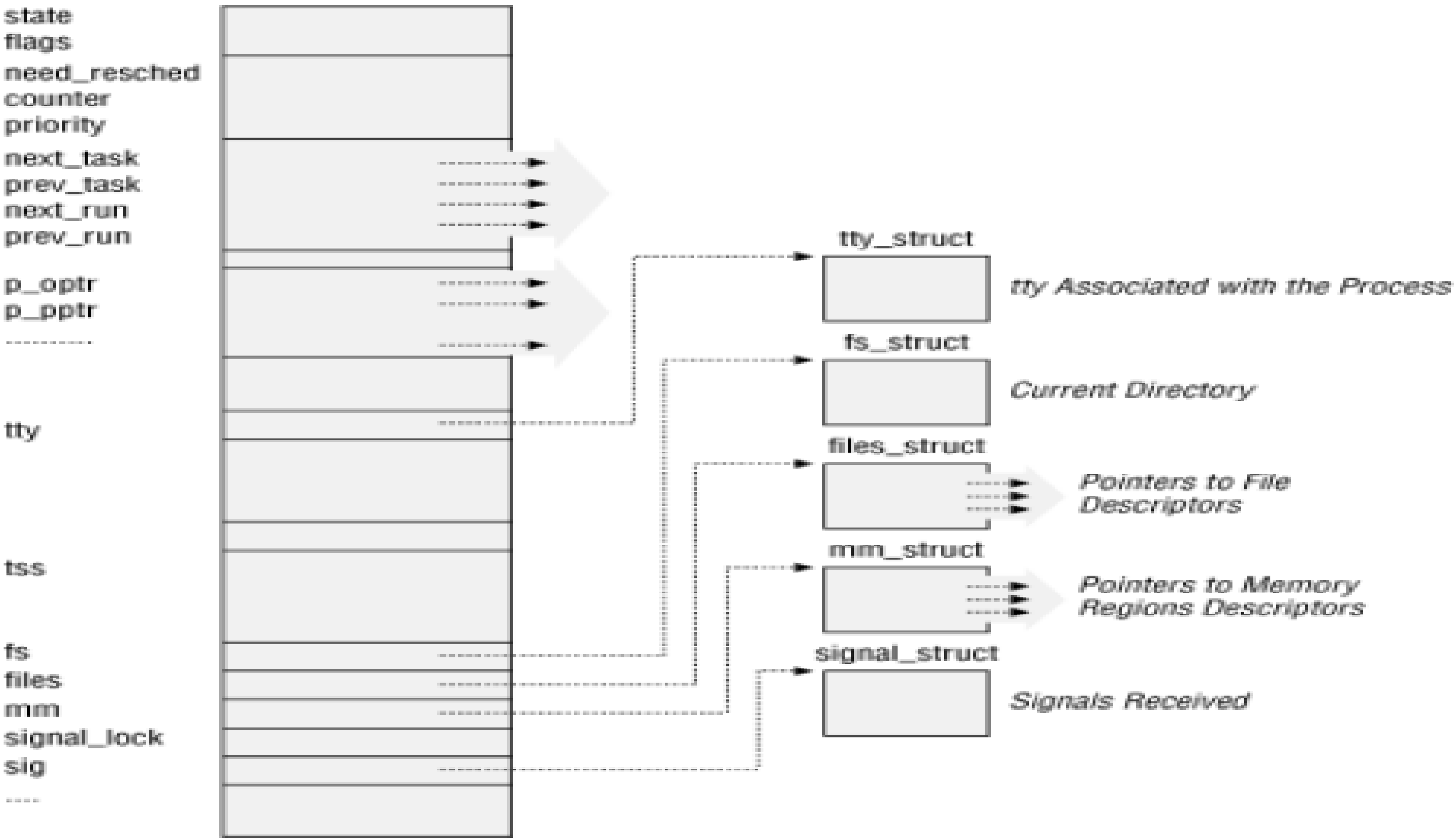
Processes

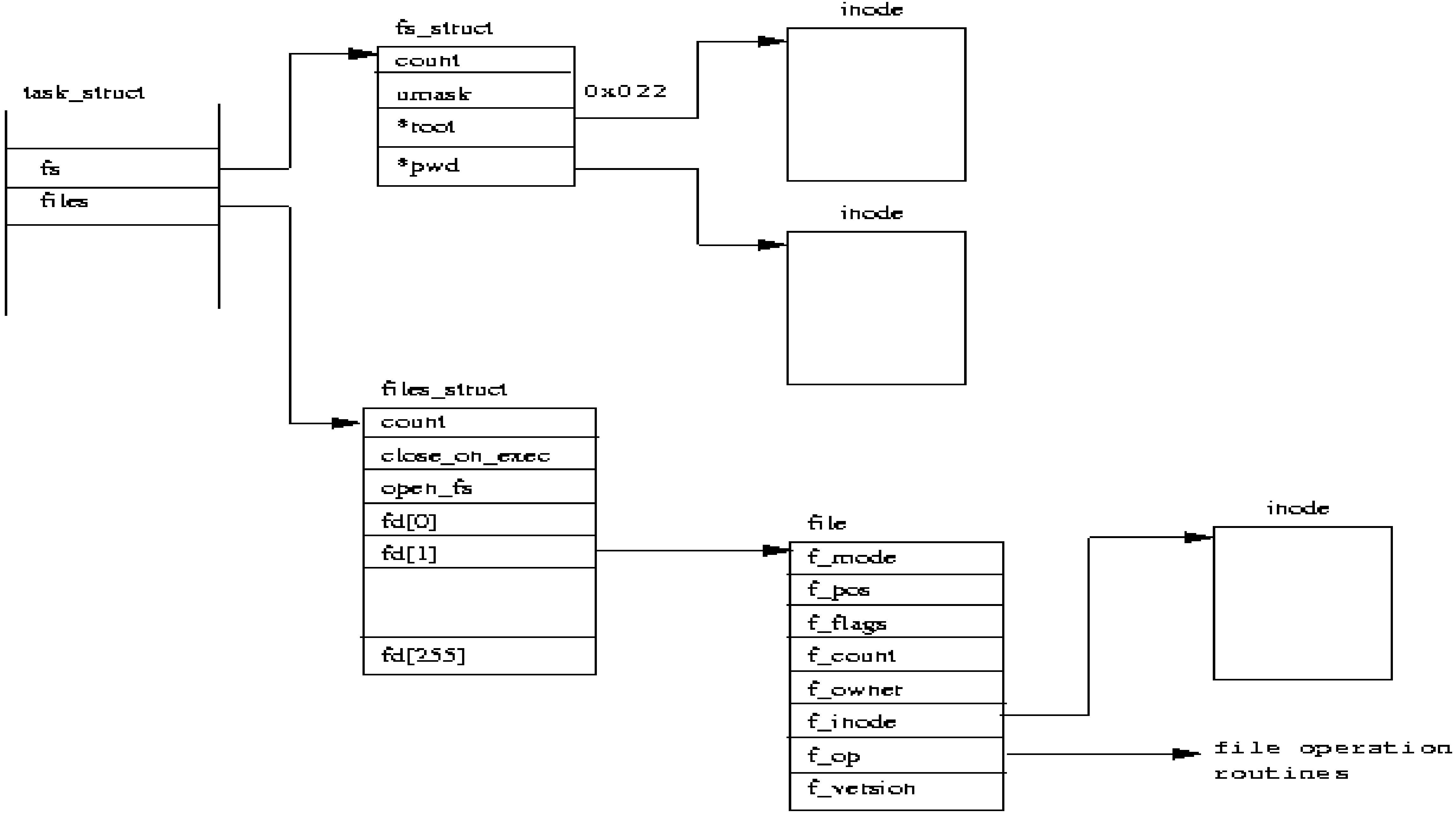
Introduction

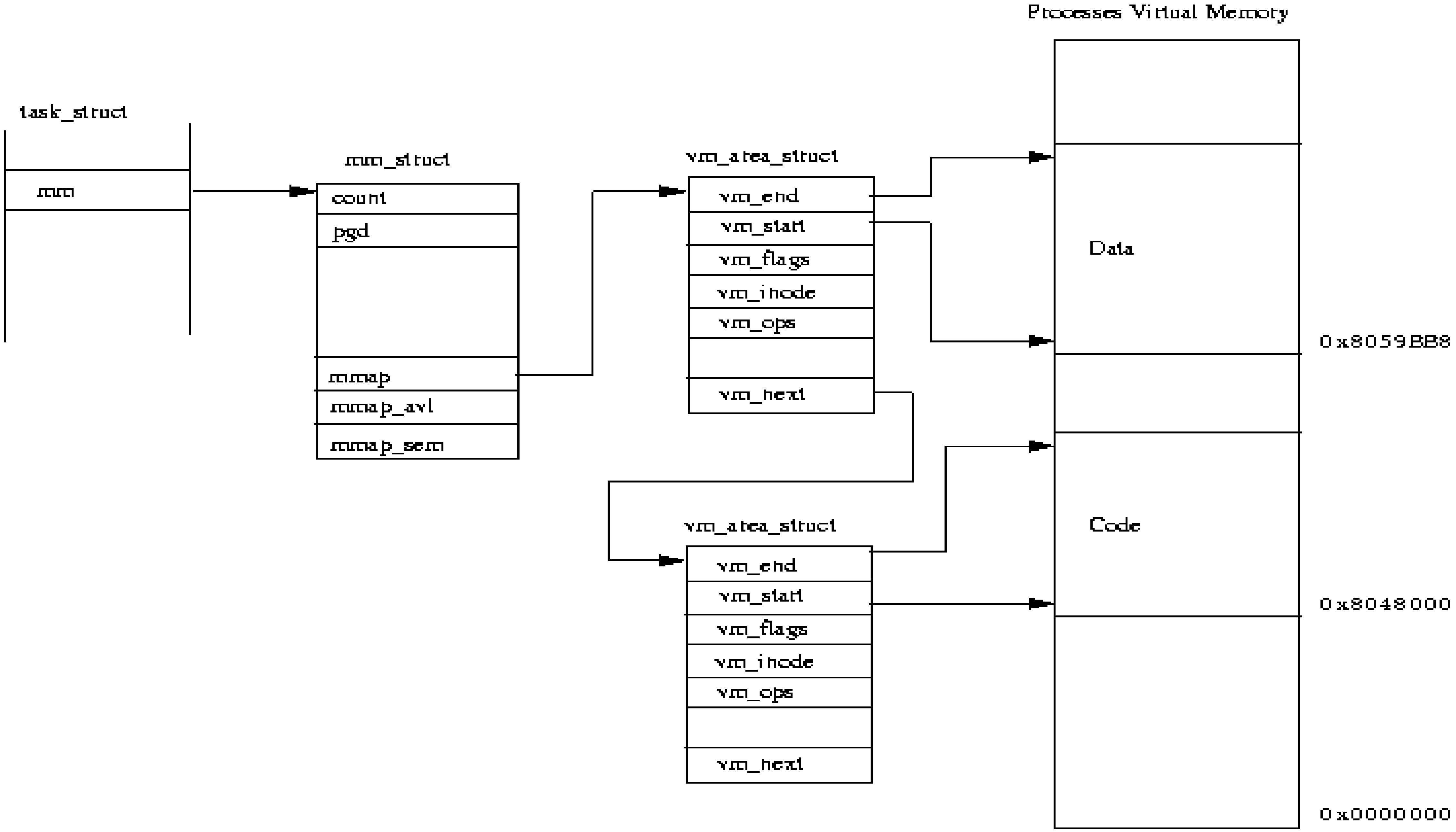
- A process is usually defined as an instance of a program in execution.
- Processes are often called "tasks" in Linux source code.

Process descriptor

- The role of a process descriptor is to make sure how the kernel is managing each processes , for instance, processes priority, whether it is running on the CPU or blocked some event, what address space has been assigned to it, which files it is allowed to address, and so on.
- That is, f a **task_struct** type structure whose fields contain all the information related to a single process.
- It is a structure associated with every process existing process in the system.
- The process descriptor is rather complex, as it is repository of all so much information. Not only does it contain many fields itself, but some contain pointers to other data structures that, in turn, contain pointers to other structures.


state
flags
need_resched
counter
priority
next_task
prev_task
next_run
prev_run
p_optr
p_pptr
....
tty
tss
fs
files
mm
signal_lock
sig
....
tty_struct
tty Associated with the Process
fs_struct
Current Directory
files_struct
Pointers to File Descriptors
mm_struct
Pointers to Memory Regions Descriptors
signal_struct
Signals Received


task_struct
fs
files
fs_struct
count
umask
*root
*pwd
0x022
inode
inode
files_struct
count
close_on_exec
open_fs
fd[0]
fd[1]
fd[255]
file
f_mode
f_pos
f_flags
f_count
f_owner
f_inode
f_op
f_version
inode
file operation
routines


task_struct
mm
mm_struct
count
pgd
mmap
mmap_avl
mmap_sem
vm_area_struct
vm_end
vm_start
vm_flags
vm_inode
vm_ops
vm_next
vm_area_struct
vm_end
vm_start
vm_flags
vm_inode
vm_ops
vm_next
Processes Virtual Memory
Data
Code
0x8059BB8
0x8048000
0x0000000

Process State

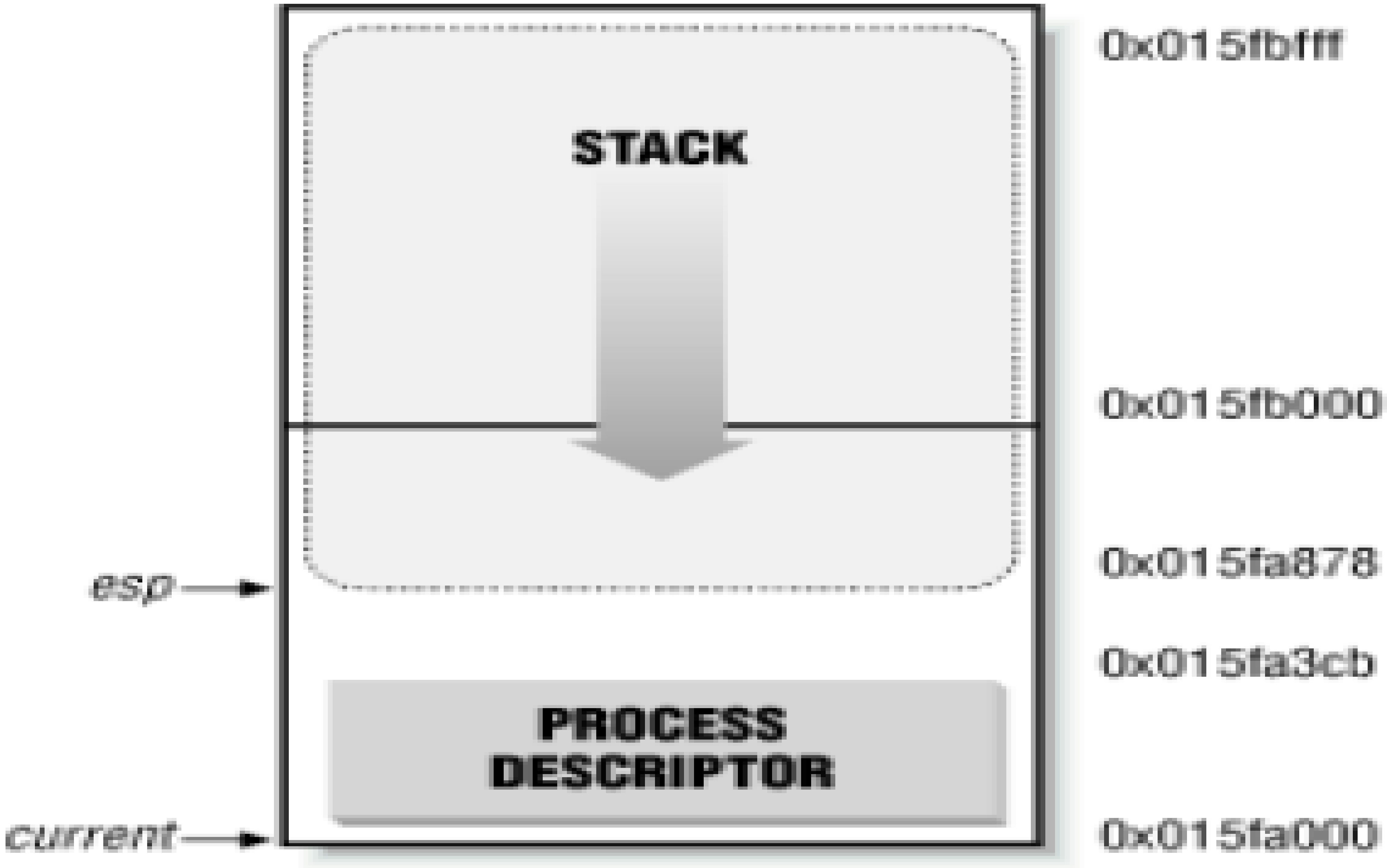
- As the name implies, the state field of the process descriptor describes what is currently happening to the process.
- It consists of an array of flags, each of which describes a possible process state.
- These states are mutually exclusive, and hence exactly one flag of state is set; the remaining flags are cleared.
- The following are the possible process states:
- TASK_RUNNING
    - The process is either executing on the 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, that is, put its state back to TASK_RUNNING.
- TASK_UNINTERRUPTIBLE
    - Like the previous state, 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. When a process is being monitored by another any signal may put the process in the TASK_STOPPED state.
- TASK_ZOMBIE
  - Process execution is terminated, but the parent process has not yet issued a wait( )- like system call (wait( ), wait3( ), wait4( ), or waitpid( )) to return 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 could need it.

Identifying a Process

- Linux processes can share a large portion of their kernel data structures—an measure known as lightweight processes—each process has its own process descriptor.
- Each execution context that can be independently scheduled must have its own process descriptor.
- Any Unix-like operating system, on the other hand, allows users to identify processes by means of a number called the Process ID (or PID).
- The PID is a 32-bit unsigned integer 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 incremented by one.
- The maximum PID number allowed on Linux is 32767. When the kernel creates the 32768th process in the system, it must start recycling the lower unused PIDs.
- **The task array** - Processes are dynamic entities whose lifetimes in the system range from a few milliseconds to months. Thus, the kernel must be able to handle many processes at the same time.
- The kernel reserves a global static array of size NR_TASKS called task in its own address space.
- The elements in the array are process descriptor pointers; a null pointer indicates that a process descriptor hasn't been associated with the array entry.

- **Storing a process descriptor** - The task array contains only pointers to process descriptors, not the sizable descriptors.
- Since processes are dynamic entities, process descriptors are stored in dynamic memory rather than in the memory area permanently assigned to the kernel.
- Linux stores two different data structures for each process in a single 8 KB memory area: the process descriptor and the Kernel Mode process stack.
- From the previous chapter, we learned that a process in Kernel Mode accesses a stack contained in the kernel data segment, which is different from the stack used by the process in User Mode.
- Since kernel control paths make little use of the stack only a few thousand bytes of kernel stack are required.


0x015fbfff
STACK
0x015fb000
0x015fa878
esp
0x015fa3cb
PROCESS DESCRIPTOR
current
0x015fa000

- The esp register is the CPU stack pointer, which is used to address the stack's top location.
- On Intel 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 memory area.
- After switching from User Mode to Kernel Mode, the esp register contains the address 0x015fc000. The process descriptor is stored starting at address 0x015fa000.
- The value of the esp is decremented as soon as data is written into the stack. Since the process descriptor is less than 1000 bytes long, the kernel stack can expand up to 7200 bytes.
- **The current macro -** The pairing between the process descriptor and the Kernel Mode stack just described offers a key benefit in terms of efficiency: the kernel can easily obtain the process descriptor pointer of the process currently running on the CPU from the value of the esp register.

- Since the memory area is 8 KB (213 bytes) long, all the kernel has to do is mask out the 13 least significant bits of esp to obtain the base address of the process descriptor.
- This is done by the current macro, which produces some Assembly instructions like the following:

```
movl $0xffffe000, %ecx
andl %esp, %ecx
movl %ecx, p
```

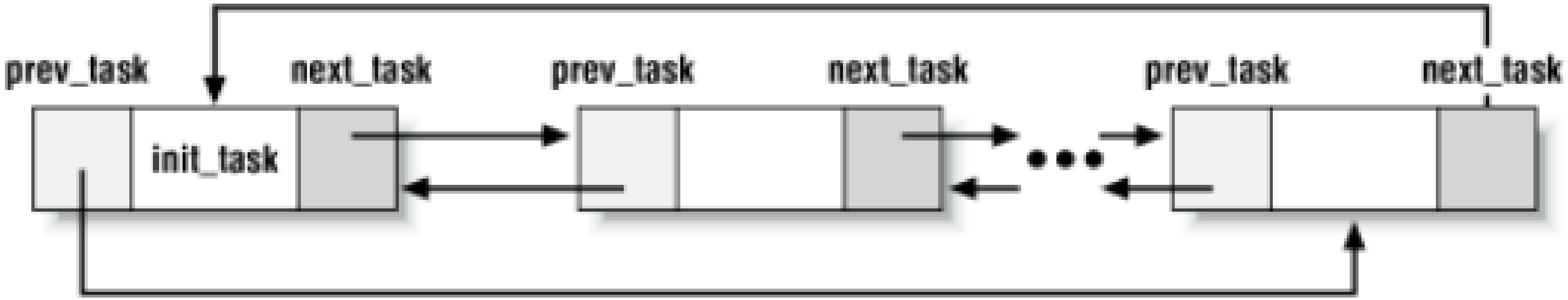
- After executing these three instructions, the local variable p contains the process descriptor pointer of the process running on the CPU.
- Another advantage of storing the process descriptor with the stack emerges on multiprocessor systems: the correct current process for each hardware processor can be derived just by checking the stack.
- Therefore, we use the current macro often appears in kernel code as a prefix to fields of the process descriptor.
- For example, **current->pid** returns the process ID of the process currently running on the CPU.

- A small cache consisting of **EXTRA_TASK_STRUCT** memory areas is used to avoid unnecessarily invoking the memory allocator.
- To understand the purpose of this cache, assume for instance that some process is destroyed and that, right afterward, a new process is created.
- Without the cache, the kernel would have to release an 8 KB memory area to the memory allocator and then, immediately afterward, request another memory area of the same size.
- This is an example of memory cache, a software mechanism introduced to bypass the Kernel Memory Allocator.
- The **task_struct_stack** array contains the pointers to the process descriptors in the cache.
- Its name comes from the fact that process descriptor releases and requests are implemented respectively as "push" and "pop" operations on the array:
- **free_task_struct()** - This function releases the 8 KB task_union memory areas and places them in the cache unless it is full.
- **alloc_task_struct( )**- This function allocates 8 KB task_union memory areas. The function takes memory areas from the cache if it is at least half-full or if there isn't a free pair of consecutive page frames available.

- **The process list** - To allow an efficient search through processes of a given type the kernel creates several lists of processes.
- Each list consists of pointers to process descriptors. A list pointer is embedded right in the process descriptor's data structure.
- A circular doubly linked list links together all existing process descriptors; we will call it the process list.
- The **prev_task** and **next_task** fields of each process descriptor are used to implement the list.
- The head of the list is the **init_task** descriptor referenced by the first element of the task array: it is the ancestor of all processes, and it is called process 0 or swapper.
- The **prev_task** field of **init_task** points to the process descriptor inserted last in the list.
- The **SET_LINKS** and **REMOVE_LINKS** macros are used to insert and to remove a process descriptor in the process list, respectively.
- These macros also take care of the parenthood relationship of the process.
- Another useful macro, called **for_each_task** , scans the whole process list. It is defined as:

```
#define for_each_task(p) \
for (p = &init_task ; (p = p->next_task) != &init_task ; )
```

- The macro is the loop control statement after which the kernel programmer supplies the loop.



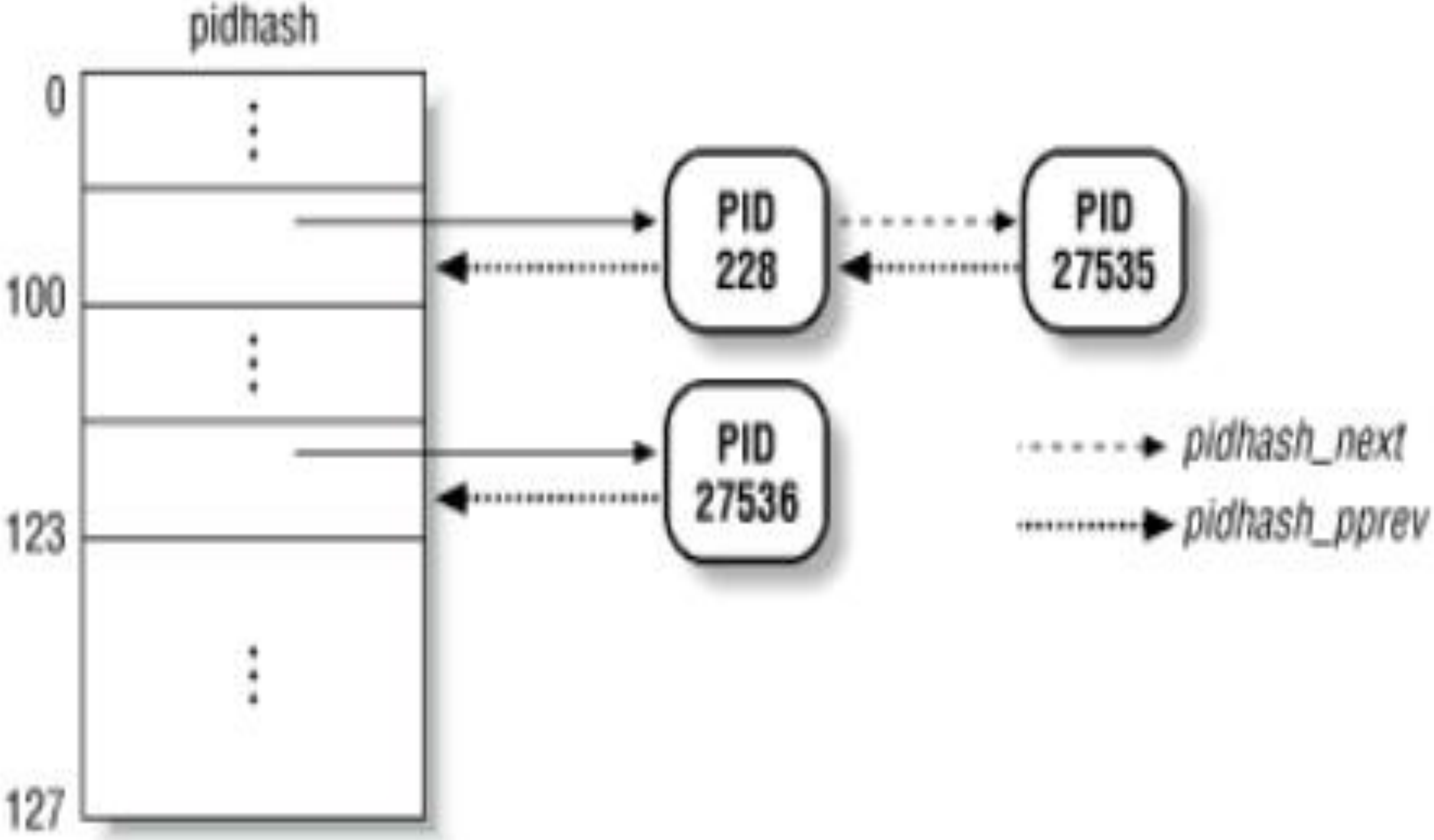
prev_task
next_task
init_task
prev_task
next_task
prev_task
next_task

- **The list of TASK_RUNNING processes** - When looking for a new process to run on the CPU, the kernel has to consider only the runnable processes.
- The process descriptors include the **next_run** and **prev_run** fields to implement the **runqueue** list.
- The **init_task** process descriptor plays the role of list header.
- The **nr_running** variable stores the total number of runnable processes.
- The **add_to_runqueue( )** function inserts a process descriptor at the beginning of the list.
- The **del_from_runqueue( )** removes a process descriptor from the list.
- For scheduling purposes, two functions, **move_first_runqueue( )** and **move_last_runqueue( )**, are provided to move a process descriptor to the beginning or the end of the runqueue, respectively.
- Finally, the **wake_up_process( )** function is used to make a process runnable. It sets the process state to **TASK_RUNNING**, invokes **add_to_runqueue( )** to insert the process in the runqueue list, and increments **nr_running**.
- It also forces the invocation of the scheduler when the process is either real-time or has a dynamic priority much larger than that of the current process.

- **The pidhash table and chained list** -
- The kernel must be able to derive the process descriptor pointer corresponding to a PID. For example, the **kill()** system call, i.e., **kill(pid2)**.
- The kernel derives the process descriptor pointer from the PID and then extracts the pointer to the data structure that records the pending signals from P2's process descriptor.
- Scanning the process list sequentially and checking the pid fields of the process descriptors would be feasible but rather inefficient.
- In order to speed up the search, a pidhash hash table consisting of **PIDHASH_SZ** elements.
- The table entries contain process descriptor pointers. The PID is transformed into a table index using the pid_hashfn macro:

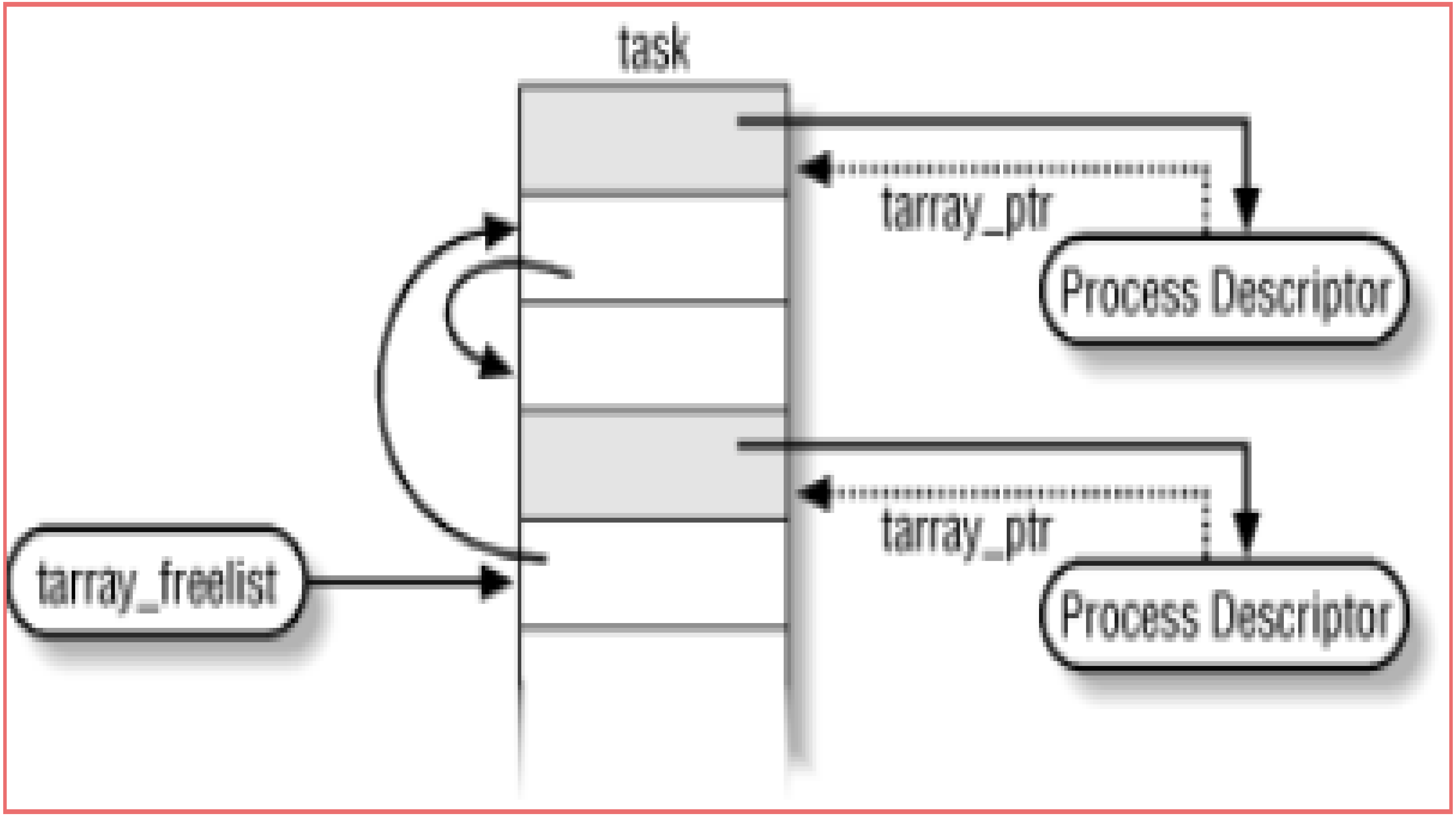
```
#define pid_hashfn(x) \
((((x) >> 8) ^ (x)) & (PIDHASH_SZ - 1))
```

- 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 a doubly linked list of colliding process descriptors.
- These lists are implemented by means of the **pidhash_next** and **pidhash_pprev** fields in the process descriptor.


pidhash
0
100
123
127
PID 228
PID 27535
PID 27536
pidhash_next
pidhash_pprev

- Hashing with chaining is preferable to a linear transformation from PIDs to table indexes, because a PID can assume any value between and 32767.
- Since NR_TASKS, the maximum number of processes, is usually set to 512, it would be a waste of storage to define a table consisting of 32768 entries.
- The **hash_ pid( )** and **unhash_ pid( )** functions are invoked to insert and remove process in the pidhash table, respectively.
- The **find_task_by_pid( )** function searches the hash table and returns the process descriptor pointer of the process with a given PID.

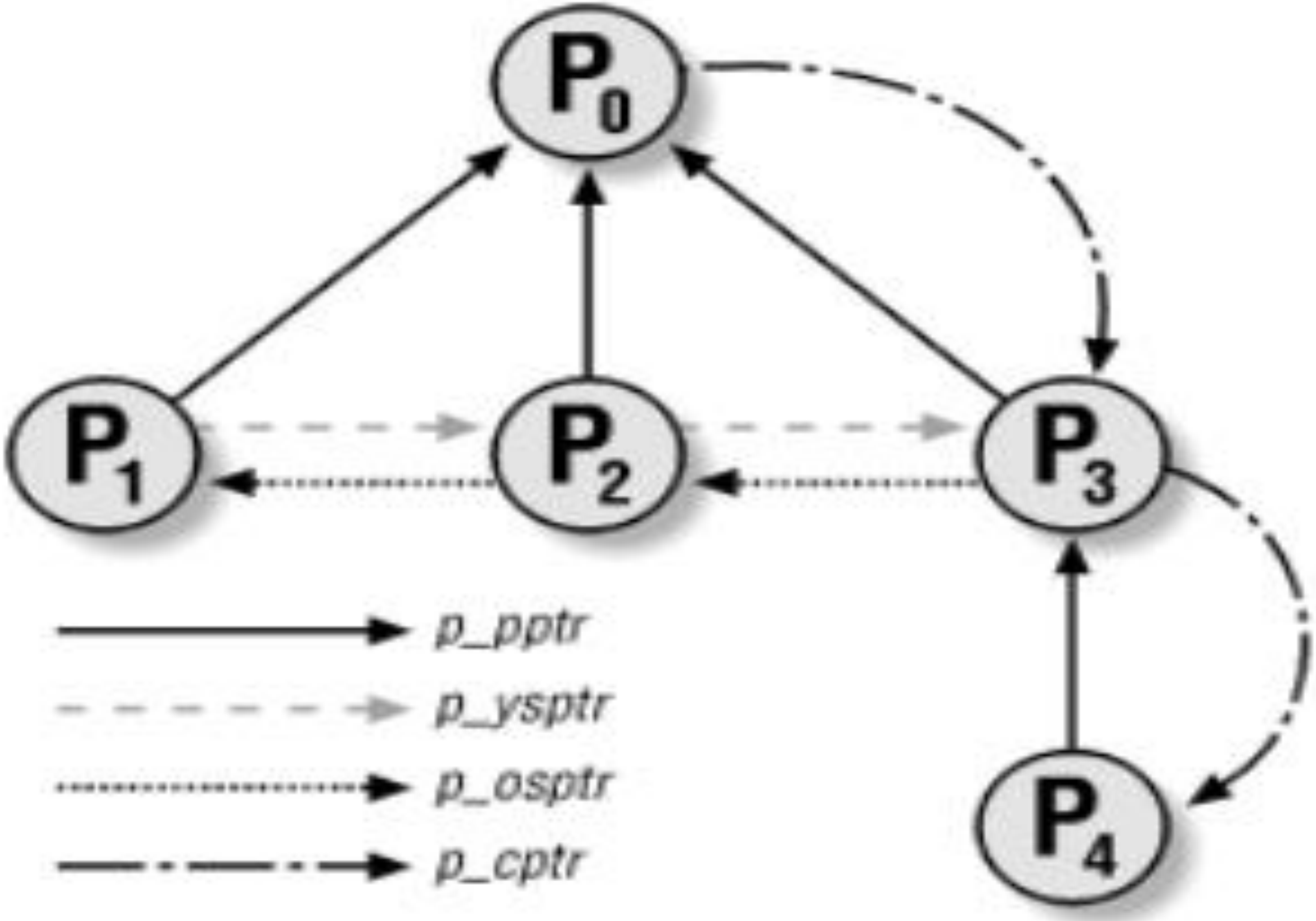
- **The list of task free entries** - The task array must be updated every time a process is created or destroyed.
- A list is used here to speed additions and deletions. Adding a new entry into the array is done efficiently: instead of searching the array linearly and looking for the first free entry, the kernel maintains a separate doubly linked, noncircular list of free entries.
- The **tarray_freelist** variable contains the first element of that list; each free entry in the array points to another free entry, while the last element of the list contains a null pointer.
- When a process is destroyed, the corresponding element in task is added to the head of the list.
- If the first element is counted as 0, the **tarray_freelist** variable points to element 4 because it is the last freed element.
- Previously, the processes corresponding to elements 2 and 1 were destroyed, in that order.
- Element 2 points to another free element of tasks.


task
tarray_ptr
Process Descriptor
tarray_ptr
Process Descriptor
tarray_freelist

- Deleting an entry from the array is also done efficiently. Each process descriptor p includes a **tarray_ ptr** field that points to the task entry containing the pointer to p.
- The **get_free_taskslot( )** and **add_free_taskslot( )** functions are used to get a free entry and to free an entry, respectively.

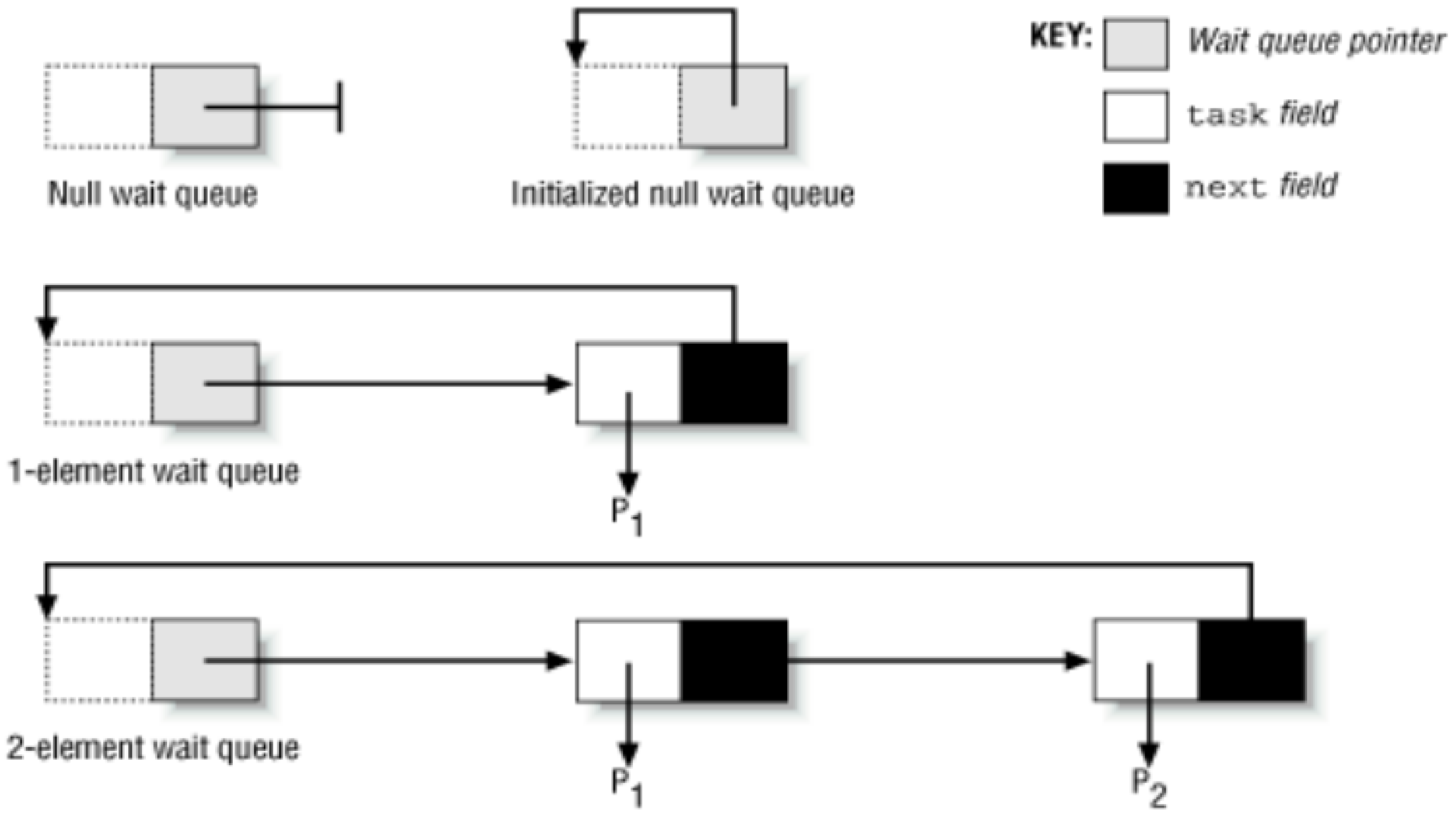
Parenthood Relationship among processes

- Processes created by a program have a parent/child relationship. Since a process can create several children, these have sibling relationships.
- **p_opptr (original 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. Thus, when a shell user starts a background process and exits the shell, the background process becomes the child of init.
- **p_pptr (parent)** - Points to the current parent of P; its value usually coincides with that of **p_opptr**. It may occasionally differ, such as when another process issues a **ptrace( )** system call requesting that it be allowed to monitor P.
- **p_cptr (child)** - Points to the process descriptor of the youngest child of P, that is, of the process created most recently by it.
- **p_ysptr (younger sibling)** - Points to the process descriptor of the process that has been created immediately after P by P's current parent.
- **p_osptr (older sibling)** - Points to the process descriptor of the process that has been created immediately before P by P's current parent.


P0
P1
P2
P3
P4
p_pptr
p_ysptr
p_osptr
p_cptr

Wait Queues

- The runqueue list groups together 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 following choices:
  - Processes in a TASK_STOPPED or in a TASK_ZOMBIE state are not linked in specific lists. There is no need to group them, because either the process PID or the process parenthood relationships may be used by the parent process to retrieve the child process.
  - 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 additional lists are called wait queues.
- Wait queues have several uses in the kernel, like interrupt handling, process synchronization, and timing.
- Wait queues implement conditional waits on events. A wait queue represents a set of sleeping processes, which are awakened by the kernel when some condition becomes true.
- Wait queues are implemented as cyclical lists whose elements include pointers to process descriptors.


KEY:
Wait queue pointer
task field
next field
Null wait queue
Initialized null wait queue
1-element wait queue
P1
2-element wait queue
P1
P2

- Each wait queue is identified by a wait queue pointer, which contains either the address of the first element of the list or the null pointer if the list is empty.
- The next field of the wait_queue data structure points to the next element in the list, except for the last element, whose next field points to a dummy list element.
- The dummy's next field contains the address of the variable or field that identifies the wait queue minus the size of a pointer (on Intel platforms, the size of the pointer is 4 bytes).
- Thus, the wait queue list can be considered by kernel functions as a truly circular list, since the last element points to the dummy wait queue structure whose next field coincides with the wait queue pointer.
- The **init_waitqueue( )** function initializes an empty wait queue; it receives the address q of a wait queue pointer as its parameter and sets that pointer to q - 4.
- The **add_wait_queue(q,entry)** function inserts a new element with address entry in the wait queue identified by the wait queue pointer q.

- Since the wait queue pointer is set to entry, the new element is placed in the first position of the wait queue list.
- If the wait queue was not empty, the next field of the new element is set to the address of the previous first element.
- Otherwise, the next field is set to the address of the wait queue pointer minus 4, and thus points to the dummy element.
- The **remove_wait_queue( )** function removes the element pointed to by entry from a wait queue.

```
struct wait_queue {
    struct task_struct * task;
    struct wait_queue * next;
    };
    if (*q != NULL)
        entry->next = *q;
    else
        entry->next = (struct wait_queue *)(q-1);
    *q = entry;
    next = entry->next;
    head = next;
        while ((tmp = head->next) != entry)
     head = tmp;
    head->next = next;
```

- A process wishing to wait for a specific condition can invoke any of the following functions:
    - The **sleep_on( )** function operates on the current process, which we'll call P:
    - The function sets P's state to TASK_UNINTERRUPTIBLE and inserts P into the wait queue whose pointer was specified as the parameter. Then it invokes the scheduler, which resumes the execution of another process. When P is awakened, the scheduler resumes execution of the **sleep_on( )** function, which removes P from the wait queue.
    - The **interruptible_sleep_on( )** function is identical to **sleep_on( )**, except that it sets the state of the current process P to TASK_INTERRUPTIBLE instead of TASK_UNINTERRUPTIBLE so that P can also be awakened by receiving a signal.
    - The **sleep_on_timeout( )** & **interruptible_sleep_on_timeout( )** functions are similar to the previous ones, but they also allow the caller to define a time interval after which the process will be woken up by the kernel. In order to do this, they invoke the **schedule_timeout( )** function instead of **schedule( )**.

```
void sleep_on(struct wait_queue **p)
    {
        struct wait_queue wait;
        current->state = TASK_UNINTERRUPTIBLE;
        wait.task = current;
        add_wait_queue(p, &wait);
        schedule( );
        remove_wait_queue(p, &wait);
    }
        if (q && (next = *q)) {
            head = (struct wait_queue *)(q-1);
         while (next != head) {
            p = next->task;
           next = next->next;
        if (p->state & mode)
            wake_up_process(p);
    }  }
```

- Processes inserted in a wait queue enter the TASK_RUNNING state by using either the **wake_up** or the **wake_up_interruptible** macros. Both macros use the **__wake_up( )** function, which receives as parameters the address q of the wait queue pointer and a bitmask mode specifying one or more states. Processes in the specified states will be woken up; others will be left unchanged.
- The function checks the state **p->state** of each process against mode to determine whether the caller wants the process woken up. Only those processes whose state is included in the mode bitmask are actually awakened.
- The wake_up macro specifies both the TASK_INTERRUPTIBLE and the TASK_UNINTERRUPTIBLE flags in mode, so it wakes up all sleeping processes. Conversely, the **wake_up_interruptible** macro wakes up only the TASK_INTERRUPTIBLE processes by specifying only that flag in mode

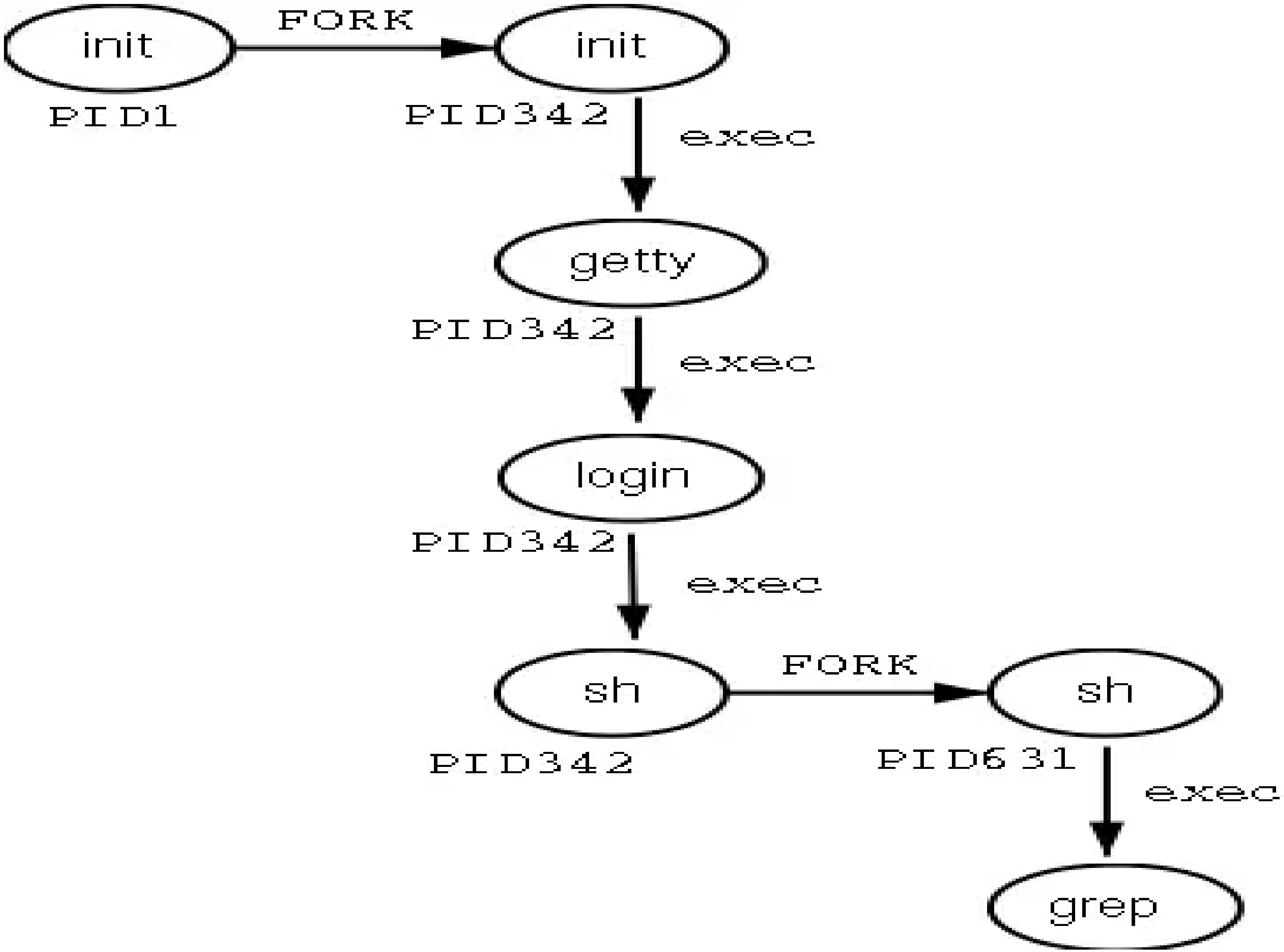
Creating Processes

- Unix operating systems rely heavily on process creation to satisfy user requests.
- As an example, the shell process 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, and a copy is granted to the child process.
- This approach makes process creation very slow and inefficient, since it requires copying the entire address space of the parent process.
- The child process rarely needs to read or modify all the resources already owned by the parent.
- 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, like the paging tables (and therefore the entire User Mode address space) and the open file tables.
- 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.

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

- A new process is created because an existing process makes an exact copy of itself. This child process has the same environment as its parent, only the process ID number is different. This procedure is called forking.
- After the forking process, the address space of the child process is overwritten with the new process data. This is done through an exec call to the system.
- The fork-and-exec mechanism thus switches an old command with a new, while the environment in which the new program is executed remains the same, including configuration of input and output devices, environment variables and priority.
- This mechanism is used to create all UNIX processes, so it also applies to the Linux operating system. Even the first process, init, with process ID 1, is forked during the boot procedure in the so-called bootstrapping procedure.


init
FORK
init
PID1
PID342
exec
getty
PID342
exec
login
PID342
exec
sh
FORK
sh
PID342
PID631
exec
grep

```
    child_pid = fork ();
         if (child_pid < 0) {
        printf("fork failed");
        return 1;
     } else if (child_pid == 0) {
        printf ("child process successfully created");
printf ("child_PID = %d,parent_PID = %d",getpid(), getppid( ) );
    } else {
        wait(NULL);
    printf ("parent process successfully created!");
printf ("child_PID = %d, parent_PID = %d", getpid( ), getppid( ) );
}
```

- Lightweight processes are created in Linux by using a function named **__clone( )**, which makes use of four 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 - Pointer to data passed to the fn( ) function.
    - flags - Miscellaneous information. 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 3 bytes encode a group of clone flags, which specify the resources shared between the parent and the child process.
- The flags, when set, have the following meanings:
- CLONE_VM - The memory descriptor and all page tables.
- CLONE_FS - The table that identifies the root directory and the current working directory.
- CLONE_FILES - The table that identifies the open files.
- CLONE_SIGHAND - The table that identifies the signal handlers

- CLONE_PID - The PID.
- CLONE_PTRACE - If a ptrace( ) system call is causing the parent process to be traced, the child will also be traced.
- CLONE_VFORK - Used for the vfork( ) system call (see later in this section).
- child_stack - Specifies the User Mode stack pointer to be assigned to the esp register of the child process. If it is equal to 0, the kernel assigns to the child the current parent stack pointer. Thus, 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. However, this parameter must have a non-null value if the child process shares the same address space as the parent.
- The **clone( )** system call receives only the flags and child_stack parameters; the new process always starts its execution from the instruction following the system call invocation.
- When the system call returns to the __**clone( )** function, it determines whether it is in the parent or the child and forces the child to execute the **fn( )** function.

- The old **vfork( )** system call, as a **clone( )** whose first parameter specifies a SIGCHLD signal and the flags CLONE_VM and CLONE_VFORK and whose second parameter is equal to 0.
- When either a **clone( ), fork( ), or vfork( )** system call is issued, the kernel invokes the **do_fork( )** function, which executes the following steps:
- If the **CLONE_PID** flag has been specified, the **do_fork( )** function checks whether the PID of the parent process is not null; if so, it returns an error code. Only the swapper process is allowed to set **CLONE_PID**; this is required when initializing a multiprocessor system.
- The alloc_task_struct( ) function is invoked in order to get a new 8 KB union task_union memory area to store the process descriptor and the Kernel Mode stack of the new process.
- The function follows the current pointer to obtain the parent process descriptor and copies it into the new process descriptor in the memory area just allocated.

- A few checks occur to make sure the user has the resources necessary to start a new process. The function gets the current number of processes owned by the user from a per-user data structure named **user_struct**. This data structure can be found through a pointer in the user field of the process descriptor.
- The **find_empty_process( )** function is invoked. If the owner of the parent process is not the superuser, this function checks whether **nr_tasks** (the total number of processes in the system) is smaller than **NR_TASKS-MIN_TASKS_LEFT_FOR_ROOT.** If so, **find_empty_process( )** invokes **get_free_taskslot( )** to find a free entry in the task array. Otherwise, it returns an error.
- The function writes the new process descriptor pointer into the previously obtained task entry and sets the **tarray_ptr** field of the process descriptor to the address of that entry.
- If the parent process makes use of some kernel modules, the function increments the corresponding reference counters. Each kernel module has its own reference counter, which indicates how many processes are using it. A module cannot be removed unless its reference counter is null.

- The function then updates some of the flags included in the flags field that have been copied from the parent process:

a. It clears the PF_SUPERPRIV flag, which indicates whether the process has used any of its superuser privileges.

b. It clears the PF_USEDFPU flag.

c. It clears the PF_PTRACED flag unless the CLONE_PTRACE parameter flag is set. When set, the CLONE_PTRACE flag means that the parent process is being traced with the **ptrace( )** function, so the child should be traced too.

d. It clears PF_TRACESYS flag unless, once again, the CLONE_PTRACE parameter flag is set.

e. It sets the PF_FORKNOEXEC flag, which indicates that the child process has not yet issued an **execve( )** system call.

f. It sets the PF_VFORK flag according to the value of the CLONE_VFORK flag. This specifies that the parent process must be woken up whenever the process (the child) issues an **execve()** system call or terminates.

- Now the function has taken almost everything that it can use from the parent process; the rest of its activities focus on setting up new resources in the child and letting the kernel know that this new process has been born. First, the function invokes the get_pid( ) function to obtain a new PID, which will be assigned to the child process (unless the CLONE_PID flag is set).
- The function then updates all the process descriptor fields that cannot be inherited from the parent process, such as the fields that specify the process parenthood relationships.
- Unless specified differently by the flags parameter, it invokes **copy_files( ), copy_fs( )**, **copy_sighand( )**, and **copy_mm( )** to create new data structures and copy into them the values of the corresponding parent process data structures.
- It invokes **copy_thread( )** to initialize the Kernel Mode stack of the child process with the values contained in the CPU registers when the **clone( )** call was issued. The function forces the value into the field corresponding to the eax register. The tss.esp field of the TSS of the child process is initialized with the base address of the Kernel Mode stack, and the address of an Assembly language function (ret_from_fork( )) is stored in the tss.eip field.
- It uses the SET_LINKS macro to insert the new process descriptor in the process list.
- It uses the **hash_pid( )** function to insert the new process descriptor in the pidhashhash table.
- It increments the values of nr_tasks and current->user->count.
- It sets the state field of the child process descriptor to TASK_RUNNING and then invokes **wake_up_process( )** to insert the child in the runqueue list.

- If the CLONE_VFORK flag has been specified, the function suspends the parent process until the child releases its memory address space. In order to do this, the process descriptor includes a kernel semaphore called **vfork_sem.**
- It returns the PID of the child, which will be eventually be read by the parent process in User Mode.

Kernel Threads

- Some of the system processes runs only in Kernel mode, therefore, modern operating systems delegate their functions to kernel threads.
- In Linux, kernel threads differ from regular processes in the following ways:
- Each kernel thread executes a single specific kernel function, while regular processes execute kernel functions only through system calls.
- Kernel threads run only in Kernel Mode, while regular processes run alternatively in Kernel Mode and in User Mode.
- Since kernel threads run only in Kernel Mode, they use only linear addresses greater than PAGE_OFFSET. Regular processes, on the other hand, use all 4 gigabytes of linear addresses, either in User Mode or in Kernel Mode.

Creating a Kernel thread

- The **kernel_thread( )** function creates a new kernel thread and can be executed only by another kernel thread.

```
int kernel_thread(int (*fn)(void *), void * arg, unsigned long flags)
{
         pid_t p;
        p = clone( 0, flags | CLONE_VM );
        if ( p ) /* parent */
                return p;
        else { /* child */
                fn(arg);
                exit( );
                }
}
```

Process 0

- The ancestor of all processes, called process 0 or the swapper process, is a kernel thread created from scratch during the initialization phase of Linux by the **start_kernel()** function.
- These Process 0 makes use of following data structures.
  - A process descriptor and a Kernel Mode stack stored in the init_task_union variable. The init_task and init_stack macros yield the addresses of the process descriptor and the stack, respectively.
  - The following tables, which the process descriptor points to:
    - init_mm
    - init_mmap
    - init_fs
    - init_files
    - init_signals
  - The tables are initialized, respectively, by the following macros:
    - INIT_MM
    - INIT_MMAP
    - INIT_FS
    - INIT_FILES
    - INIT_SIGNALS

- A TSS segment, initialized by the INIT_TSS macro.
- Two Segment Descriptors, namely a TSSD and an LDTD, which are stored in the GDT.
- A Page Global Directory stored in swapper_pg_dir, which may be considered as the kernel Page Global Directory since it is used by all kernel threads.
- The **start_kernel( )** function initializes all the data structures needed by the kernel, enables interrupts, and creates another kernel thread, named process 1, more commonly referred to as the init process :

kernel_thread(init, NULL, CLONE_FS | CLONE_FILES | CLONE_SIGHAND);

- The newly created kernel thread has PID 1 and shares with process all per-process kernel data structures. Moreover, when selected from the scheduler, the init process starts executing the **init( )** function.
- After having created the init process, process executes the **cpu_idle( )** function, which essentially consists of repeatedly executing the hlt Assembly language instruction with the interrupts enabled.
- Process is selected by the scheduler only when there are no other processes in the TASK_RUNNING state.

Process 1

- The kernel thread created by process executes the **init( )** function, which in turn invokes the **kernel_thread( )** function four times to initiate four kernel threads needed for routine kernel tasks:

```
kernel_thread(bdflush, NULL, CLONE_FS | CLONE_FILES | CLONE_SIGHAND);

kernel_thread(kupdate, NULL, CLONE_FS | CLONE_FILES | CLONE_SIGHAND);

kernel_thread(kpiod, NULL,  CLONE_FS | CLONE_FILES | CLONE_SIGHAND);

kernel_thread(kswapd, NULL, CLONE_FS | CLONE_FILES | CLONE_SIGHAND);
```

- As a result, four additional kernel threads are created to handle the memory cache and the swapping activity:
  - kflushd (also bdflush) - Flushes "dirty" buffers to disk to reclaim memory
  - kupdate - Flushes old "dirty" buffers to disk to reduce risks of filesystem inconsistencies.
  - kpiod - Swaps out pages belonging to shared memory mappings.
  - kswapd - Performs memory reclaiming
- Then init( ) invokes the execve( ) system call to load the executable program init.
- As a result, the init kernel thread becomes a regular process having its own per-process kernel data structure.
- The init process never terminates, since it creates and monitors the activity of all the processes that implement the outer layers of the operating system.

Destroying Processes

- The usual way for a process to terminate is to invoke the exit( ) system call.
- This system call may be inserted by the programmer explicitly.
- Additionally, the exit( ) system call is always executed when the control flow reaches the last statement of the main procedure.
- Alternatively, the kernel may force a process to die.
- This typically occurs when the process has received a signal that it cannot handle or ignore or when an unrecoverable CPU exception has been raised in Kernel Mode while the kernel was running on behalf of the process.

Process Termination

- All process terminations are handled by the do_exit( ) function, which removes most references to the terminating process from kernel data structures. The do_exit( ) function executes the following actions:

1. Sets the PF_EXITING flag in the flag field of the process descriptor to denote that the process is being eliminated.

2. Removes, if necessary, the process descriptor from an IPC semaphore queue via the **sem_exit( )** function or from a dynamic timer queue via the **del_timer( )** function

3. Examines the process's data structures related to paging, filesystem, open file descriptors, and signal handling, respectively, with the **__exit_mm( ), __exit_files( ), __exit_fs( ), and _ _exit_sighand( )** functions. These functions also remove any of these data structures if no other process is sharing it.

4. Sets the state field of the process descriptor to TASK_ZOMBIE. We shall see what happens to zombie processes in the following section.

5. Sets the exit_code field of the process descriptor to the process termination code. This value is either the **exit( )** system call parameter or an error code supplied by the kernel.

6. Invokes the **exit_notify( )** function to update the parenthood relationships of both the parent process and the children processes. All children processes created by the terminating process become children of the init process.

7. Invokes the **schedule( )** function to select a new process to run.Since a process in a TASK_ZOMBIE state is ignored by the scheduler, the process will stop executing right after the switch_to macro in **schedule( )** is invoked.