processes** and then describe how **process switching** is performed by the kernel.
- We also describe how Linux supports **multithreaded applications** relies on so-called **lightweight processes (LWP)**.
- The last two sections describe **how processes can be created and destroyed**.
- References:
  - Understanding Linux Kernel 2nd edition, Chapter 3
  - Linux Kernel Development 2nd edition, Chapter 3
  - Intel Documentation

Introduction

- The process is one of the **fundamental abstractions** in Unix operating systems, The other fundamental abstraction is files.

- Processes are, however, more than just the **executing program code** (often called the text section in Unix). They also include a **set of resources** such as open files and pending signals, internal kernel data, processor state, an address space, one or more threads of execution, and a data section containing global variables.

- Threads of execution, often shortened to threads, are the objects of activity within the process. **Each thread includes a unique program counter, process stack, and set of processor registers**. The kernel schedules individual threads, not processes.

- In traditional Unix systems, each process consists of one thread. In modern systems, however, **multithreaded programs**—those that consist of more than one thread—are common.

- **Linux has a unique implementation of threads**: It does not differentiate between threads and processes. To Linux, a thread is just a special kind of process.

Introduction

- On modern operating systems, processes provide two virtualizations: a virtualized processor and virtual memory.
  - The **virtual processor** gives the process the illusion that it alone monopolizes the system, despite possibly sharing the processor among dozens of other processes.
  - **Virtual memory** lets the process allocate and manage memory as if it alone owned all the memory in the system.
- Interestingly, note that **threads share the virtual memory** abstraction while each receives its own virtualized processor.
- Two or more processes can exist that are **executing the same program**.
- In fact, two or more processes can exist that **share various resources**, such as open files or an address space.

Create a new process

- In Linux, this occurs by means of the `fork()` system call, which 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 returns.
- **The `fork()` system call returns from the kernel twice: once in the parent process and again in the newborn child.**
- Often, immediately after a fork it is desirable to execute a new, different, program. The `exec*()` family of function calls is used to create a new address space and load a new program into it.

Terminate and remove

- Finally, a program exits via the exit() system call. This function terminates the process and frees all its resources.
- A parent process can inquire about the status of a terminated child via the wait4() system call, which enables a process to wait for the termination of a specific process.
- The kernel implements the wait4() system call. Linux systems, via the C library, typically provide the wait(),waitpid(),wait3() , and wait4() functions. All these functions return status about a terminated process, albeit with slightly different semantics.
- When a process exits, it is placed into a special zombie state that is used to represent terminated processes until the parent calls wait() or waitpid().

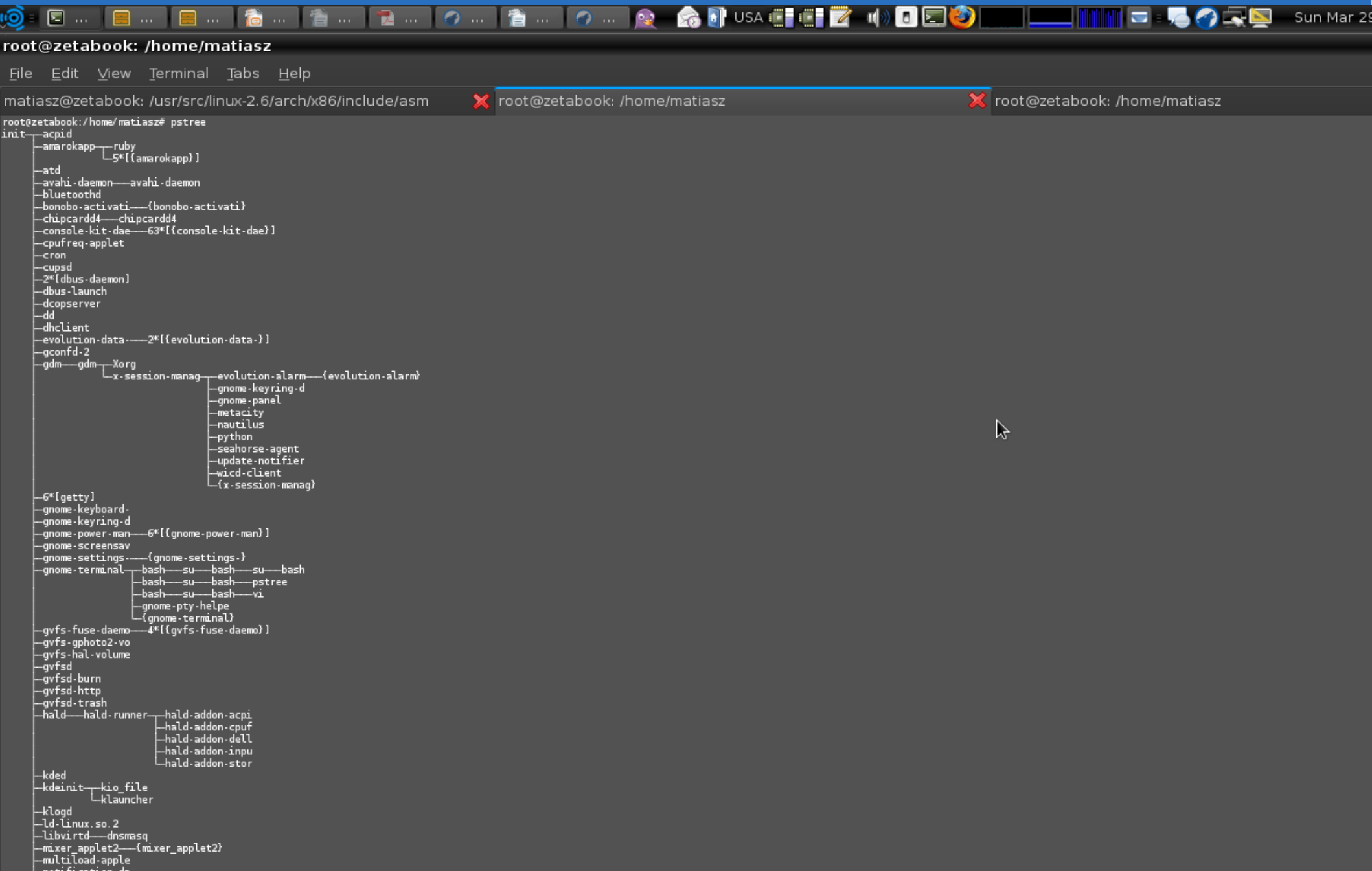
Example code

```
/* example fork exec together */
void main() {
  int pid;

  pid = fork();

  /* child executing ls program */
  if (pid == 0) {
    execl("/bin/ls", "ls", "-l", (char *)0);
  }

  /* parent waits for child to finish */
  if (pid > 0)
    wait((int *)0);
}
```

The Process Family Tree

- All processes are **descendents of the `init` process**, whose PID is **1**.
- The kernel starts `init` in the last step of the boot process.
- The `init` process, in turn, reads the system init-scripts and executes more programs, eventually completing the boot process.
- Every process on the system has exactly **one parent**.
- Likewise, every process has **zero or more children**.
- Processes that are all direct children of the same parent are called **siblings**.

Example of pstree



3.1. Processes, Lightweight Processes, and Threads

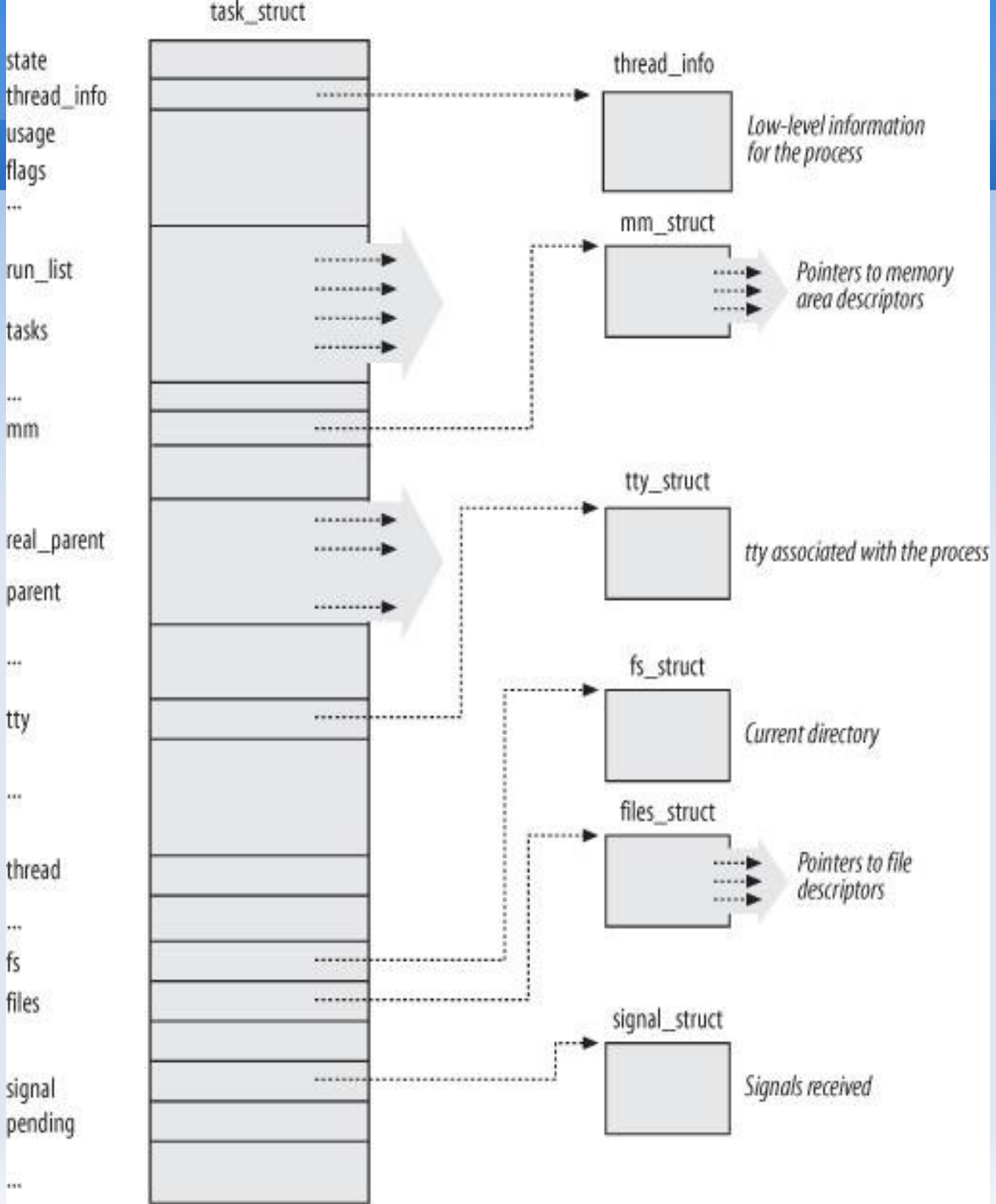
- From the kernel's point of view, the purpose of a process is to act as an entity to which system resources (CPU time, memory, etc.) are allocated.
- (Traditionally) When a process is created, it is **almost identical to its parent**.
  - It receives a (logical) **copy of the parent's address space** and **executes the same** code as the parent
  - they have **separate copies of the data (stack and heap)**, so that changes by the child to a memory location are invisible to the parent (and vice versa).
- While **earlier Unix kernels** employed this simple model, **modern Unix** systems do not.
- They support **multithreaded applications** user programs having many relatively independent execution flows sharing a large portion of the application data structures.
- Most multithreaded applications are written using standard sets of library functions called **pthread (POSIX thread)**.

Lightweight processes and threads

- Linux uses **lightweight processes** to offer better support for multithreaded applications.
- Basically, two lightweight processes **may share some resources**, like the address space, the open files, and so on. Whenever one of them modifies a shared resource, the other immediately sees the change.
- A straightforward way to implement multithreaded applications is to **associate a lightweight process with each thread**.
- Each thread can be **scheduled independently** by the kernel so that one may sleep while another remains runnable.
- **Examples of POSIX-compliant pthread libraries** that use Linux's lightweight processes are **LinuxThreads**, Native POSIX Thread Library (**NPTL**), and IBM's Next Generation Posix Threading Package(**NGPT**).
- multithreaded applications represented by **"thread groups":**
  - basically a **set of lightweight processes** that **act as a whole with regards to some system calls** such as getpid( ) , kill( ) , and _exit( ).

3.2. Process Descriptor

- To manage processes, **the kernel must have a clear picture of what each process is doing**.
  - the process's priority, whether it is running on a CPU or blocked on an event, what address space has been assigned to it, which files it is allowed to address, etc
- This is the role of the process descriptor a **task_struct type structure** whose fields contain **all the information related to a single process**.
- **The process descriptor is rather complex**.
  - In addition to a large number of fields containing process attributes, the process descriptor contains several **pointers to other data structures** that, in turn, contain pointers to other structures.
- Figure 3-1 describes the Linux process descriptor schematically. The six data structures on the right side of the figure refer to specific **resources owned by the process**.
- Most of these resources will be covered in future chapters. This chapter focuses on two types of fields that refer to the **process state** and to **process parent/child relationships.**

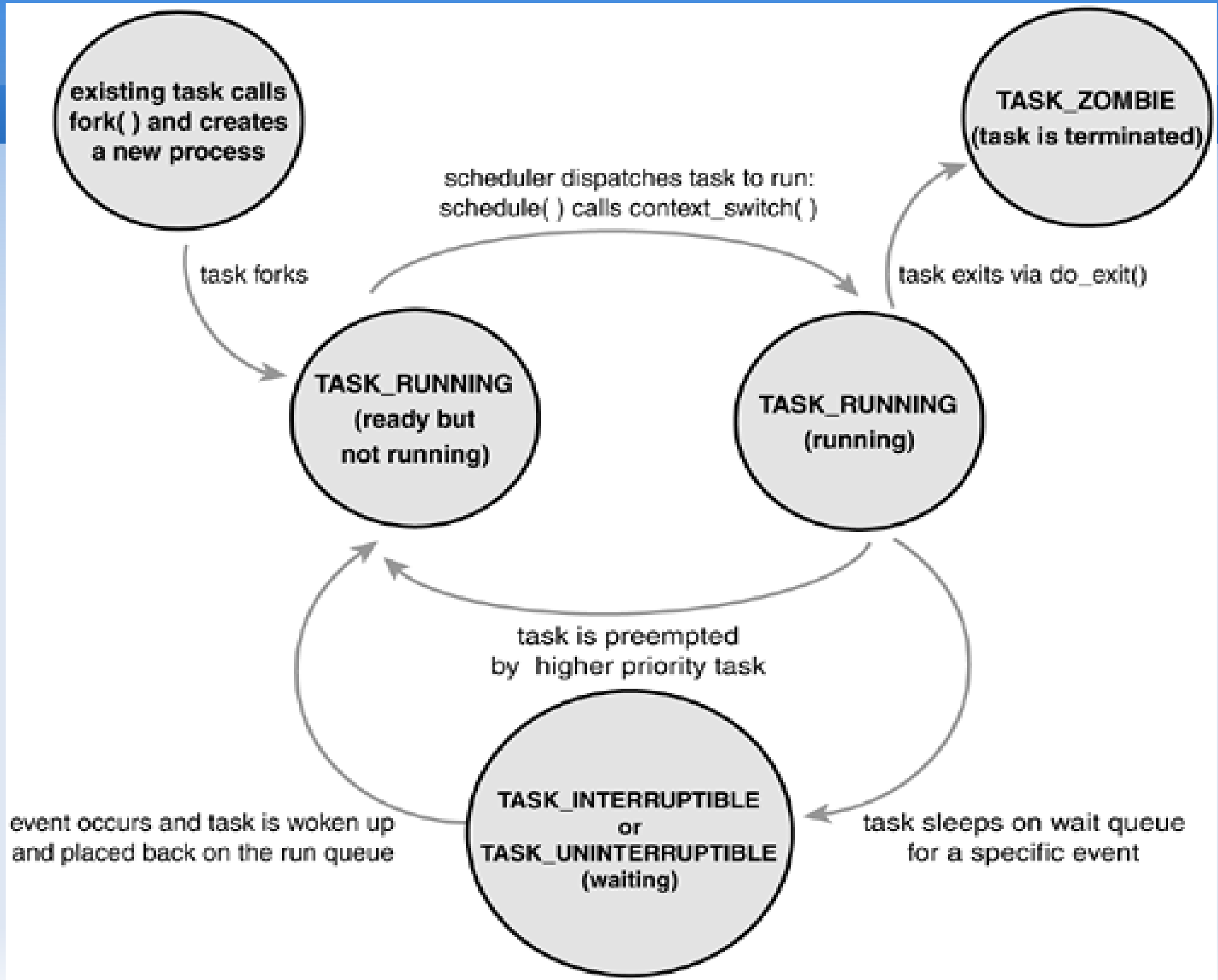

task_struct
state
thread_info
usage
flags
...
run_list
tasks
...
mm
real_parent
parent
...
tty
...
thread
...
fs
files
signal
pending
...
thread_info
Low-level information for the process
mm_struct
Pointers to memory area descriptors
tty_struct
tty associated with the process
fs_struct
Current directory
files_struct
Pointers to file descriptors
signal_struct
Signals received

3.2.1. Process State

- **TASK_RUNNING:** The process is either executing on a CPU or waiting to be executed.
- **TASK_INTERRUPTIBLE:** The process is suspended (sleeping) until some condition becomes true. Raising a hardware interrupt, releasing a system resource the process is waiting for, or delivering a signal are examples of conditions that might wake up the process (put its state back to TASK_RUNNING).
- **TASK_UNINTERRUPTIBLE:** Like TASK_INTERRUPTIBLE, except that delivering a signal to the sleeping process leaves its state unchanged. This process state is seldom used.
- **TASK_STOPPED:** Process execution has been stopped; the process enters this state after receiving a SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU signal.
- **TASK_TRACED:** Process execution has been stopped by a debugger. When a process is being monitored by another (such as when a debugger executes a ptrace( ) system call to monitor a test program), each signal may put the process in the TASK_TRACED state.

More process states

- Two additional states of the process can be stored both in the **state field and in the exit_state field** of the process descriptor;
- as the field name suggests, a process reaches one of these two states only when its execution is terminated:
- **EXIT_ZOMBIE:** Process execution is terminated, but the parent process has not yet issued a wait4( ) or waitpid( ) system call to return information about the dead process.
  - Before the wait( )-like call is issued, the kernel cannot discard the data contained in the dead process descriptor because the parent might need it.
- **EXIT_DEAD:** The final state: the process is being removed by the system because the parent process has just issued a wait4( ) or waitpid( ) system call for it.
  - Changing its state from EXIT_ZOMBIE to EXIT_DEAD avoids race conditions due to other threads of execution that execute wait( )-like calls on the same process.


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

3.2.2. Identifying a Process

- As a general rule, **each execution context that can be independently scheduled must have its own process descriptor**;
  - therefore, **even lightweight processes**, which share a large portion of their kernel data structures, have their own task_struct structures.
- The 32-bit addresses of the task_struct structure are referred to as **process descriptor pointers**. Most of the references to processes that the kernel makes are through process descriptor pointers.
- On the other hand, Unix-like operating systems allow users to identify processes by means of a number called the **Process ID (or PID)**, which is stored in the **pid field** of the process descriptor.
  - PIDs are **numbered sequentially**: the PID of a newly created process is normally the PID of the previously created process increased by one.

Thread Groups, tgid and pid

- Linux associates a **different PID with each process** or lightweight process in the system. This approach allows the maximum flexibility, because **every execution context in the system can be uniquely identified.**

- On the other hand, **Unix programmers expect threads in the same group to have a common PID**. **POSIX 1003.1c standard** states that all threads of a multithreaded application must have the same PID.

  - For instance, it should be possible to a **send a signal** specifying a PID that affects all threads in the group.

- Linux makes use of **thread groups**. The identifier shared by the threads is the **PID of the thread group leader** , that is, the PID of the **first lightweight process** in the group; it is stored in the **tgid field** of the process descriptors.

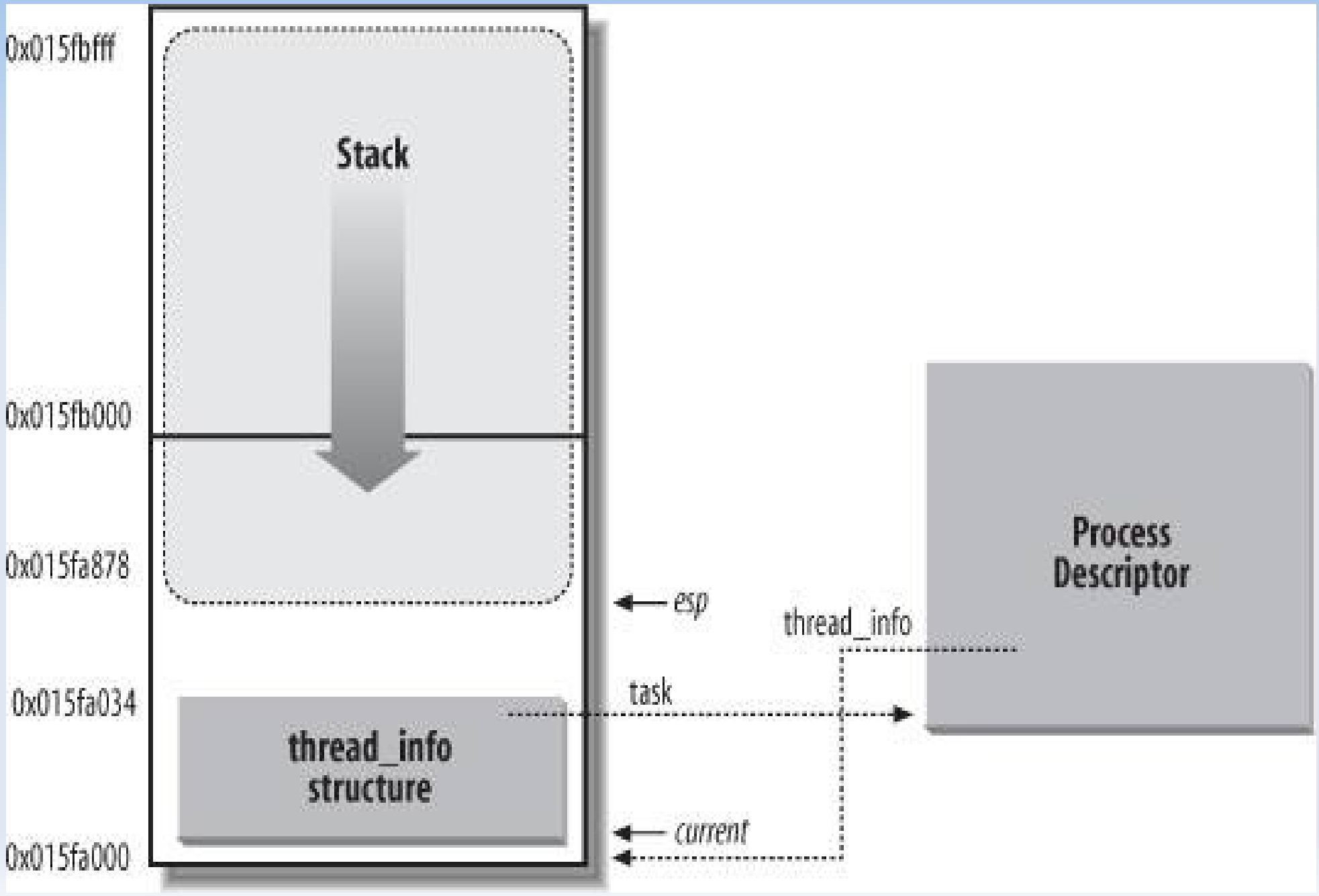
- The **getpid( ) system call** returns the value of tgid relative to the current process instead of the value of pid, so all the threads of a multithreaded application share the same identifier.

  - Most processes belong to a **thread group consisting of a single member**; as thread group leaders, they have the **tgid field equal to the pid** field, thus the getpid( ) system call works as usual for this kind of process.

3.2.2.1. Process descriptors handling

- Process descriptors are stored in **dynamic memory** rather than in the memory area permanently assigned to the kernel.
- For each process, Linux packs **two different data structures** in a single per-process memory area:
  - a small data structure linked to the process descriptor, namely the **thread_info structure**,
  - and the **Kernel Mode process stack**.
- The length of this memory area is usually 8,192 bytes (**two page frames**).
  - For reasons of efficiency (avoiding **fragmentation**) the kernel can be configured at **compilation time** so that the memory area including stack and thread_info structure spans a **single page frame** (4,096 bytes).

thread_info and kernel stack



thread_info and kernel stack

- The **esp register** is the CPU stack pointer, which is used to address the **stack's top location**.
- On **80x86 systems**, the stack starts at the end and **grows toward the beginning** of the memory area.
- Right after **switching from User Mode to Kernel Mode**, the kernel stack of a process is always **empty**, and therefore the esp register points to the byte immediately following the stack.
- The value of the esp is decreased as soon as data is written into the stack. Because the thread_info structure is 52 bytes long, the kernel stack can expand up to 8,140 bytes.
- The C language allows the thread_info structure and the kernel stack of a process to be conveniently represented by means of the following **union construct**:

```
union thread_union {
    struct thread_info thread_info;
    unsigned long stack[2048]; /* 1024 for 4KB stacks */
};
```

3.2.2.2. Identifying the current process

- **benefit in terms of efficiency**: the kernel can easily obtain the address of the thread_info structure of the process currently running on a CPU from the value of the esp register.

- In fact, if the thread_union structure is 8 KB (2^13 bytes) long, the kernel **masks out the 13 least significant bits of esp** to obtain the base address of the thread_info structure;

  - This is done by the **current_thread_info( )** function.

- Most often the kernel needs the address of the process descriptor rather than the address of the thread_info structure.

- To get the process descriptor pointer of the process currently running on a CPU, the kernel makes use of the **current** macro, which is essentially equivalent to **current_thread_info( )->task :**

```
movl $0xffffe000,%ecx /* or 0xfffff000 for 4KB stacks */

andl %esp,%ecx

movl (%ecx),p
```

- For example, current->pid returns the process ID of the process currently running on the CPU.

3.2.2.3. Doubly linked lists

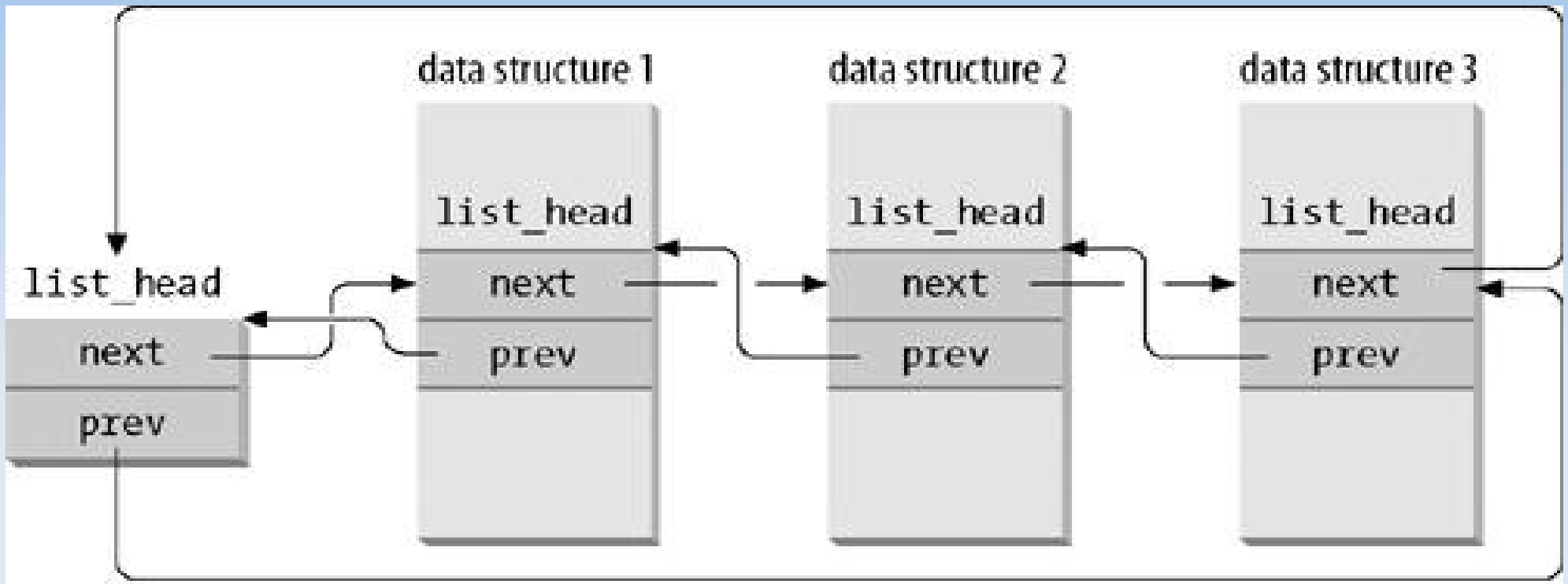
- The Linux kernel defines the **list_head data structure**, whose only fields next and prev represent the forward and back pointers of a generic doubly linked list element, respectively.

- It is important to note, however, that the pointers in a list_head field store the addresses of other list_head fields rather than the addresses of the whole data structures in which the list_head structure is included; see Figure 3-3 (a).

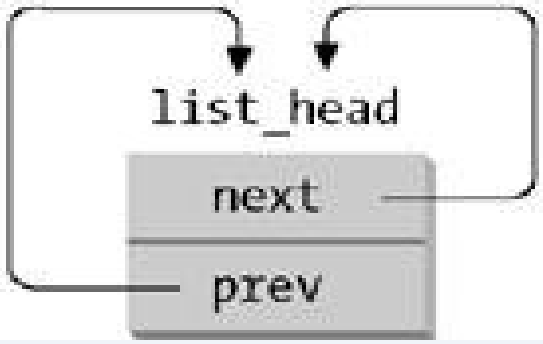
- A new list is created by using the **LIST_HEAD**(list_name) macro.

  - It declares a new variable named **list_name of type list_head**, which is a **dummy first element** that acts as a placeholder for the head of the new list, and initializes the prev and next fields of the list_head data structure so as to point to the list_name variable itself.

Doubly linked lists



(a) a doubly linked listed with three elements



(b) an empty doubly linked list

Several functions and macros implement the primitives

- **list_add(n,p)** Inserts an element pointed to by n right after the specified element pointed to by p. (To insert n at the beginning of the list, set p to the address of the list head.)
- **list_add_tail(n,p)** Inserts an element pointed to by n right before the specified element pointed to by p. (To insert n at the end of the list, set p to the address of the list head.)
- **list_del(p)** Deletes an element pointed to by p. (There is no need to specify the head of the list.)
- **list_empty(p)** Checks if the list specified by the address p of its head is empty.
- **list_entry(p,t,m)** Returns the address of the data structure of type t in which the list_head field that has the name m and the address p is included.
- **list_for_each(p,h)** Scans the elements of the list specified by the address h of the head; in each iteration, a pointer to the list_head structure of the list element is returned in p.
- **list_for_each_entry(p,h,m)** Similar to list_for_each, but returns the address of the data structure embedding the list_head structure rather than the address of the list_head structure itself.

Other Linked lists data type

- The Linux kernel 2.6 sports **another kind of doubly linked list**, which mainly differs from a list_head list because it is **not circular**;
- it is mainly **used for hash tables**, where space is important, and finding the the last element in constant time is not.
- The list head is stored in an **hlist_head** data structure, which is simply a pointer to the first element in the list (NULL if the list is empty).
- Each element is represented by an **hlist_node** data structure, which includes a pointer **next** to the next element, and a pointer **pprev** to the next field of the previous element.
- Because the list is not circular, **the pprev field of the first element and the next field of the last element are set to NULL**.
- The list can be handled by means of **several helper functions and macros** similar to those listed in Table 3-1: hlist_add_head( ), hlist_del( ), hlist_empty( ), hlist_entry, hlist_for_each_entry, and so on.

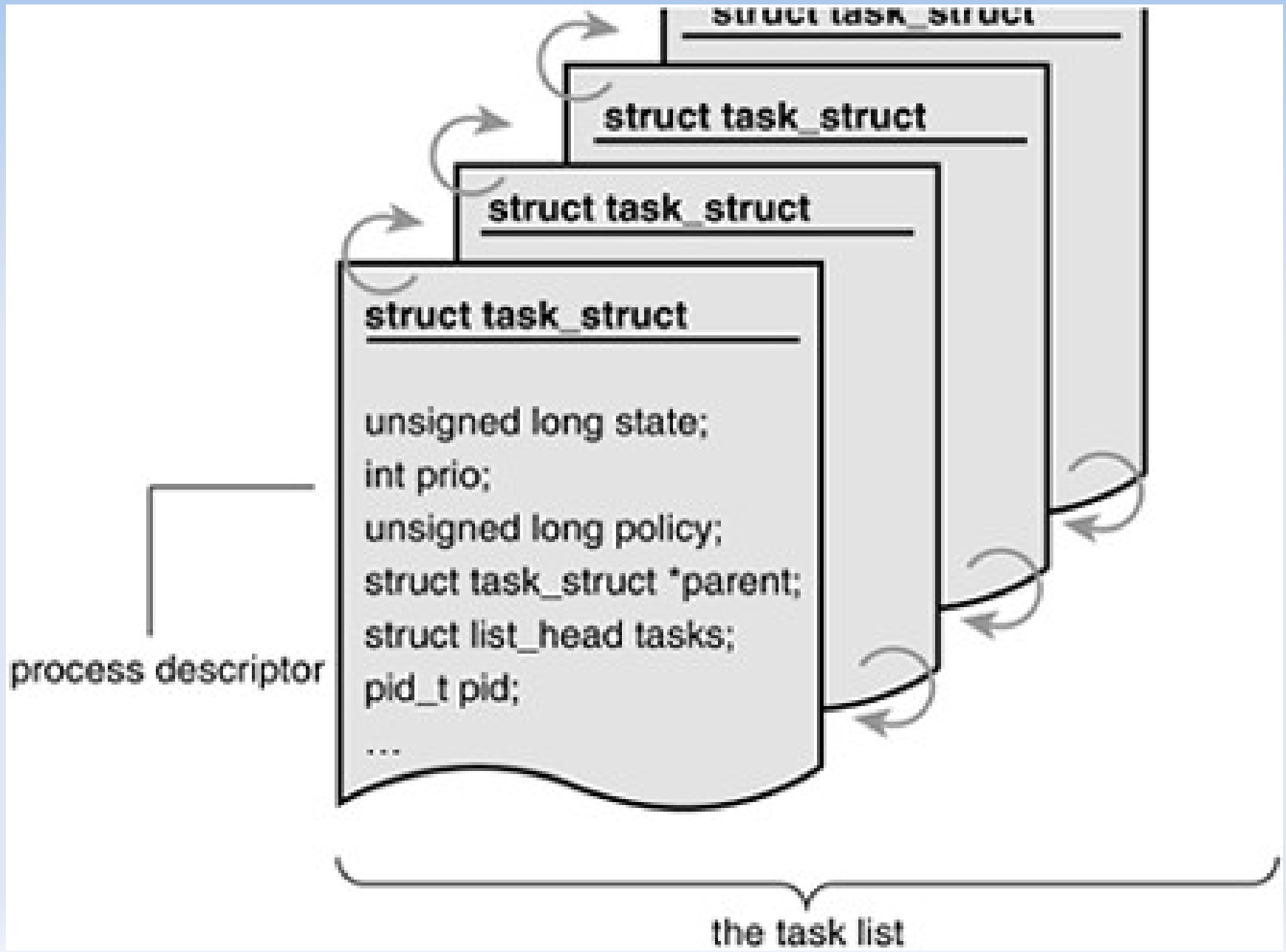
3.2.2.4. The process list (or task list)

- The first example of a **doubly linked list** we will examine is the **process list**, a list that links together **all existing process descriptors**.

- Each task_struct structure includes a **tasks** field of type list_head whose prev and next fields point, respectively, to the previous and to the next task_struct element.

- The head of the process list is the **init_task** task_struct descriptor; it is the process descriptor of the so-called **process 0 or swapper**.

The process descriptor and task list.



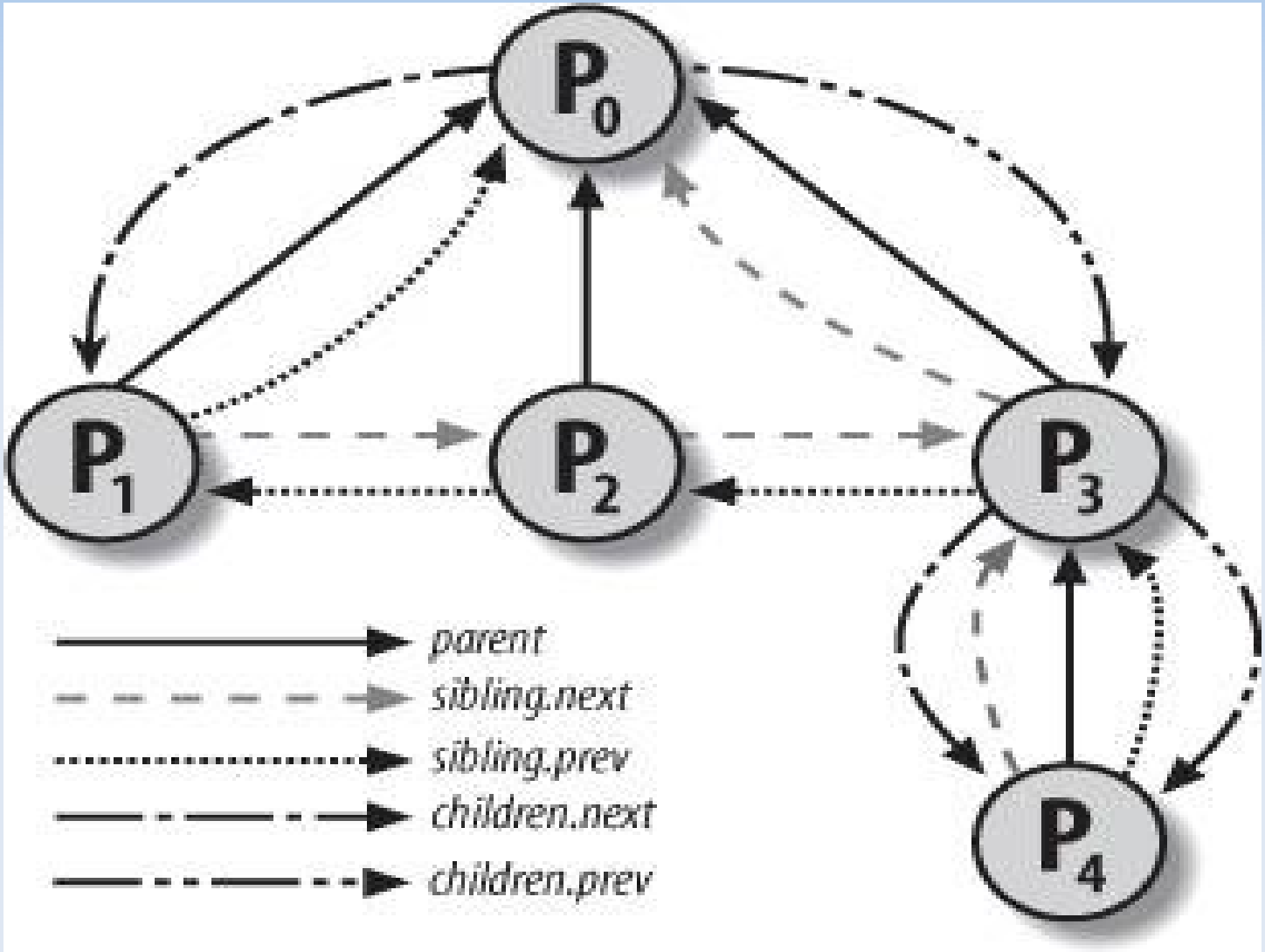
3.2.2.5. The lists of TASK_RUNNING processes

- When looking for a new process to run on a CPU, the kernel has to consider only the runnable processes.
- Earlier Linux versions put all runnable processes in the same list called **runqueue**. The earlier schedulers were compelled to **scan the whole list** in order to select the "best" runnable process, an O(n) operation.
- Linux 2.6 implements the runqueue differently. The aim is to allow the scheduler to select the best runnable process in **constant time** O(1), independently of the number of runnable processes.
- The trick consists of **splitting the runqueue** in many lists of runnable processes, **one list per process priority**. Each task_struct descriptor includes a **run_list** field of type list_head. If the process **priority is equal to k** (a value ranging between 0 and 139), **the run_list field links the process descriptor into the list of runnable processes having priority k**.
- Furthermore, on a **multiprocessor system**, **each CPU has its own runqueue**, that is, its own set of lists of processes.
  - This is a classic example of making a data structures more complex to improve performance: to make scheduler operations more efficient, the runqueue list has been split into 140 different lists!

3.2.3. Relationships Among Processes

- Several fields must be introduced in a process descriptor to represent relationships.
- **Processes 0 and 1 are created by the kernel**; as we'll see later, **process 1 (init) is the ancestor of all other processes**.
- Table 3-3. Fields of a process descriptor used to express parenthood relationships
  - **real_parent** : Points to the process descriptor of the process that created P or to the descriptor of process 1 (init) if the parent process no longer exists.
  - **parent** : Points to the current parent of P (this is the process that must be signaled when the child process terminates); its value usually coincides with that of real_parent.
    - It may occasionally differ, such as when another process issues a ptrace( ) system call requesting that it be allowed to monitor P.
  - **children**: The head of the list containing all children created by P.
  - **sibling:** The pointers to the next and previous elements in the list of the sibling processes, those that have the same parent as P.

Process relationships



- Figure 3-4 illustrates the parent and sibling relationships of a group of processes. Process P0 successively created P1, P2, and P3. Process P3, in turn, created process P4.

Furthermore, there exist other relationships among processes:

- a process can be a **leader of a process group** or of a **login session**,
- it can be a **leader of a thread group**,
- **fields of the process descriptor that establish non-parenthood relationships:**
  - **group_leader:** Process descriptor pointer of the group leader of P
  - **signal->pgrp:** PID of the group leader of P
  - **tgid:** PID of the thread group leader of P
  - **signal->session:** PID of the login session leader of P

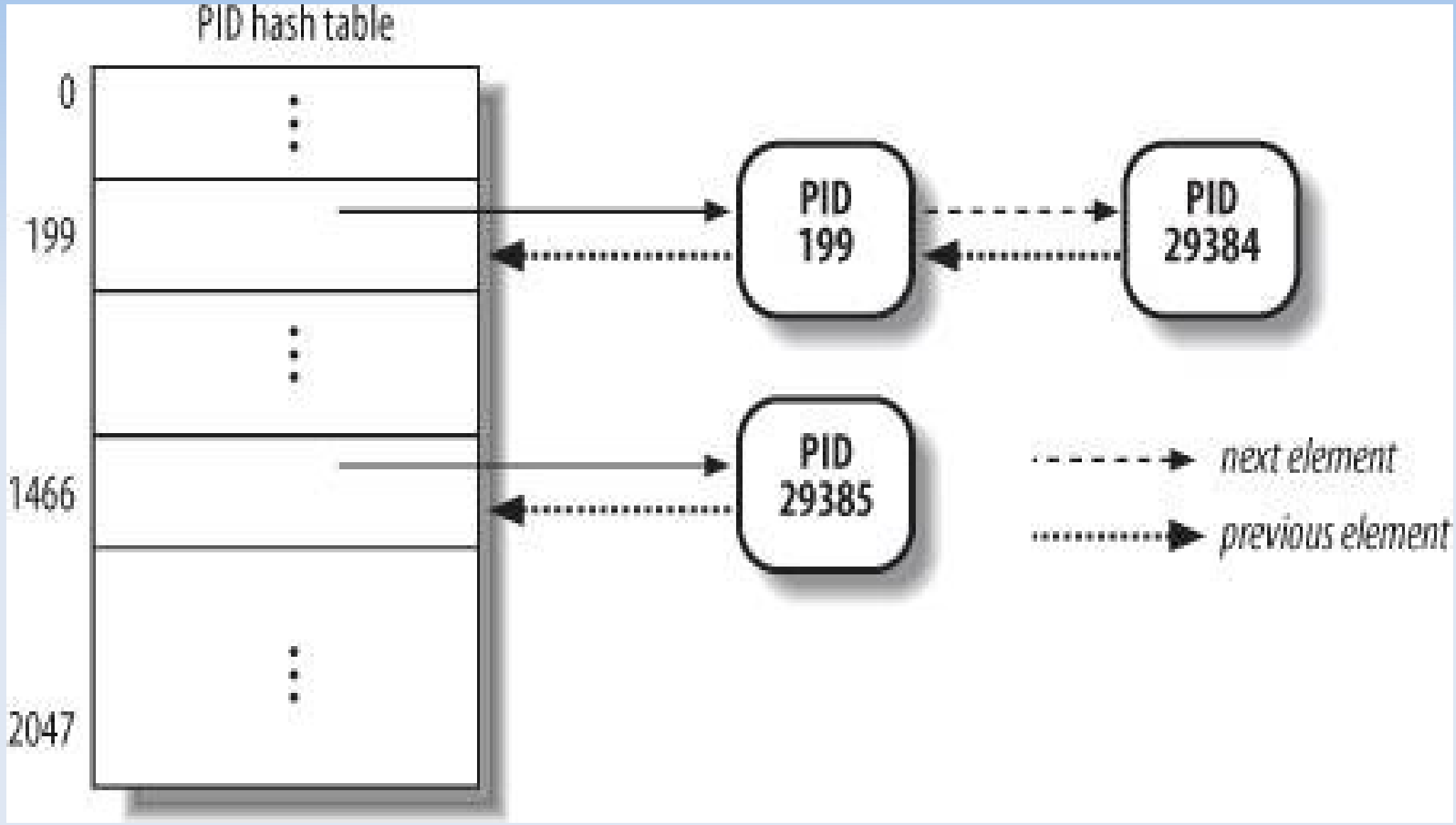
3.2.3.1. The pidhash table and chained lists

- In several circumstances, the kernel must be able to **derive the process descriptor pointer corresponding to a PID**.
- **Scanning the process list** sequentially and checking the pid fields of the process descriptors is feasible but **rather inefficient**.
- To speed up the search, **four hash tables** have been introduced. **Process descriptor includes fields that represent different types of PID** (see Table), and each type of PID requires its own hash table.
- Table 3-5. The four hash tables and corresponding fields in the process descriptor

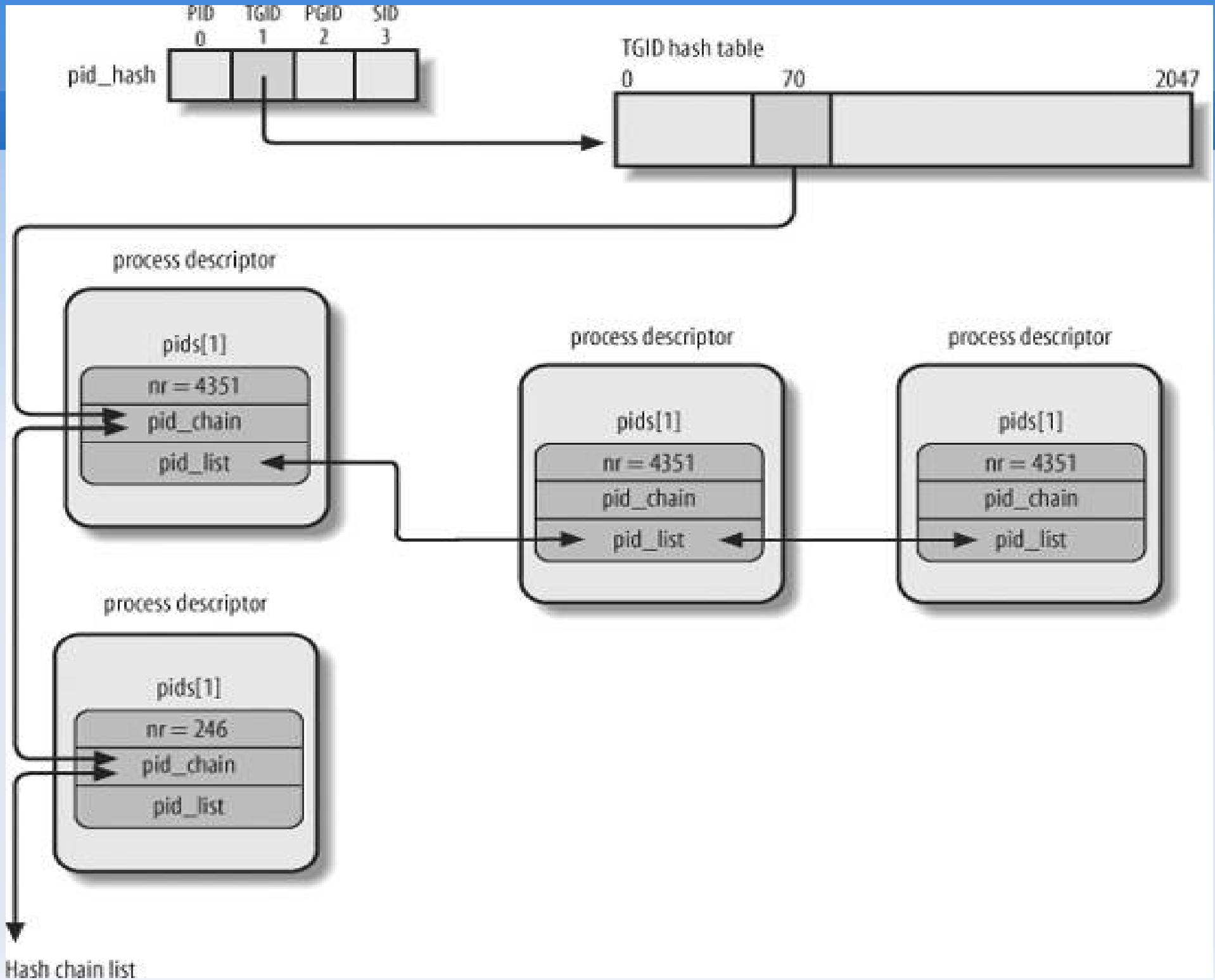
Hash table type	Field name	Description
PIDTYPE_PID	pid	PID of the process
PIDTYPE_TGID	tgid	PID of thread group leader process
PIDTYPE_PGID	pgrp	PID of the group leader process
PIDTYPE_SID	session	PID of the session leader process

Hashing with chaining

- a hash function does not always ensure a one-to-one correspondence between PIDs and table indexes. Two different PIDs that hash into the same table index are said to be colliding.
- **Linux uses chaining to handle colliding PIDs**; each table entry is the head of a doubly linked list of colliding process descriptors.
- **Hashing with chaining is preferable to a linear transformation.** It would be a waste of storage to define a table consisting of 32,768 entries, if, at any given instance, most such entries are unused.
- The data structures used in the PID hash tables are quite sophisticated, because they must keep track of the relationships between the processes.



a PID hash table with two lists. The processes having PIDs 2,890 and 29,384 hash into the 200th element of the table, while the process having PID 29,385 hashes into the 1,466th element of the table.


PID
TGID
PGID
SID
0
1
2
3
pid_hash
TGID hash table
0
70
2047
process descriptor
pids[1]
nr = 4351
pid_chain
pid_list
process descriptor
pids[1]
nr = 4351
pid_chain
pid_list
process descriptor
pids[1]
nr = 4351
pid_chain
pid_list
process descriptor
pids[1]
nr = 246
pid_chain
pid_list
Hash chain list

3.2.4. How Processes Are Organized

- The **runqueue** lists group all processes in a **TASK_RUNNING** state.
- When it comes to **grouping processes in other states**, the various states call for different types of treatment, with Linux opting for one of the choices shown in the following list.
  - Processes in a **TASK_STOPPED, EXIT_ZOMBIE, or EXIT_DEAD** state are **not linked in specific lists**.
    - There is no need to group processes in any of these three states, because stopped, zombie, and dead processes are **accessed only via PID or via linked lists of the child processes** for a particular parent.
  - Processes in a **TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE** state are **subdivided into many classes**, each of which corresponds to a **specific event**.
    - In this case, the process state does not provide enough information to retrieve the process quickly, so it is necessary to introduce additional lists of processes. These are called **wait queues** and are discussed next.

3.2.4.1. Wait queues

- Wait queues **implement conditional waits on events**: a process wishing to wait for a specific event places itself in the proper wait queue and relinquishes control.
- Therefore, **a wait queue represents a set of sleeping processes**, which are woken up by the kernel when some condition becomes true.
- Wait queues are **implemented as doubly linked lists** whose elements include pointers to process descriptors.

```
struct __wait_queue_head{
    spinlock_t lock;
    struct list_head task_list;
};
typedef struct __wait_queue_head wait_queue_head_t;
```

- Because wait queues are **modified by interrupt handlers** as well as by major kernel functions, the doubly linked lists must be **protected from concurrent accesses**.

Wait queues

Elements of a wait queue list are of type wait_queue_t:

```
struct __wait_queue {
    unsigned int flags;
    struct task_struct * task;
    wait_queue_func_t func;
    struct list_head task_list;
};
typedef struct __wait_queue wait_queue_t;
```

- **Each element in the wait queue list represents a sleeping process**, which is **waiting for some event to occur**; its descriptor address is stored in the task field.
- However, it is not always convenient to wake up all sleeping processes in a wait queue:
- "**thundering herd**" with which multiple processes are wakened only to race for a resource that can be accessed by one of them, with the result that remaining processes must once more be put back to sleep.

Exclusive and nonexclusive sleeping processes

- Thus, there are **two kinds of sleeping processes**:
  - **exclusive processes** (denoted by the value 1 in the flags field of the corresponding wait queue element) are **selectively woken up by the kernel**,
  - while **nonexclusive processes** (denoted by the value 0 in the flags field) are always woken up by the kernel when the event occurs.
- A process waiting for a **resource that can be granted to just one process** at a time is a typical exclusive process. Processes waiting for an event that may concern any of them are nonexclusive.

Symple sleep: wait_event()

- The **wait_event** and **wait_event_interruptible** macros put the calling process to sleep on a wait queue until a given condition is verified.
- For instance, the **wait_event(wq,condition)** macro essentially yields the following fragment:

```
DEFINE_WAIT(__wait);
for (;;) {
    prepare_to_wait(&wq, &__wait, TASK_UNINTERRUPTIBLE);
    if (condition)
        break;
    schedule( );
}
finish_wait(&wq, &__wait);
```

Wake up !

The kernel awakens processes in the wait queues, putting them in the TASK_RUNNING state, by means of one of the wake_up family macros
For instance, the wake_up macro is essentially equivalent to the following code fragment:

```
void wake_up(wait_queue_head_t *q)
{
    struct list_head *tmp;
    wait_queue_t *curr;

    list_for_each(tmp, &q->task_list) {
        curr = list_entry(tmp, wait_queue_t, task_list);
        if (curr->func(curr, TASK_INTERRUPTIBLE|TASK_UNINTERRUPTIBLE,
                    0, NULL) && curr->flags)
            break;
    }
}
```

3.2.5. Process Resource Limits

- Each process has an associated **set of resource limits** , which specify the amount of system resources it can use.
- These limits **keep a user from overwhelming the system** (its CPU, disk space, and so on).
- The resource limits for the current process are stored in the current->signal->rlim field, that is, in a field of the process's signal descriptor. The field is an **array of elements of type struct rlimit**, one for each resource limit:

```
struct rlimit {
    unsigned long rlim_cur;
    unsigned long rlim_max;
};
```

Table 3-7. Resource limits

- **RLIMIT_AS** The maximum size of process **address space**, in bytes. The kernel checks this value when the process uses malloc( ) or a related function to enlarge its address space.
- **RLIMIT_CORE** The maximum core dump file size, in bytes. The kernel checks this value when a process is aborted, before creating a core file in the current directory of the process (see the section "Actions Performed upon Delivering a Signal" in Chapter 11). If the limit is 0, the kernel won't create the file.
- **RLIMIT_CPU** The maximum **CPU time** for the process, in seconds. If the process exceeds the limit, the kernel sends it a SIGXCPU signal, and then, if the process doesn't terminate, a SIGKILL signal (see Chapter 11).
- **RLIMIT_DATA** The maximum **heap size**, in bytes. The kernel checks this value before expanding the heap of the process (see the section "Managing the Heap" in Chapter 9).
- **RLIMIT_FSIZE** The maximum **file size** allowed, in bytes. If the process tries to enlarge a file to a size greater than this value, the kernel sends it a SIGXFSZ signal.
- **RLIMIT_LOCKS** Maximum number of file locks (currently, not enforced).

Table 3-7. Resource limits

- **RLIMIT_MEMLOCK** The maximum size of nonswappable memory, in bytes. The kernel checks this value when the process tries to lock a page frame in memory using the mlock( ) or mlockall( ) system calls (see the section "Allocating a Linear Address Interval" in Chapter 9).
- **RLIMIT_MSGQUEUE** Maximum number of bytes in POSIX message queues (see the section "POSIX Message Queues" in Chapter 19).
- **RLIMIT_NOFILE** The maximum number of **open file descriptors** . The kernel checks this value when opening a new file or duplicating a file descriptor (see Chapter 12).
- **RLIMIT_NPROC** The maximum **number of processes** that the user can own (see the section "The clone( ), fork( ), and vfork( ) System Calls" later in this chapter).
- **RLIMIT_RSS** The maximum number of **page frames** owned by the process (currently, not enforced).
- **RLIMIT_SIGPENDING** The maximum number of pending signals for the process (see Chapter 11).
- **RLIMIT_STACK** The maximum stack size, in bytes. The kernel checks this value before expanding the User Mode stack of the process (see the section "Page Fault Exception Handler" in Chapter 9).

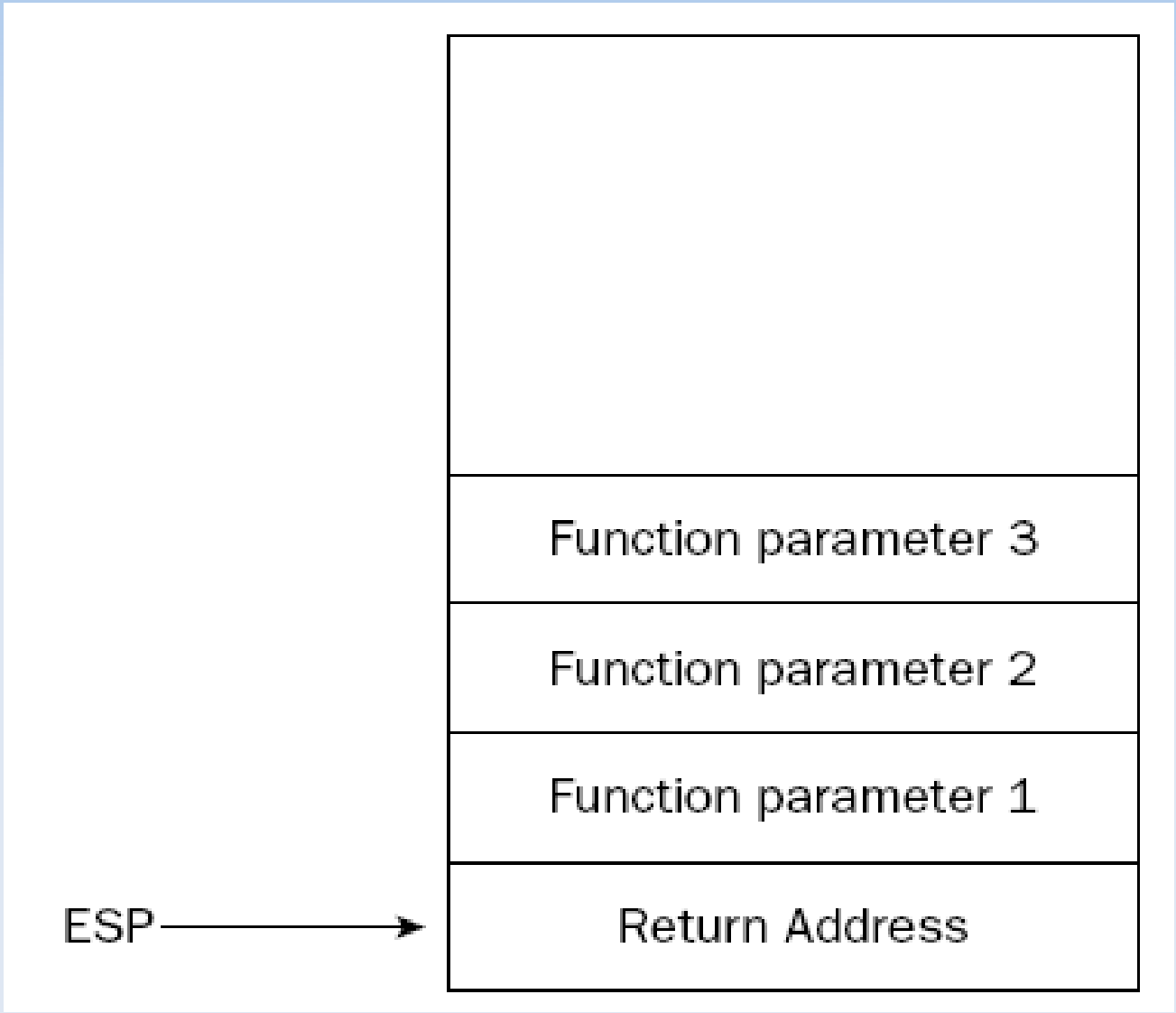
3.3. Process Switch

- To control the execution of processes, the kernel must be able to suspend the execution of the process running on the CPU and resume the execution of some other process previously suspended.
- This activity goes variously by the names **process switch, task switch, or context switch**.
- The next sections describe the elements of process switching in Linux.

3.3.1. Hardware Context

- While each process can have its own address space, **all processes have to share the CPU registers**.
- So before resuming the execution of a process, the kernel must ensure that each such register is loaded with **the value it had** when the process was suspended.
- **The set of data that must be loaded into the registers before the process resumes its execution on the CPU is called the hardware context** .
- The hardware context is a **subset of the process execution context**, which includes all information needed for the process execution.
- The 80x86 architecture includes a specific segment type called the Task State Segment (TSS), to **store hardware contexts**.
- In Linux,
  - a part of the hardware context of a process is stored in the **process descriptor**,
  - while the remaining part is saved in the **Kernel Mode stack**.

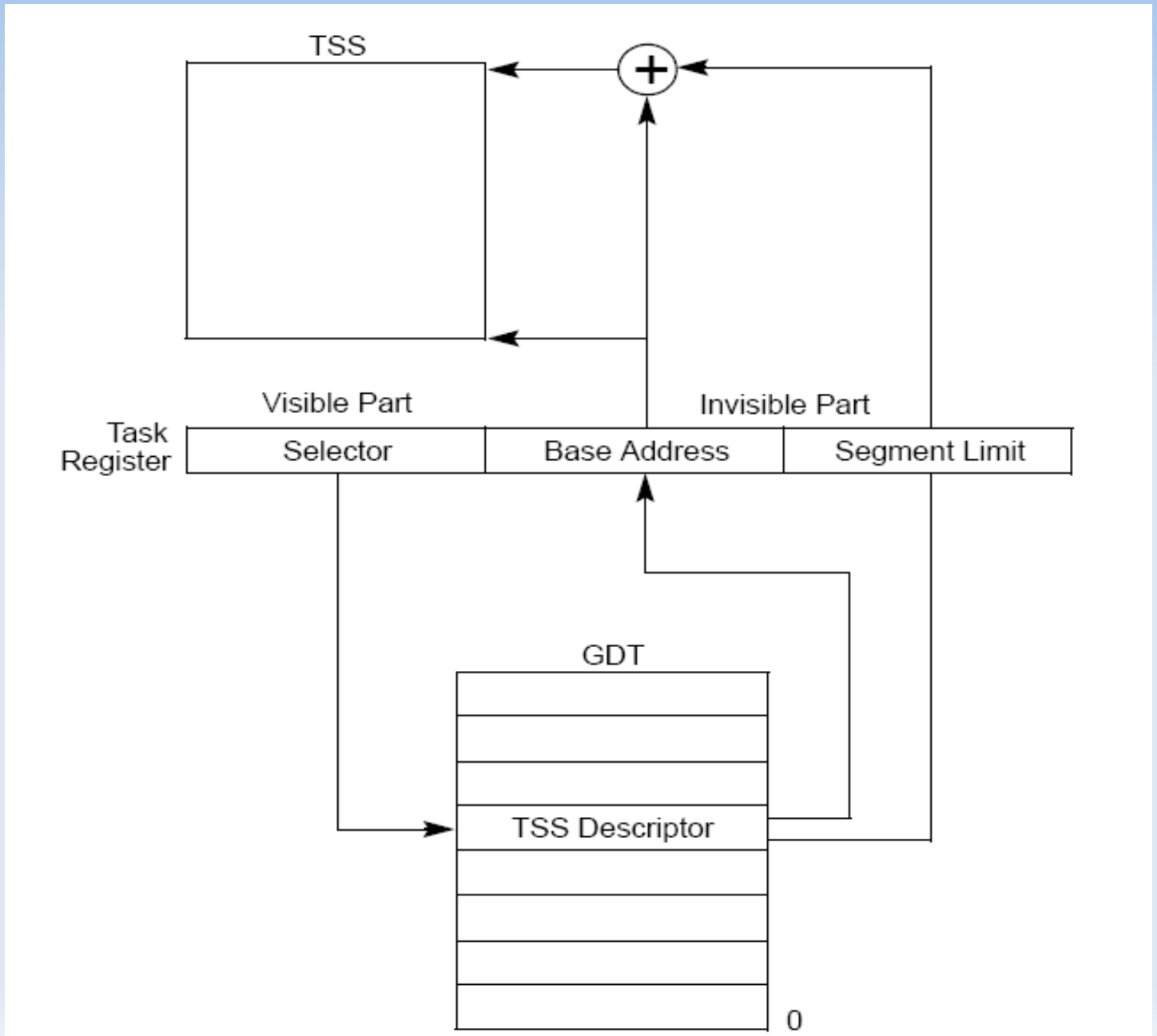
Stack



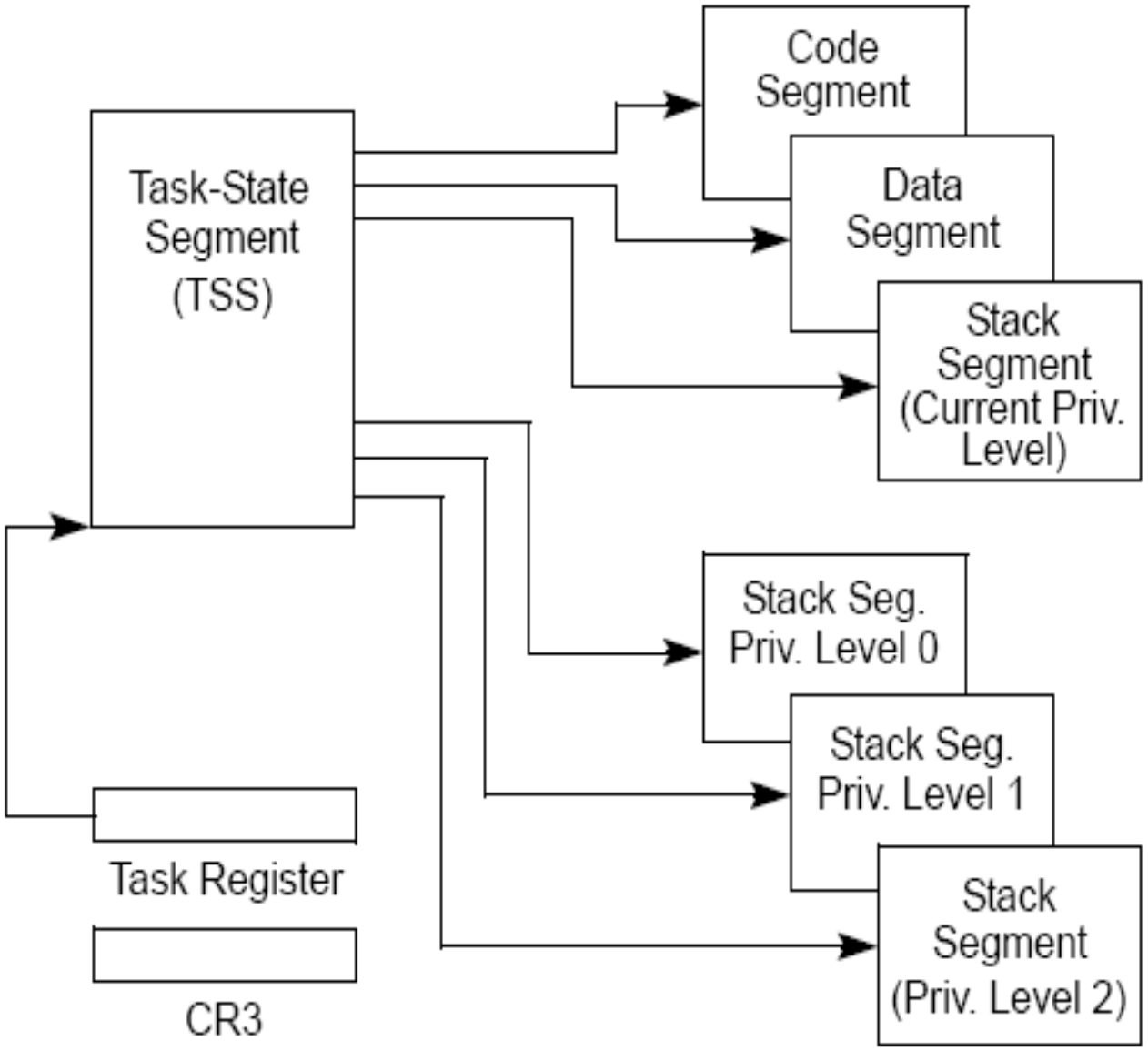
High address

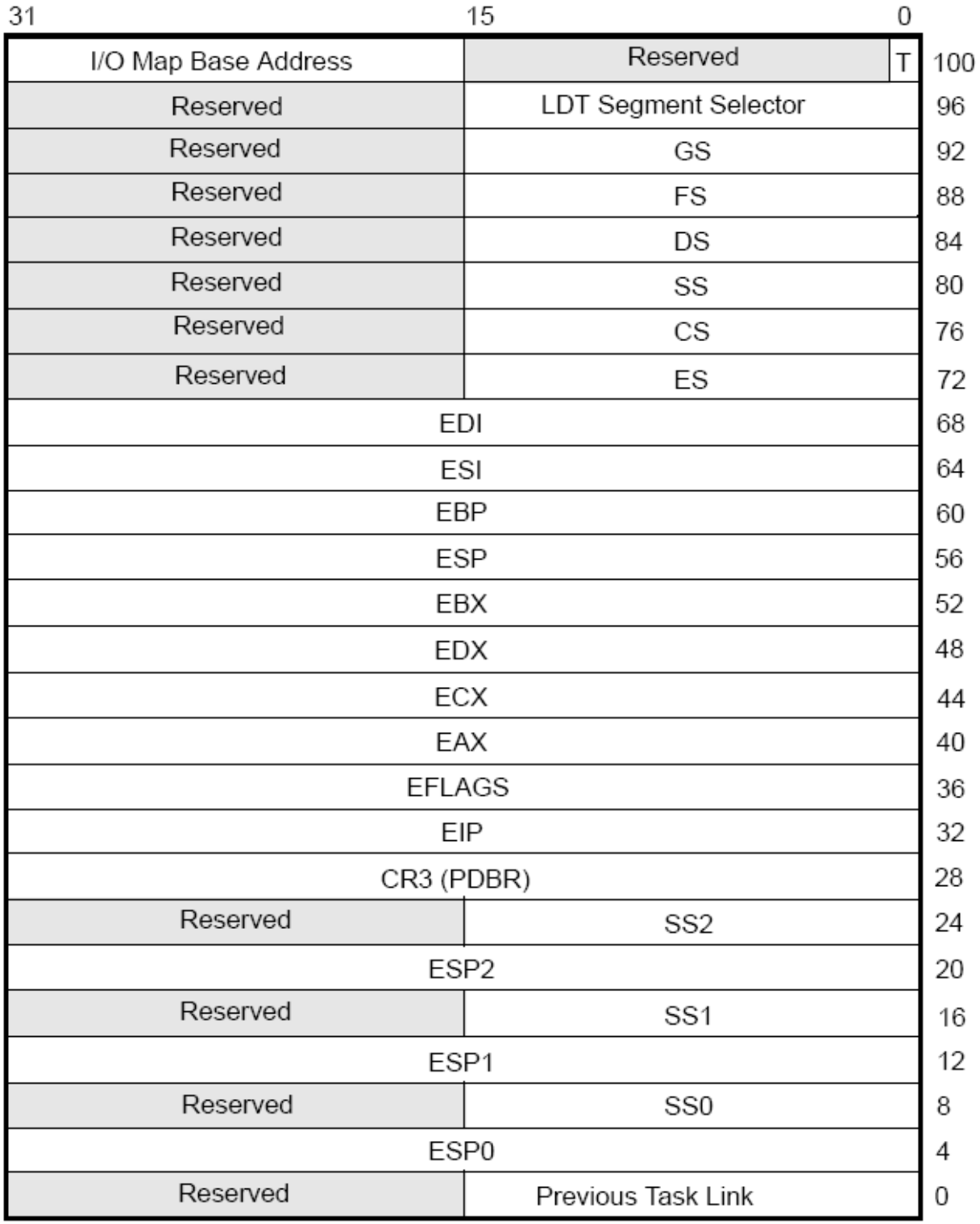
Low address

Task Register (tr)



Structure of a Task




31
15
0
I/O Map Base Address
Reserved
T
100
Reserved
LDT Segment Selector
96
Reserved
GS
92
Reserved
FS
88
Reserved
DS
84
Reserved
SS
80
Reserved
CS
76
Reserved
ES
72
EDI
68
ESI
64
EBP
60
ESP
56
EBX
52
EDX
48
ECX
44
EAX
40
EFLAGS
36
EIP
32
CR3 (PDBR)
28
Reserved
SS2
24
ESP2
20
Reserved
SS1
16
ESP1
12
Reserved
SS0
8
ESP0
4
Reserved
Previous Task Link
0

hardware/software context switch

- Old versions of Linux took advantage of the hardware support offered by the 80x86 architecture and performed a process switch through a **far jmp instruction** to the **selector of the Task State Segment Descriptor of the next process**.
- But **Linux 2.6 uses software to perform a process switch** for the following reasons:
  - Step-by-step switching performed through a sequence of mov instructions allows **better control over the validity of the data being loaded**.
  - The **amount of time** required by the old approach and the new approach **is about the same**.
- **Process switching occurs only in Kernel Mode**.
  - The contents of all registers used by a process in User Mode have already been saved on the Kernel Mode stack before performing process switching.
  - This includes the contents of the ss and esp pair that specifies the User Mode stack pointer address.

3.3.2. Task State Segment

- Although Linux doesn't use hardware context switches, it is nonetheless **forced to set up a TSS for each distinct CPU in the system**. This is done for **two main reasons**:
  - When an 80x86 CPU **switches from User Mode to Kernel Mode**, it fetches the address of the **Kernel Mode stack from the TSS.**
  - When a **User Mode process attempts to access an I/O port** by means of an in or out instruction, the **CPU may need to access an I/O Permission Bitmap stored in the TSS** to verify whether the process is allowed to address the port.
- The **tss_struct** structure describes the format of the TSS.
- the **init_tss** array stores one TSS for each CPU on the system.
- **At each process switch, the kernel updates some fields of the TSS** so that the corresponding CPU's control unit may safely retrieve the information it needs.
- Thus, the TSS reflects the privilege of the current process on the CPU, but **there is no need to maintain TSSs for processes when they're not running**.

3.3.2.1. The thread field

- At every process switch, the hardware context of the process being replaced must be saved somewhere.
- It **cannot be saved on the TSS**, as in the original Intel design, because Linux uses a single TSS for each processor, instead of one for every process.
- Thus, each process descriptor includes a **field called thread of type thread_struct**, in which **the kernel saves the hardware context** whenever the process is being switched out.
- As we'll see later, **this data structure includes fields for most of the CPU registers**, **except the general-purpose registers** such as eax, ebx, etc., which are **stored in the Kernel Mode stack.**

3.3.3. Performing the Process Switch

- **A process switch may occur at just one well-defined point: the schedule( ) function**, which is discussed at length in Chapter 7. Here, we are only concerned with how the kernel performs a process switch.

- Essentially, **every process switch consists of two steps**:

  - Switching the Page Global Directory to **install a new address space**; we'll describe this step in Chapter 9.

  - **Switching the Kernel Mode stack and the hardware context**, which provides all the information needed by the kernel to execute the new process, including the CPU registers. This is performed by the **switch_to macro**. It is one of the **most hardware-dependent routines** of the kernel.

steps performed by switch_to()

The switch_to macro is coded in extended inline assembly language, but we'll describe what the switch_to macro typically does on an 80x86 microprocessor by using standard assembly language:

1. Saves the values of prev and next in the eax and edx registers, respectively:

```
movl prev, %eax
movl next, %edx
```

2. Saves the contents of the **eflags and ebp registers in the prev Kernel Mode stack**. They must be saved because the compiler assumes that they will stay unchanged until the end of switch_to:

```
pushfl
pushl %ebp
```

3. Saves the content of **esp in prev->thread.esp** so that the field points to the top of the prev Kernel Mode stack:

```
movl %esp,484(%eax)
```

The 484(%eax) operand identifies the memory cell whose address is the contents of eax plus 484.

steps performed by switch_to()

4. Loads next->thread.esp in esp. From now on, the kernel operates on the Kernel Mode stack of next, so **this instruction performs the actual process switch** from prev to next. Because the address of a process descriptor is closely related to that of the Kernel Mode stack, **changing the kernel stack means changing the current process**:

```
movl 484(%edx), %esp
```

.....

7. Jumps to the _ _switch_to( ) C function (see next):

```
jmp _ _switch_to
```

The steps performed by __switch_to():

1. Executes the code yielded by the **__unlazy_fpu( )** macro to optionally save the contents of the FPU, MMX, and XMM registers of the prev_p process.

```
_ _unlazy_fpu(prev_p);
```

.....

3. Loads next_p->thread.esp0 in the esp0 field of the TSS relative to the local CPU:

```
init_tss[cpu].esp0 = next_p->thread.esp0;
```

.....

8. **Updates the I/O bitmap in the TSS, if necessary**. This must be done when either next_p or prev_p has its own customized I/O Permission Bitmap:

```
if (prev_p->thread.io_bitmap_ptr || next_p->thread.io_bitmap_ptr)
    handle_io_bitmap(&next_p->thread, &init_tss[cpu]);
```

3.3.4. Saving and Loading the FPU, MMX, and XMM Registers

- Starting with the Intel **80486DX**, the arithmetic floating-point unit (**FPU**) has been integrated into the CPU.
  - To maintain compatibility with older models, however, floating-point arithmetic functions are performed with **ESCAPE instructions** , which are instructions with a prefix byte ranging between 0xd8 and 0xdf.
  - These instructions act on the set of floating-point registers included in the CPU. Clearly, if a process is using ESCAPE instructions, the contents of the **floating-point registers belong to its hardware context** and should be saved.
- In **later Pentium** models, Intel introduced **MMX instructions** and are supposed to speed up the execution of multimedia applications (because they introduce a single-instruction multiple-data (SIMD) pipeline inside the processor)
  - MMX instructions act on the floating-point registers of the FPU.
  - The obvious **disadvantage** of this architectural choice is that programmers cannot mix floating-point instructions and MMX instructions.
  - The **advantage** is that operating system designers can ignore the new instruction set, because the same facility of the task-switching code for saving the state of the floating-point unit can also be relied upon to save the MMX state.

SSE & SSE2

- The **Pentium III** model extends that SIMD capability: it introduces the **SSE extensions** (Streaming SIMD Extensions), which adds facilities for handling floating-point values contained in eight 128-bit registers called the XMM registers.
  - Such registers do not overlap with the FPU and MMX registers , so SSE and FPU/MMX instructions may be freely mixed.
- The **Pentium 4** model introduces yet another feature: the **SSE2 extensions**, which is basically an extension of SSE supporting higher-precision floating-point values. SSE2 uses the same set of XMM registers as SSE.

Hardware support for handling the special registers

- The 80x86 microprocessors **do not automatically save** the FPU, MMX, and XMM registers in the TSS.

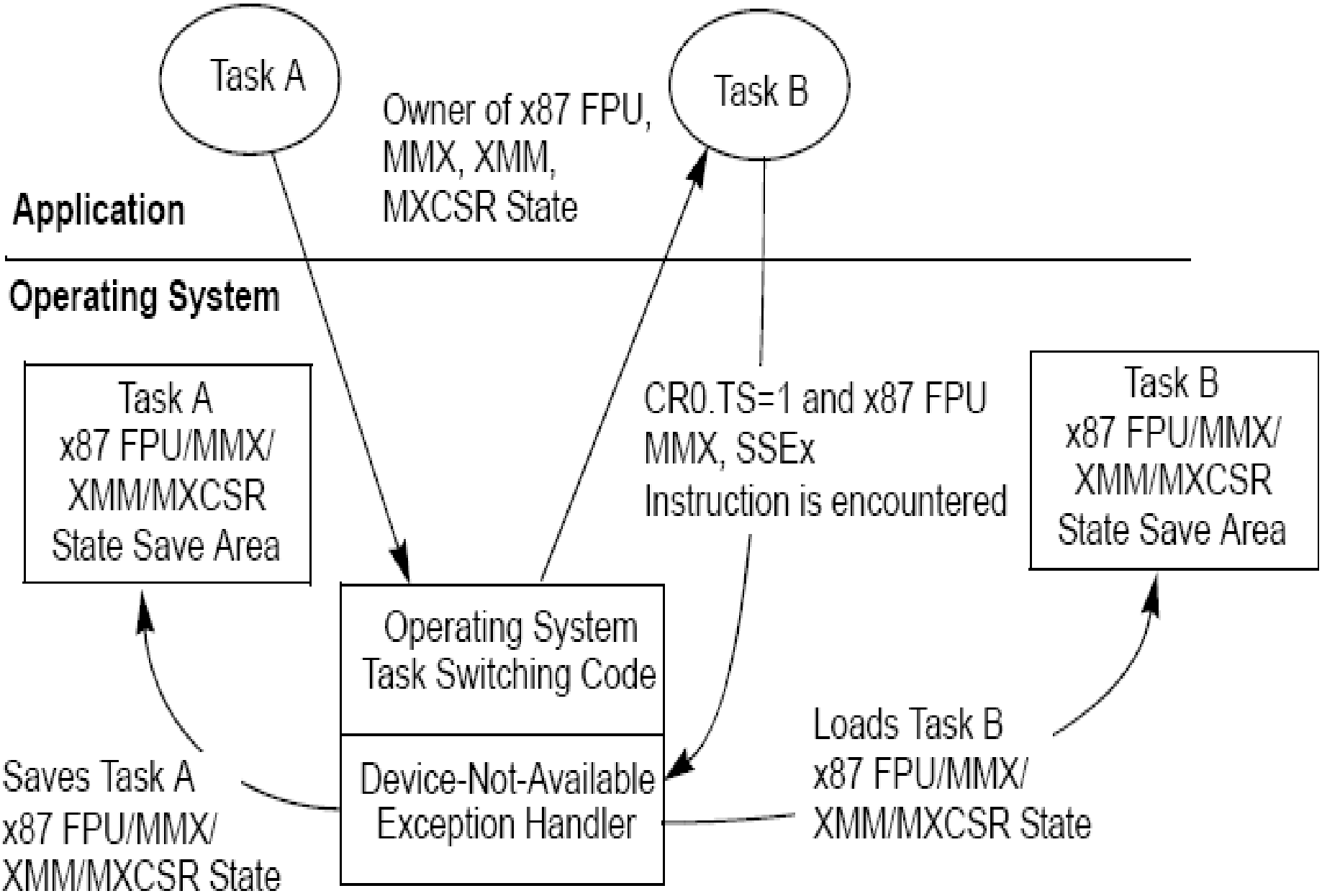
- However, they include some **hardware support that enables kernels** to save these registers only when needed. The hardware support consists of a **TS (Task-Switching) flag in the cr0 register**, which obeys the following rules:

- Every time a hardware context switch is performed, the TS flag is set.

- Every time an ESCAPE, MMX, SSE, or SSE2 instruction is executed when the TS flag is set, the **control unit raises a "Device not available " exception** (see Chapter 4).

- The TS flag allows the kernel to save and restore the FPU, MMX, and XMM registers **only when really needed**.

Saving x87 fpu, mmx, sse* during task switch



3.4. Creating Processes

- Unix operating systems rely heavily on process creation to satisfy user requests. For example, the shell creates a new process that executes another copy of the shell whenever the user enters a command.
- **Traditional Unix** systems treat all processes in the same way: **resources owned by the parent process are duplicated in the child process**. This approach makes process creation very slow and inefficient.
- In many cases, the child issues an **immediate execve( )** and wipes out the address space that was so carefully copied.
- **Modern Unix** kernels solve this problem by introducing **three different mechanisms**:
  - The **Copy On Write** technique allows both the parent and the child to read the same physical pages. Whenever either one tries to write on a physical page, the kernel copies its contents into a new physical page that is assigned to the writing process.
  - **Lightweight processes** allow both the parent and the child to share many per-process kernel data structures, such as the paging tables, the open file tables, and the signal dispositions.
  - The **vfork( ) system call** creates a process that shares the memory address space of its parent. To prevent the parent from overwriting data needed by the child, the parent's execution is blocked until the child exits or executes a new program.

3.4.1. The clone( ), fork( ), and vfork( ) System Calls

- Lightweight processes are created in Linux by using a function named **clone( )**, which uses the following parameters:
- **fn:** Specifies a function to be executed by the new process; when the function returns, the child terminates. The function returns an integer, which represents the exit code for the child process.
- **arg:** Points to data passed to the fn( ) function.
- **flags:** The low byte specifies the signal number to be sent to the parent process when the child terminates; the **SIGCHLD signal** is generally selected. The remaining three bytes encode a group of clone flags.
- **child_stack:** Specifies the User Mode stack pointer to be assigned to the esp register of the child process. The invoking process (the parent) should always allocate a new stack for the child.
- **tls:** Specifies the address of a data structure that defines a Thread Local Storage segment for the new lightweight process.
- **ptid:** Specifies the address of a User Mode variable of the parent process that will hold the PID of the new lightweight process.
- **ctid:** Specifies the address of a User Mode variable of the new lightweight process that will hold the PID of such process.

Table 3-8. Clone flags

- **CLONE_VM:** Shares the memory descriptor and all Page Tables (see Chapter 9).
- **CLONE_FS:** Shares the table that identifies the root directory and the current working directory, as well as the value of the bitmask used to mask the initial file permissions of a new file (the so-called file umask).
- **CLONE_FILES:** Shares the table that identifies the open files (see Chapter 12).
- **CLONE_SIGHAND:** Shares the tables that identify the signal handlers and the blocked and pending signals (see Chapter 11). If this flag is true, the CLONE_VM flag must also be set.
- **CLONE_PTRACE:** If traced, the parent wants the child to be traced too. Furthermore, the debugger may want to trace the child on its own; in this case, the kernel forces the flag to 1.
- **CLONE_VFORK:** Set when the system call issued is a vfork( ) (see later in this section).
- **CLONE_PARENT:** Sets the parent of the child (parent and real_parent fields in the process descriptor) to the parent of the calling process.
- **CLONE_THREAD:** Inserts the child into the same thread group of the parent, and forces the child to share the signal descriptor of the parent. The child's tgid and group_leader fields are set accordingly. If this flag is true, the CLONE_SIGHAND flag must also be set.

Linux implements fork() and vfork() with clone()

- The fork(), vfork(), and __clone() library calls all invoke the clone() system call with the requisite flags. The clone() system call, in turn, calls do_fork().

- The traditional **fork( )** system call is **implemented by Linux as a clone( ) system call** whose flags parameter specifies both a **SIGCHLD** signal and all the clone flags cleared, and whose child_stack parameter is the current parent stack pointer.

  - Therefore, the parent and child temporarily share the same User Mode stack. But thanks to the **Copy On Write mechanism**, they usually **get separate copies** of the User Mode stack as soon as one tries to change the stack.

- The **vfork( )** system call, introduced in the previous section, is implemented by Linux as a **clone( )** system call whose flags parameter specifies both a **SIGCHLD** signal and the flags **CLONE_VM** and **CLONE_VFORK**, and whose child_stack parameter is equal to the current parent stack pointer.

3.4.1.1. The do_fork( ) function

- **Here are the main steps performed by do_fork( ):**
  - Allocates a **new PID** for the child.
  - Invokes **copy_process()** to make a **copy of the process descriptor**. If all needed resources are available, this function returns the address of the task_struct descriptor just created.
  - If the **CLONE_VFORK** flag is specified, it **inserts the parent process in a wait queue** and suspends it **until the child releases its memory address space** (that is, until the child either terminates or executes a new program).

3.4.1.2. The copy_process( ) function

- The copy_process( ) function sets up the **process descriptor** and any other **kernel data structure required for a child's execution.**

- Its parameters are the same as do_fork( ), plus the PID of the child. Here is a description of its most significant steps:

  - Sanity checks (CLONE_* flags combinations)

  - Invokes dup_task_struct( ) to get the process descriptor for the child.

  - Checks whether the value stored in current->signal->rlim[RLIMIT_NPROC].rlim_cur is smaller than or equal to the current number of processes owned by the user. If so, an error code is returned, unless the process has root privileges

  - Checks that the **number of processes in the system** (stored in the nr_threads variable) does not exceed the value of the max_threads variable.

    - The default value of this variable depends on the amount of RAM in the system. The general rule is that the space taken by all thread_info descriptors and Kernel Mode stacks cannot exceed 1/8 of the physical memory.

    - However, the system administrator may change this value by writing in the **/proc/sys/kernel/threads-max** file.

copy_process( )

- Invokes copy_semundo( ), copy_files( ), copy_fs( ), copy_sighand( ), copy_signal( ), copy_mm( ), and copy_namespace( ) to **create new data structures and copy into them the values of the corresponding parent process data structures**, unless specified differently by the clone_flags parameter.

- Invokes **copy_thread( )** to initialize the **Kernel Mode stack of the child** process with the **values contained in the CPU registers when the clone( ) system call was issued**.

  - However, the function forces the value 0 into the field corresponding to the eax register (this is the child's return value of the clone( ) system call).

  - The **thread.esp** field in the descriptor of the child process is initialized with the base address of the **child's Kernel Mode stack**, and the address of an assembly language function (**ret_from_fork( )**) is stored in the **thread.eip** field.

  - If the parent process makes use of an **I/O Permission Bitmap, the child gets a copy** of such bitmap. Finally, if the CLONE_SETTLS flag is set, the child gets the TLS segment specified by the User Mode data structure pointed to by the tls parameter of the clone( ) system call.

copy_process( )

- Initializes the **tsk->exit_signal** field with the signal number encoded in the low bits of the clone_flags parameter, unless the CLONE_THREAD flag is set, in which case initializes the field to -1.
  - **only the death of the last member of a thread group** (usually, the thread group leader) **causes a signal notifying the parent** of the thread group leader.
- Invokes **sched_fork( )** to complete the initialization of the scheduler data structure of the new process. The function also sets the state of the new process to **TASK_RUNNING.**
  - in order to keep process scheduling fair, **the function shares the remaining timeslice of the parent between the parent and the child**.
- **Initializes the fields that specify the parenthood relationships**.

Let's go back to what happens after do_fork() terminates

- Now we have a **complete child process in the runnable state**. But it isn't actually running. It is up to the scheduler to decide when to give the CPU to this child.

- At some future process switch, the schedule bestows this favor on the child process by loading a few CPU registers with the values of the thread field of the child's process descriptor.

- The new process then starts its execution right at the end of the fork( ), vfork( ), or clone( ) system call.

- The child process executes the same code as the parent, except that the fork returns a 0. The developer of the application can exploit this fact, in a manner familiar to Unix programmers, by inserting a conditional statement in the program based on the PID value that forces the child to behave differently from the parent process.