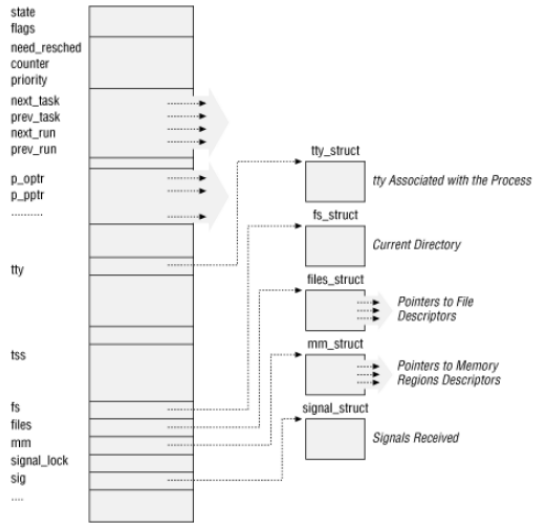
PROCESSES CHAPTER 3

Introduction:
- A process is an instance of a program in execution.
- In Linux source code, processes are often referred to as "tasks."

Process Descriptor:
- The role of a process descriptor is to manage how the kernel handles each process.
- It includes information such as process priority, CPU execution status, address space, file permissions, and more.
- This is represented by a complex structure called `task_struct`, associated with every existing process in the system.
- The process descriptor contains numerous fields, some of which are pointers to other data structures.



Process State:
- The process state field in the process descriptor describes the current status of the process.
- It consists of an array of flags, with only one flag set at a time, representing the process's state.
- The possible process states include:
  1. **TASK_RUNNING:** The process is executing on the CPU or waiting to be executed.
  2. **TASK_INTERRUPTIBLE:** The process is sleeping and can be woken up by certain conditions like hardware interrupts or resource availability.
  3. **TASK_UNINTERRUPTIBLE:** Similar to TASK_INTERRUPTIBLE but cannot be woken up by signals.
  4. **TASK_STOPPED:** Process execution has been halted, often due to signals like SIGSTOP or SIGTSTP.
  5. **TASK_ZOMBIE:** The process has terminated, but the parent process hasn't yet collected its termination status.

Identifying a Process (Process ID or PID):

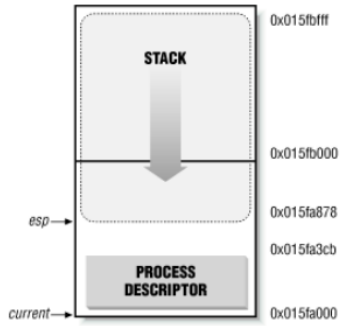
- In Linux, each process is given a unique number called a "Process ID" or PID.
- PIDs are like numbers for identifying processes. They start from 1 and go up.
- The highest possible PID on Linux is 32,767.
- When the system creates a new process, it gives it the next available PID.

Task Array:

- To manage all these processes, Linux has a big list called the "task array."
- Think of it as a list of process information, where each process gets a spot.
- If a spot in the list is empty (like a null), it means there's no process there.

Storing a Process Descriptor:

- In Linux, information about each process is stored in something called a "process descriptor."
- Instead of storing these descriptors in a fixed place, Linux keeps them in dynamic memory.
- Each process has its own process descriptor, and it's not fixed in memory.
- Linux stores two things for each process: the process descriptor and a special stack for Kernel Mode.
- Processes have their own stacks for different modes (Kernel Mode and User Mode).
- The Kernel Mode stack is smaller because it's used less frequently.



Process Stack and Descriptor:

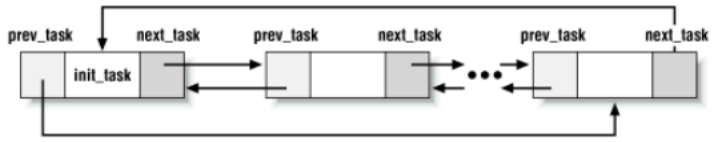
- The CPU uses the `esp` register to keep track of the stack, which is like a temporary memory.
- On Intel systems, the stack starts at the end of memory and goes backward.
- When a process switches from User Mode to Kernel Mode, its Kernel Mode stack is initially empty.
- The `esp` value decreases as data is added to the stack.
- It can grow to accommodate the process descriptor, which is a small chunk of information.
- The `current` thing helps the kernel quickly find a process's information.

Process Descriptor Cache:

- There's a cache called `EXTRA_TASK_STRUCT` that stores process descriptors to save memory.
- This cache avoids creating and deleting memory areas repeatedly when processes are created and removed quickly.
- When memory is not full, `free_task_struct()` puts it in the cache.
- When memory is needed, `alloc_task_struct()` takes it from the cache if it's at least half full or there's no consecutive memory available.

Process Lists:
- The kernel keeps lists of processes for efficient organization.
- Each list has pointers to process info, and each process info has pointers to the previous and next process in the list.
- There's a main list called the "process list," and it's like a circle because the last process points to the first.

- The first process in this list is the "init_task," which is like the ancestor of all processes.
- Special tools like `SET_LINKS` and `REMOVE_LINKS` help add or remove processes from this list while keeping track of their relationships.
- There's a handy `for_each_task` tool that helps the kernel go through this list quickly.

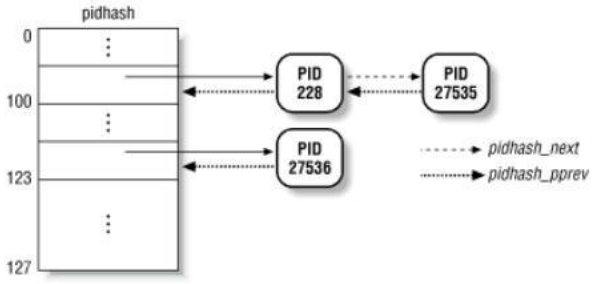


Managing Runnable Processes:

- Kernel selects processes in `TASK_RUNNING` state to run on the CPU.
- Each process has `next_run` and `prev_run` fields for order.
- `init_task` is at the front; `nr_running` tracks waiting processes.
- Functions like `add_to_runqueue()` add to the front, and `del_from_runqueue()` remove.
- Scheduling functions like `move_first_runqueue()` and `move_last_runqueue()` reorder.
- `wake_up_process()` makes a process ready, increments count, and alerts the scheduler if important.

PID Hash Table and Chained List:

- Linux uses a PID hash table for fast PID-based process lookup.
- A hash function converts PIDs to table indexes.
- Collisions (multiple PIDs at one index) are resolved with chained doubly-linked lists.
- Process descriptors use `pidhash_next` and `pidhash_pprev` for list maintenance. chain.



Hashing with Chaining:

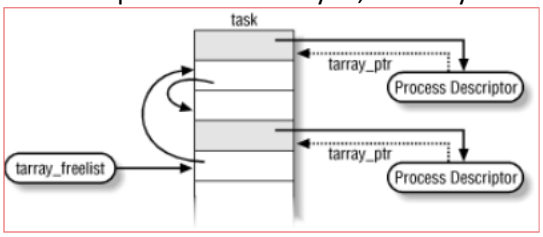
- Chaining is preferred for PID hashing due to PID's wide range (0-32767).
- `NR_TASKS` (max processes) is typically smaller, e.g., 512.
- Avoids inefficiently large tables (32768 entries).

PID Hash Functions:

- `hash_pid()` and `unhash_pid()` insert and remove processes from the PID hash table.
- `find_task_by_pid()` searches the table for a PID and returns the process descriptor.

List of Task Free Entries:

- `task_array` updates with process creation and destruction.
- Maintains a doubly-linked list of free entries for efficient handling.
- `tarray_freelist` points to the first free element.
- Each free entry points to another, and the last has a null pointer.
- When a process is destroyed, its entry is added to the head of the list.

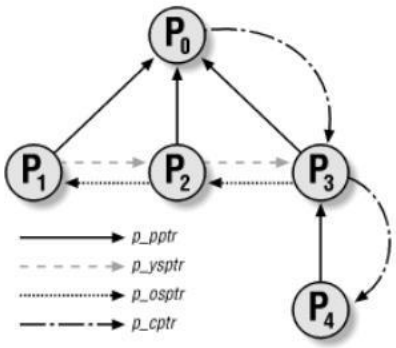


Efficient Entry Deletion:
- To delete an entry from the array efficiently, each process descriptor includes an `array_ptr` field.
- This field points to the task entry that contains the pointer to the process descriptor.
- When you need to remove an entry, you can directly access and manipulate the entry through the `array_ptr` field.

Getting and Adding Free Task Slots:
- The `get_free_taskslot()` function is used to obtain a free entry in the array of task descriptors efficiently.
- The `add_free_taskslot()` function is used to release and free an entry, making it available for future use.

In process relationships:

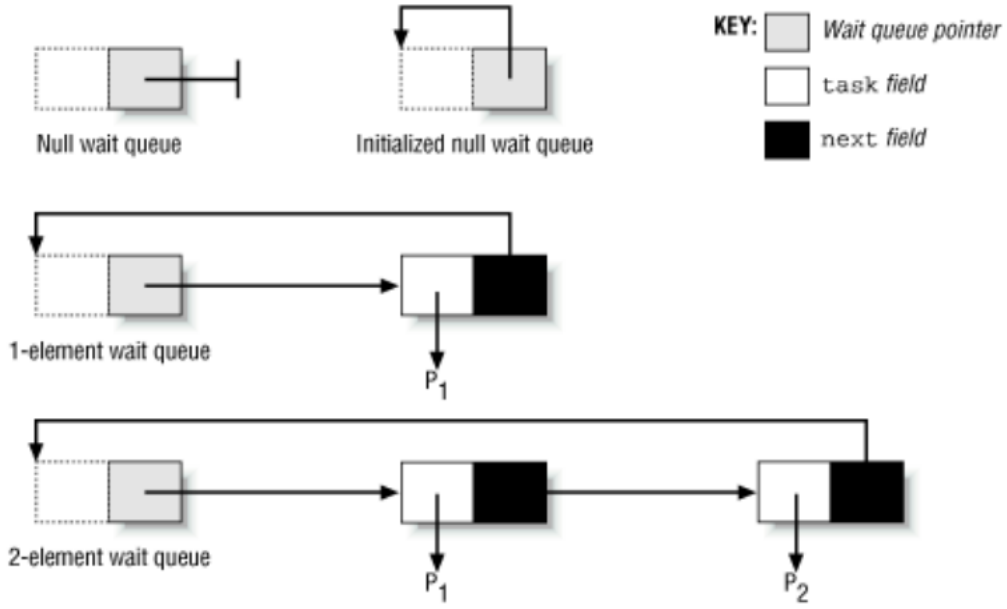
- Processes have parent-child and sibling relationships.
- `p_opptr` points to the initial parent or `init` if the original parent is gone.
- `p_pptr` points to the current parent, which may differ in some cases.
- `p_cptr` points to the youngest child created by the process.
- `p_ysptr` points to the next younger sibling.
- `p_osptr` points to the previous older sibling.



Wait Queues in process management:

- Wait queues group processes in `TASK_RUNNING` state.
- Different approaches are used for processes in other states.
- `TASK_STOPPED` and `TASK_ZOMBIE` processes are not in specific lists; PID or relationships can retrieve them.

- `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE` processes are categorized into classes based on events.
- Additional lists, called wait queues, are used to manage such processes.
- Wait queues are vital for interrupt handling, synchronization, and timing.
- They facilitate conditional waits on events, awakening sleeping processes when conditions change.
- Wait queues are circular lists with pointers to process descriptors.



Sleeping and Waking Up Processes:
- `sleep_on(q)` puts the current process to sleep (`TASK_UNINTERRUPTIBLE`) and adds it to queue `q`.
- `interruptible_sleep_on(q)` does the same but allows waking up by a signal (`TASK_INTERRUPTIBLE`).
- `sleep_on_timeout(q, timeout)` and `interruptible_sleep_on_timeout(q, timeout)` wake up after a timeout.
- To awaken processes in the queue, use `wake_up(q)` or `wake_up_interruptible(q)`.

Creating Processes in Unix:

Unix operating systems frequently create processes to fulfill user requests. For instance, when a user enters a command, the shell process creates a new process to execute it.

In traditional Unix systems, all processes are treated the same way: resources of the parent process are duplicated for the child process. However, this method is slow and inefficient because it involves copying the entire parent process's address space, even though the child often doesn't need all of it.

Modern Unix kernels address this issue with three mechanisms:

The Copy On Write technique lets the parent and child read the same pages. When either tries to write, the kernel copies to a new page for that process.

Lightweight processes share per-process kernel structures, like paging tables and open file tables.

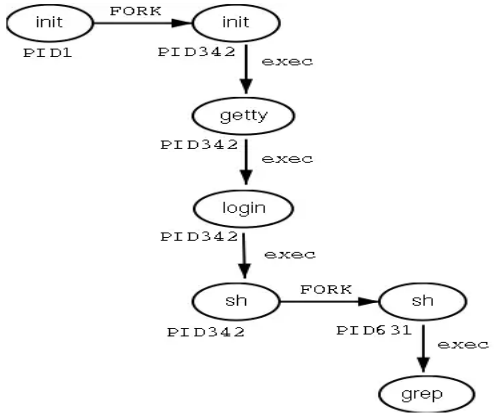
vfork() creates a process sharing memory with its parent. The parent waits until the child exits or runs a new program to prevent data conflicts.

Creating New Processes:
- Copying a running program creates a child.
- Same environment, different ID.

fork() System Call:
- Snapshots parent, creates identical child.
- Child can change its data.

vfork() System Call:
- Like `fork()` but parent-child sharing.
- Parent waits for child, faster for some tasks.

clone() System Call:
- More flexible than `fork()`.
- Customize copying, greater control.



Process Creation Concepts:

- `fork()`: Creates a new process by duplicating the existing one. The child inherits the environment but gets a different PID.
- PID: Unique numerical identifier for each process.
- `getpid()`: Retrieves the current process's PID.
- `getppid()`: Retrieves the parent process's PID.

Creating Processes in Linux:

- `fork()`: Copies parent to create a child process.
- `clone()`: Provides fine-grained control over process creation.

- Processes are fundamental for running programs and managing resources.

Kernel Threads:
- In modern operating systems like Linux, certain system processes exclusively run in Kernel Mode, and these tasks are assigned to "kernel threads."
- Kernel threads in Linux differ from regular processes in several ways:
  - They execute specific kernel functions.
  - They operate exclusively in Kernel Mode.
  - They use linear addresses greater than PAGE_OFFSET.
- To create a kernel thread in Linux, the `kernel_thread()` function is often used.

Process 0 (Swapper Process):
- Process 0, also known as the "swapper process," is the very first process and serves as the ancestor of all other processes.
- It is created from scratch during the initialization phase of the Linux operating system.
- Process 0 plays a pivotal role in system management and employs various data structures such as process descriptors, tables, and segments to oversee the system.
- The `start_kernel()` function is responsible for initializing these critical data structures, and it also creates another kernel thread known as "process 1" or the "init process."

Process 1 (Init Process):
- Process 1, established by Process 0 during system initialization, is commonly referred to as the "init process."
- The init process executes the `init()` function and assumes responsibility for initializing the entire system and monitoring other processes.
- In addition to its core functions, the init process creates additional kernel threads to manage tasks like memory caching and swapping activities.

These kernel threads, along with Process 0 and Process 1, form the foundation for system management and are integral to the proper functioning of the Linux operating system.

Destroying Processes:

- Processes can exit using `exit()` or forcibly terminated by the kernel in case of unrecoverable errors.

Process Termination:

- `do_exit()` manages process terminations with the following steps:
  1. Sets `PF_EXITING` flag to mark process as exiting.
  2. Removes from semaphore or timer queues if needed.
  3. Cleans up memory, file systems, file descriptors, signals.
  4. Sets process state to `TASK_ZOMBIE`.
  5. Records termination code in `exit_code`.
  6. Updates parent-child relationships using `exit_notify()`.
  7. Calls `schedule()` to choose a new process to run.

In summary, kernel threads are specialized processes running exclusively in Kernel Mode. Process 0 and Process 1 (init) are pivotal for system initialization and management. When a process exits, it undergoes a structured termination process, ensuring resources are cleaned up before becoming a "zombie" process.