Chapter 3

Processes

A study material for the students of GLS University

3.1. Processes
3.2. Lightweight processes and threads
3.3. Process descriptor
3.4. Process switch
3.5. Creating processes
3.6. Destroying Processes

3.1 Processes

- An instance of a program in execution
- If 16 users are running *at once, there are 16 separate processes*
- Processes are often called *tasks or threads in the Linux* source code
- When Process is created
  - almost identical to its parent
  - It receives a copy of the parent's address space
  - executes the same code as the parent
  - beginning at the next instruction following the process creation system call
  - The parent and child may share the pages containing the program code
  - They have separate copies of the data, so that changes by the child to a memory location are invisible to the parent (and vice versa)

3.2. Lightweight processes and Threads

Lightweight processes

- *Support for multithreaded applications*
- Two lightweight processes may share some resources, like the address space, the open files
- Whenever one of them modifies a shared resource, the other immediately sees the change
- Two processes must synchronize themselves when accessing the shared resource

Threads

- Normal process
- Implement multithreaded applications is to associate a lightweight process with each thread
- The threads can access the same set of application data structures by simply sharing the same memory address space, the same set of open files
- At the same time, each thread can be scheduled independently by the kernel so that one may sleep while another remains runnable
- . Multithreaded applications are best handled by kernels that support “thread groups”
- ***Thread group*** *is basically a set of lightweight processes* that implement a multithreaded application

3.3. Process Descriptor

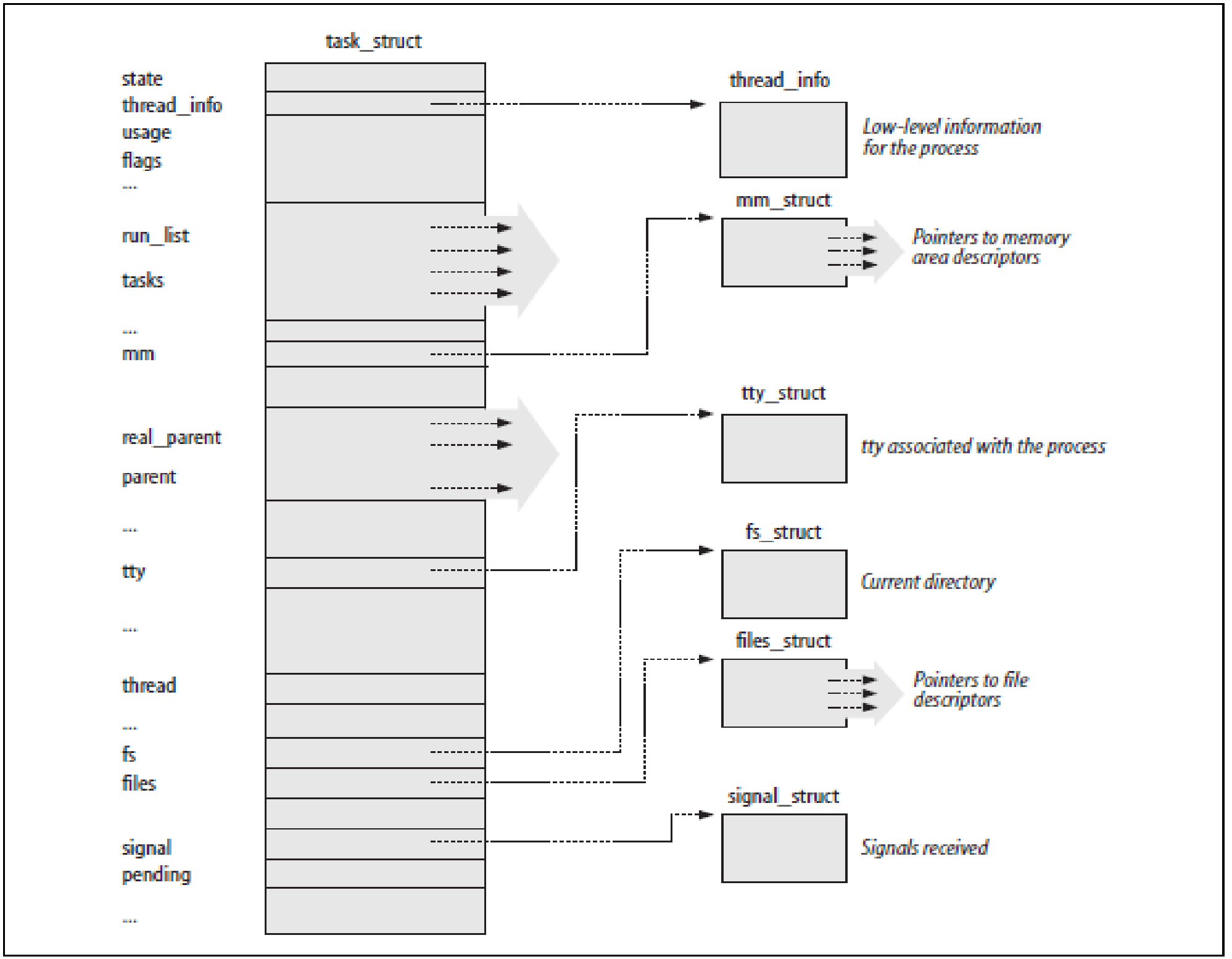
1. Process State
2. Identifying a Process
3. Relationships Among Processes
4. How Processes Are Organized
5. Process Resource Limits

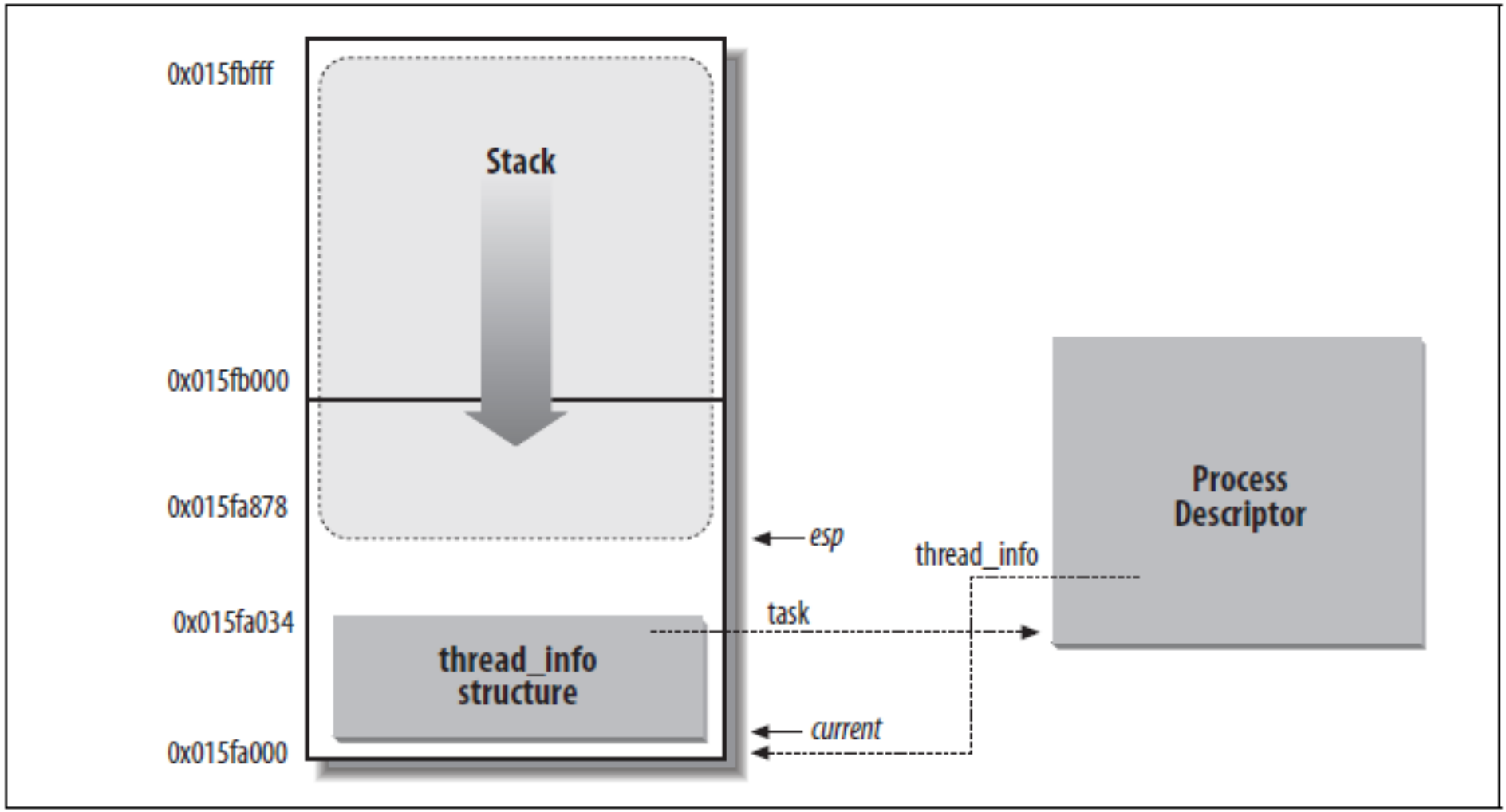
- ***Process descriptor—****a task*_struct type structure whose fields contain all the information related to a single process
- Process descriptor contains several pointers to other data structures that, in turn, contain pointers to other structures
- Figure: Process Descriptor
- The six data structures on the right side of the figure refer to specific resources owned by the process.
- This chapter focuses on two types of fields that refer to the **process state and to process parent/child relationships**

3.3.1 Process State

- Describes what is currently happening to the process
- Consists of an array of flags, each of which describes a possible process state
- Mutually exclusive - exactly one flag of state always is set; the remaining flags are cleared
- **Process states:**

1. TASK_RUNNING
   Process is either executing on a CPU or waiting to be executed
2. TASK_INTERRUPTIBLE
   Process is suspended (sleeping) until some condition becomes true
3. TASK_UNINTERRUPTIBLE
   Under certain specific conditions in which a process must wait until a given event occurs without being interrupted
4. TASK_STOPPED
   Process execution has been stopped
5. TASK_TRACED
   Process execution has been stopped by a debugger.


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


0x015fbfff
Stack
0x015fb000
0x015fa878
esp
thread_info
Process
Descriptor
0x015fa034
task
thread_info
structure
current
0x015fa000

exit_state field:

Two more state

1. 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. (Parent might need data from child)

2. 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

3.3.2. Identifying a Process

- Unique 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
- 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

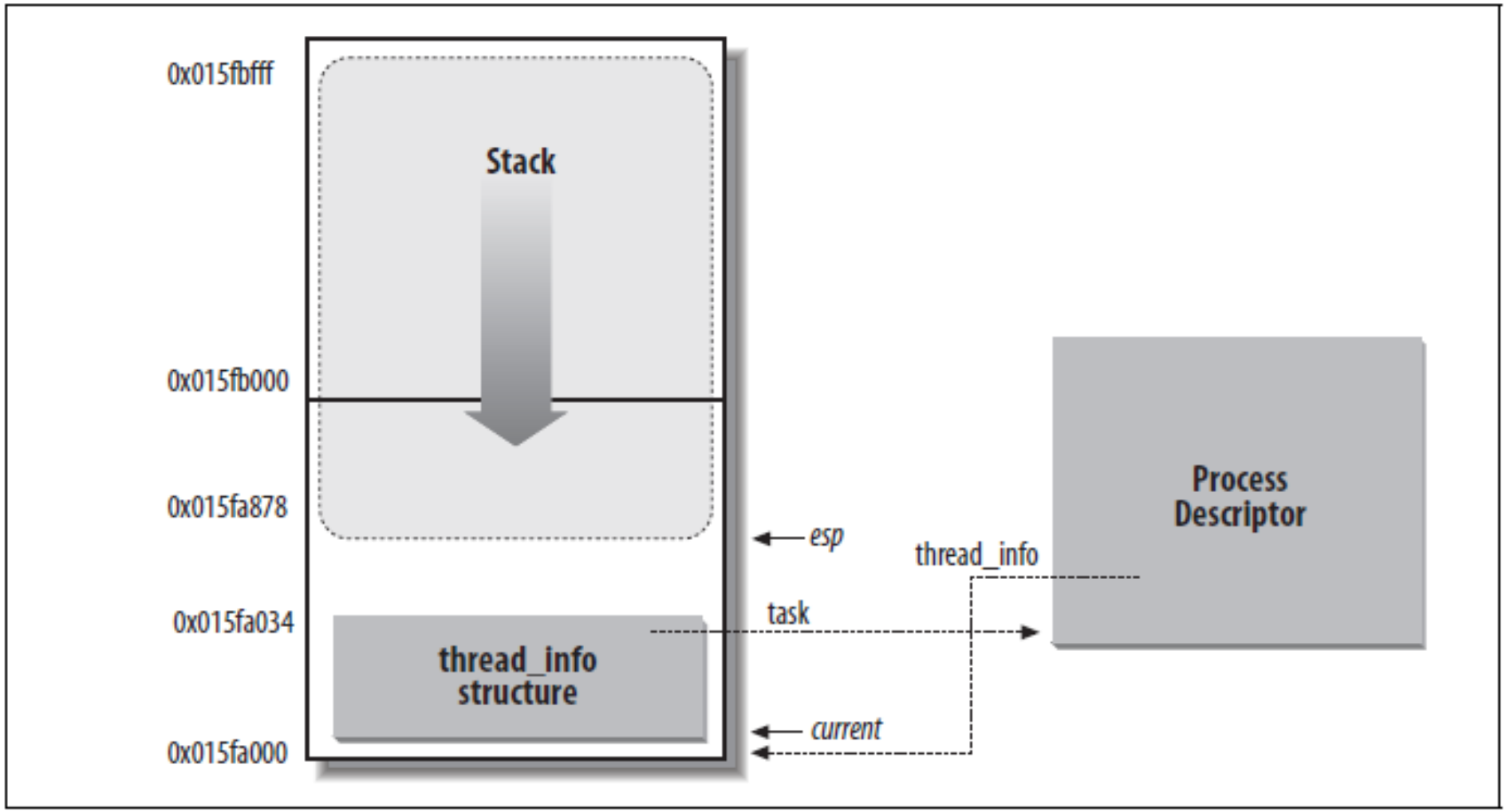
Process descriptors handling

- Small data structure linked to the process descriptor, namely the **thread_info structure**, and the Kernel Mode process **stack**

- The **esp register** is the CPU stack pointer, which is used to address the stack's top location

- The thread_info structure stored starting at address 0x015fa000, and the stack is stored starting at address 0x015fc000.

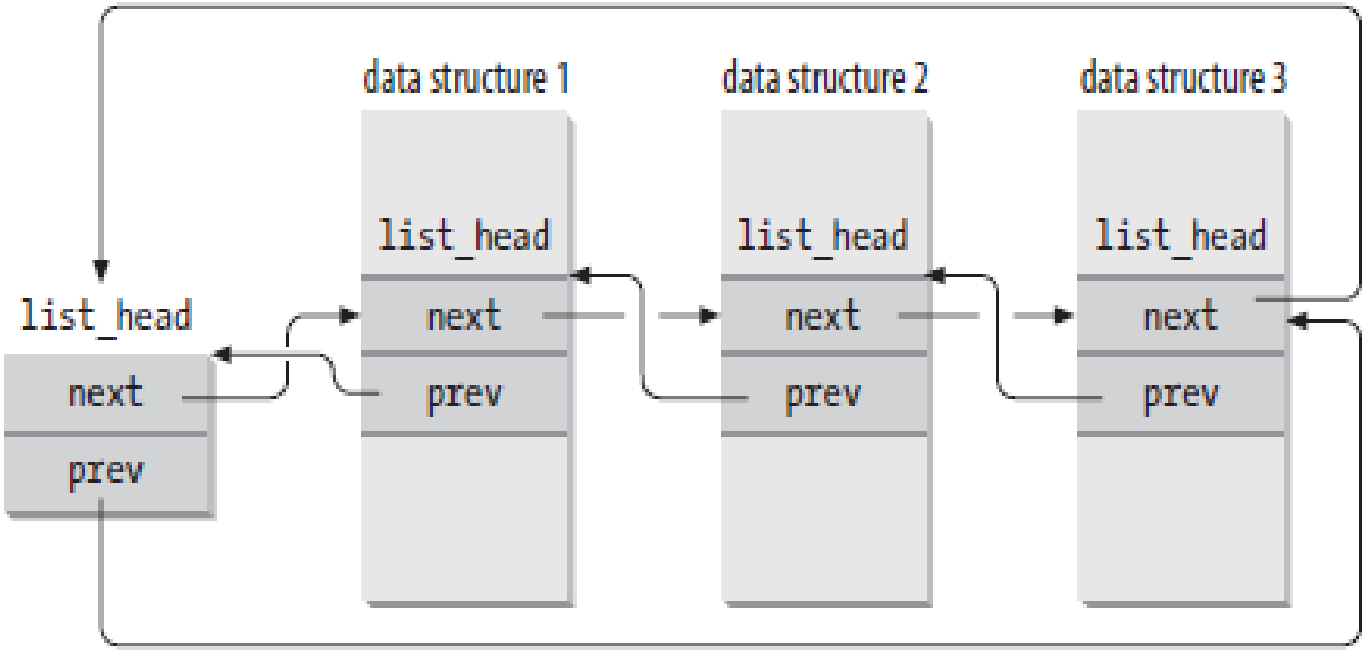
- The kernel uses the alloc_thread_info and free_thread_info macros to allocate and release the memory area storing a thread_info structure and a kernel stack


0x015fbfff
Stack
0x015fb000
0x015fa878
esp
thread_info
Process
Descriptor
0x015fa034
task
thread_info
structure
current
0x015fa000

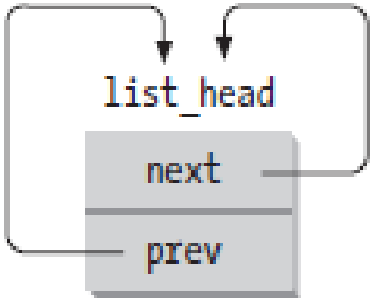
Doubly linked lists

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

Name	Description
`list_add(n,p)`	Inserts an element pointed to by `n` right after the specified element pointed to by `p`.
`list_add_tail(n,p)`	Inserts an element pointed to by `n` right before the specified element pointed to by `p`.
`list_del(p)`	Deletes an element pointed to by `p`.
`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`.



(a) a doubly linked listed with three elements



(b) an empty doubly linked list

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 SET_LINKS and REMOVE_LINKS macros are used to insert and to remove a process descriptor in the process list, respectively

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

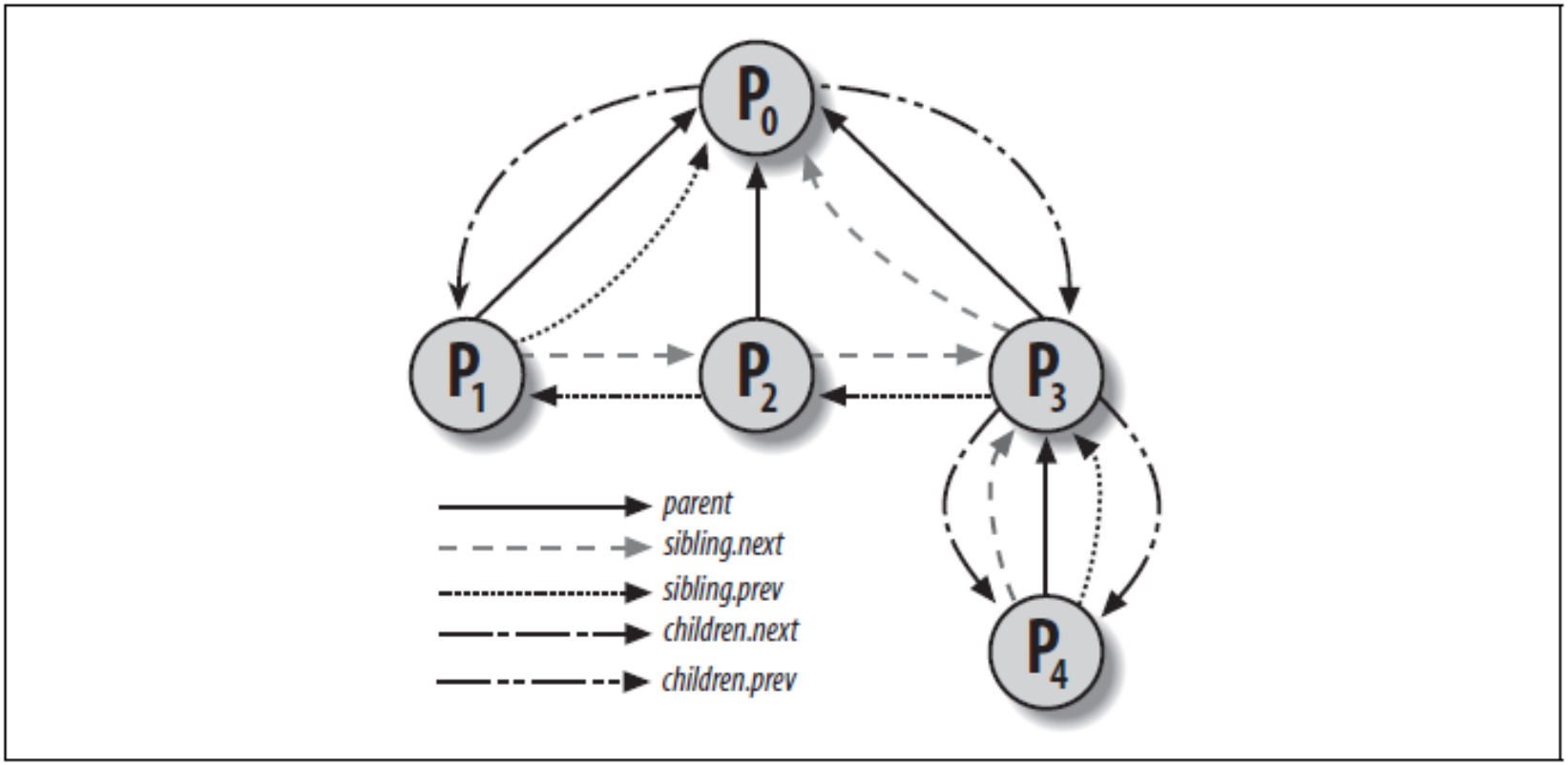
- The enqueue_task(p,array) function inserts a process descriptor into a runqueue list

- The dequeue_task(p,array) function removes a process descriptor from a runqueue list.

3.3.3. Relationships Among Processes

- Processes created by a program have a parent/child relationship
- When a process creates multiple children, these children have sibling relationships
- Several fields must be introduced in a process descriptor to represent these relationships

Field name	Description
real_parent	Points to the process descriptor of the process that created P or to the descriptor of process 1
parent	Points to the current parent of 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 P0 successively created P1, P2, and P3. Process P3, in turn, created process P4.

3.3.4 How Processes Are Organized

•The runqueue lists group all processes in a **TASK_RUNNING state**

•**Wait queues**

- Wait queues have several uses in the kernel, particularly for interrupt handling, process synchronization, and timing

- Wait queue represents a set of sleeping processes, which are woken up by the kernel when some condition becomes true

- There are two kinds of sleeping processes: *exclusive processes* are selectively woken up by the kernel, while *nonexclusive processes* are always woken up by the kernel when the event occurs

Handling wait queues

•A new wait queue head may be defined by using the DECLARE_WAIT_QUEUE_HEAD

•The init_waitqueue_head() function used to initialize a wait queue

•The add_wait_queue() function inserts a nonexclusive process in the first position of a wait queue list

•The add_wait_queue_exclusive() function inserts an exclusive process in the last position of a wait queue list

•The remove_wait_queue() function removes a process from a wait queue list

3.3.5 Process Resource Limits

- Which specify the amount of system resources it can use
- ***Resource limits Fields:***

Field name	Description
RLIMIT_AS	The maximum size of process address space, in bytes.
RLIMIT_CPU	The maximum CPU time for the process, in seconds.
RLIMIT_DATA	The maximum heap size, in bytes.
RLIMIT_FSIZE	The maximum file size allowed, in bytes.
RLIMIT_NPROC	The maximum number of processes that the user can own
RLIMIT_RSS	The maximum number of page frames owned by the process
RLIMIT_SIGPENDING	The maximum number of pending signals for the process
RLIMIT_STACK	The maximum stack size, in bytes.

3.4 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 is called *process switch, task switch, or context switch*

3.4 Process Switch

1. Hardware Context
2. Task State Segment
3. Performing the Process Switch

3.4.1. Hardware Context

- Each process can have its **own address space** and all processes have to **share the CPU registers**

- 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**

- 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**

- **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

3.4.2 Task State Segment (TSS)

- Store hardware contexts

- Although Linux doesn't use hardware context switches, but this is done for two main reasons:
    1. When CPU switches from User Mode to Kernel Mode, it fetches the address of the Kernel Mode stack from the TSS
    2. When a User Mode process attempts to access an I/O port - Permission Bitmap stored in the TSS to verify whether the process is allowed to address the port.

- Each TSS has its own 8-byte *Task State Segment Descriptor (TSSD). This descriptor* includes a 32-bit Base field that points to the TSS starting address and a 20-bit Limit field

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

3.4.3 Performing the Process Switch

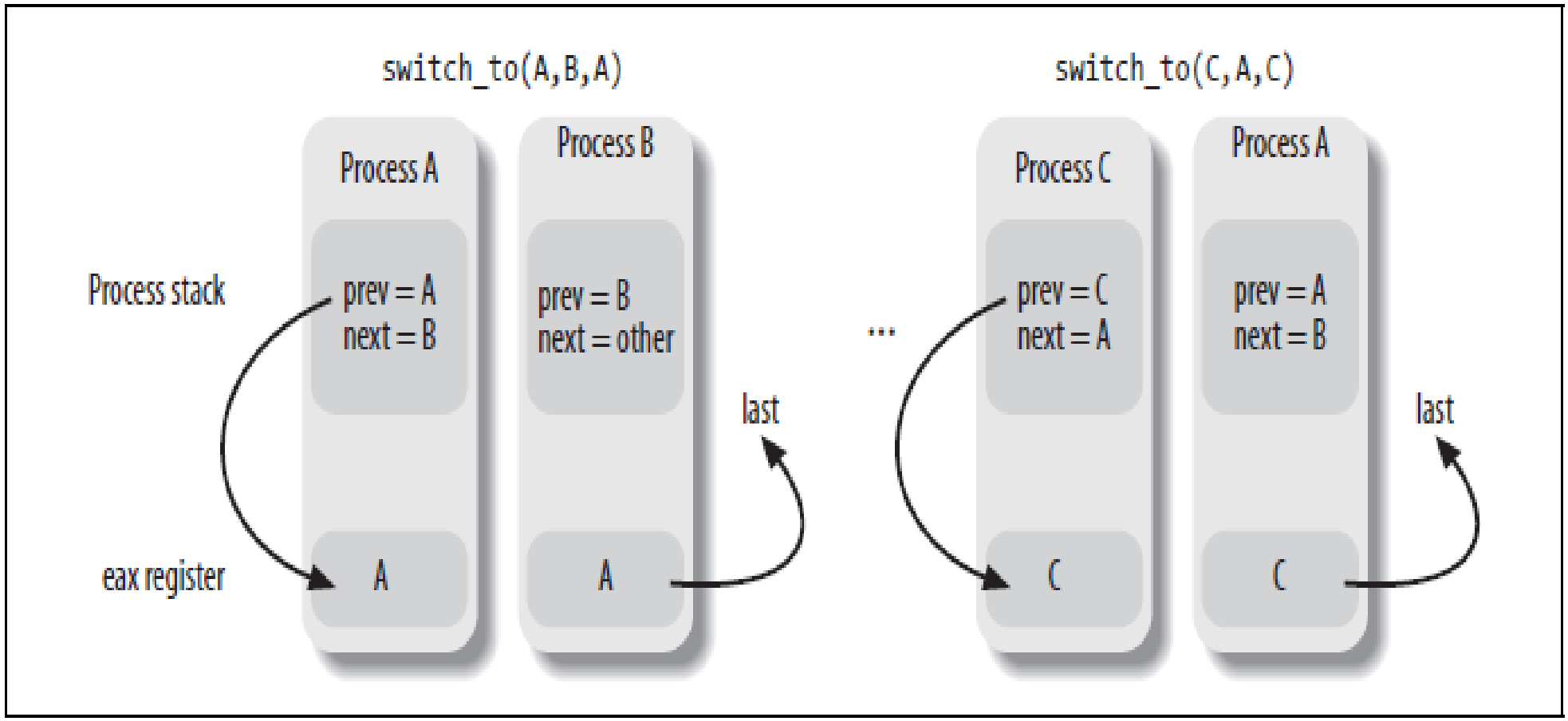
- A process switch may occur at just one well-defined point: the schedule() function
- Essentially, every process switch consists of two steps:
  1. Switching the Page Global Directory to install a new address space
  2. Switching the Kernel Mode stack and the hardware context, which provides all the information needed by the kernel to execute the new process, including the CPU registers.
- Again, we assume that prev points to the descriptor of the process being replaced, and next to the descriptor of the process being activated. (prev and next are local variables of the schedule() function)

The switch_to macro

- The second step of the process switch is performed by the switch_to macro
- First of all, the macro has three parameters, called **prev, next, and last**
- prev and next: they are descriptor address of the process being replaced and the descriptor address of the new process, respectively

•in any process switch three processes are involved, not just two. Suppose the kernel decides to switch off process A and to activate process B. In the schedule() function, prev points to A's descriptor and next points to B's descriptor. As soon as the switch_to macro deactivates A, the execution flow of A freezes.

•Later, when the kernel wants to **reactivate A,** it must switch off another process C by executing another switch_to macro with prev pointing to C and next pointing to A. When A resumes its execution flow, it finds its old Kernel Mode stack, so the prev local variable points to A's descriptor and next points to B's descriptor. The scheduler, which is now executing on behalf of process A, has lost any reference to C.


switch_to(A,B,A)
switch_to(C,A,C)
Process A
Process B
Process C
Process A
Process stack
prev = A
next = B
prev = B
next = other
prev = C
next = A
prev = A
next = B
last
last
eax register
A
A
C
C

The _ _switch_to( ) function

The _ _switch_to( ) function does the bulk of the process switch started by the switch_to( ) macro.

3.5 Creating Processes

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

1. fork(): creates a new child process, which is a complete copy of the parent process. Child and parent processes use different virtual address spaces, which is initially populated by the same memory pages.

2. vfork(): creates a new child process, which is a "quick" copy of the parent process. In contrast to the system call fork(), child and parent processes share the same virtual address space.

3. clone(): creates a new child process. Various parameters of this system call, specify which parts of the parent process must be copied into the child process and which parts will be shared between them. As a result, this system call can be used to create all kinds of execution entities, starting from threads and finishing by completely independent processes. clone() is used to implement multiple threads

3.6 Destroying Processes

- Process Termination

- Two system calls that terminate a User Mode application

1. The exit_group() system call, which terminates a full thread group, that is, a whole multithreaded application.

2. The _exit() system call, which terminates a single process, regardless of any other process in the thread group of the victim.

THANK YOU