Linux Versus Other Unix-Like Kernels

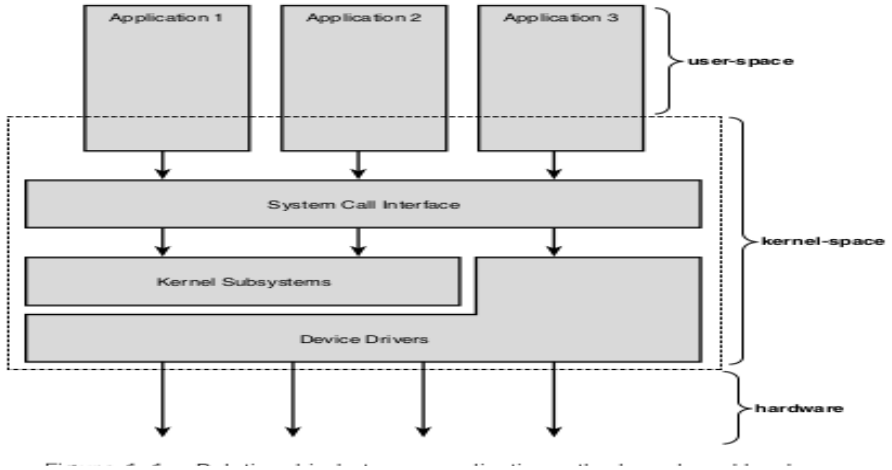
1. **Free and Open Source:** Linux is free, while some Unix-like systems may have costs.
2. **Customizable:** Linux is highly customizable due to its open-source nature.
3. **Runs on Older Hardware:** Linux works well on older, low-end hardware.
4. **Efficient:** Linux prioritizes efficiency and performance.
5. **Strong Community Support:** Linux has active community support for updates and solutions.

Multiuser Systems:
- Multiuser systems allow multiple users to run applications concurrently.
- Key features include user authentication, protection against buggy and malicious programs, resource allocation limits, and hardware protection.
- Unix is an example of a multiuser system that enforces hardware resource protection.

Users and Groups:
- In multiuser systems, each user has a private space with storage and mail.
- User access to private space must be restricted to the owner.
- Users are identified by a UserID (UID) and can belong to one or more groups identified by a (gid)
- The root or superuser account is used for system administration.

Processes:
- Processes are fundamental in operating systems and represent running programs.
- A process can be seen as an instance of a program in execution.
- Processes execute instructions in an address space, which is the memory they can access.
- Modern systems allow multiple execution flows within a single process.
- Concurrent active processes are known as multiprogramming or multiprocessing.

Kernel Architecture



Microkernel Operating Systems:
- Microkernel OSs have a small kernel with essential functions.
- System processes implement other OS functions like memory allocation and device drivers
- Easily portable to different hardware architectures.
- Efficient memory usage by swapping out or destroying unneeded system processes

Hard Links:
- A filename included in a directory is called a hard link, or simply a link.
- The same file can have multiple hard links, meaning it has several filenames.

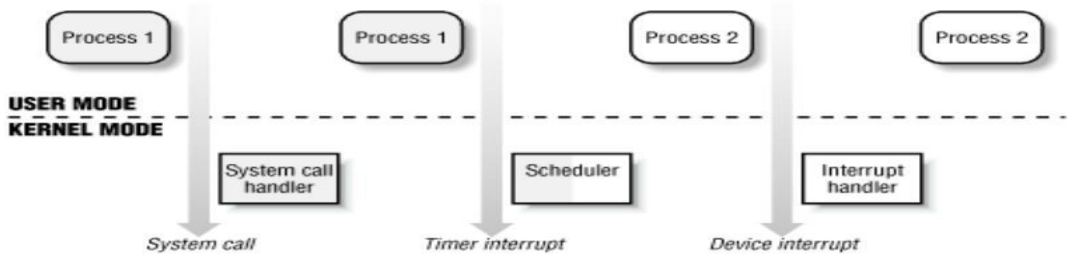
- You create a hard link using the `ln` command, like `$ ln f1 f2`.
- Hard links have two limitations: you can't create hard links for directories, and they must be within the same filesystem.

Soft Links (Symbolic Links):
- To overcome the limitations of hard links, symbolic links (soft links) were introduced.
- Symbolic links are short files that contain the pathname of another file.
- They can refer to files located in different filesystems.
- You create a symbolic link using the `ln -s` command, like `$ ln -s f1 f2`.

hard links are multiple names for the same file within the same filesystem, while symbolic links are shortcuts that can point to files in different filesystems.

- **File Descriptors:** These are numbers assigned to open files by the operating system. They help processes read, write, and manage file positions.
- **Inodes:** These are data structures used to store information about a file, like owner, permissions, and data block locations.
- **File Handling System Calls:** Processes open files using `open()`, read/write data with `read()` and `write()`, change file positions with `lseek()`, and close files with `close()`.

- **Process/Kernel Mode:** CPUs can run in either User Mode or Kernel Mode.
- User Mode: Programs cannot directly access kernel data or programs.
- Kernel Mode: Programs have full access to kernel resources.
- Special instructions allow switching between User and Kernel Modes.
- Most of the time, programs run in User Mode and switch to Kernel Mode only when they need a service from the kernel.



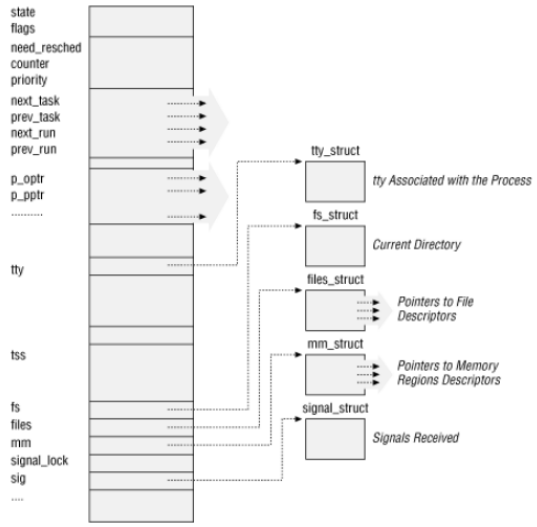
- **Processes:** Processes are dynamic entities managed by the kernel.
- The kernel manages processes but is not a process itself.
- Processes use system calls to request kernel services.
- System calls set up parameters and switch the CPU from User to Kernel Mode.
- Kernel threads are privileged, run in Kernel Mode, and handle various tasks like exceptions, interrupts, and I/O operations.

In summary, processes in User Mode request kernel services using system calls, which switch the CPU to Kernel Mode for servicing. Kernel threads run in Kernel Mode to handle various tasks in the system.

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):

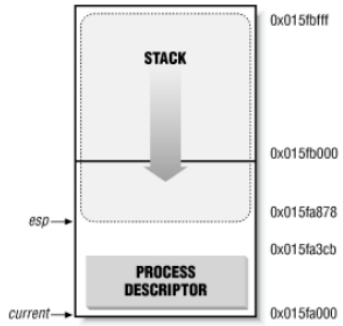
- In Linux, each process is given a unique number called a "Process ID" or PID.
- PIDs are like numbers for identifying processes. They start from 1 and go up.
- The highest possible PID on Linux is 32,767.
- When the system creates a new process, it gives it the next available PID.

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:

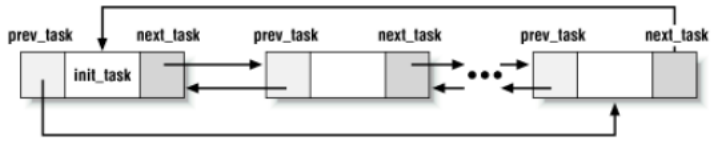
- The CPU uses the `esp` register to keep track of the stack, which is like a temporary memory.
- On Intel systems, the stack starts at the end of memory and goes backward.
- When a process switches from User Mode to Kernel Mode, its Kernel Mode stack is initially empty.
- The `esp` value decreases as 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.

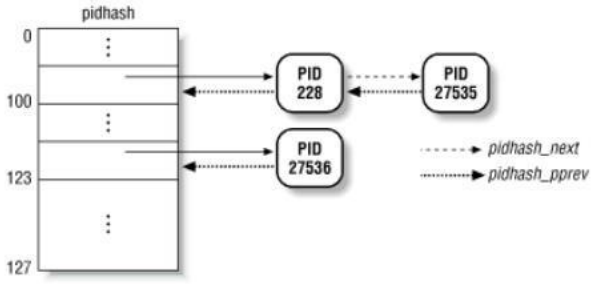


Managing Runnable Processes:

- 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:

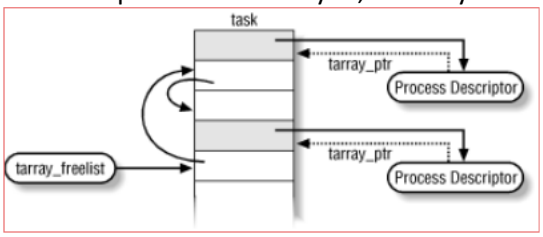
- Chaining is preferred for PID hashing due to PID's wide range (0-32767).
- `NR_TASKS` (max processes) is typically smaller, e.g., 512.
- Avoids inefficiently large tables (32768 entries).

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.

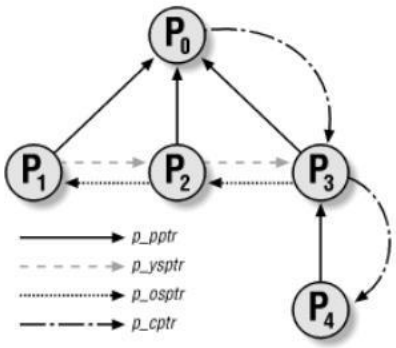


Efficient Entry Deletion:
- To delete an entry from the array efficiently, each process descriptor includes an `array_ptr` field.
- This field points to the task entry that contains the pointer to the process descriptor.
- When you need to remove an entry, you can directly access and manipulate the entry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

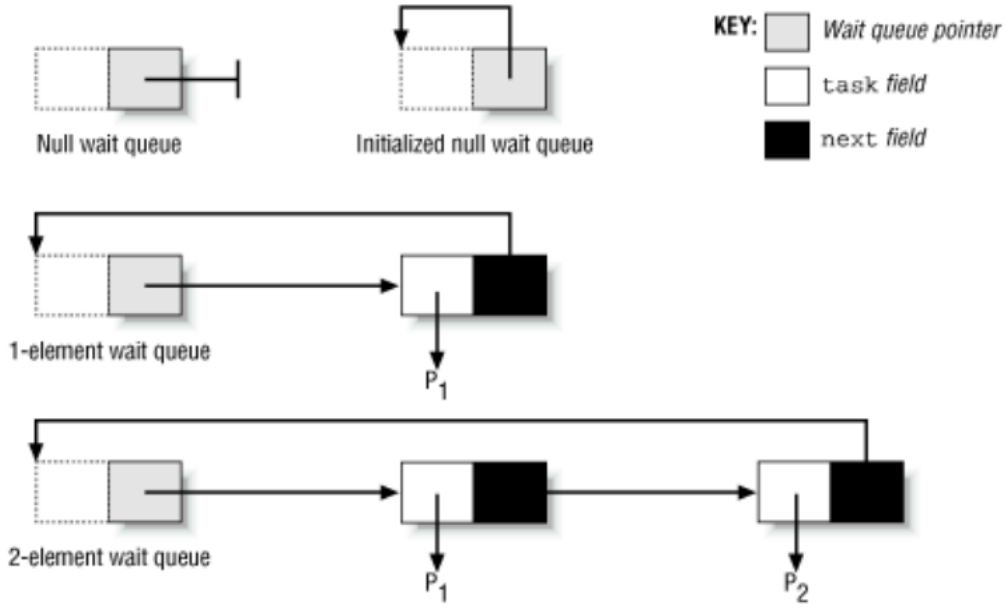
- Processes have parent-child and sibling relationships.
- `p_opptr` points to the initial parent or `init` if the original parent is gone.
- `p_pptr` points to the current parent, which may differ in some cases.
- `p_cptr` points to the youngest child created by the process.
- `p_ysptr` points to the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

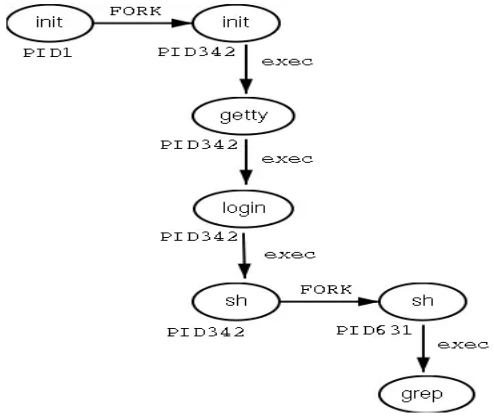
vfork() creates a process sharing memory with its parent. The parent waits until the child exits or runs a new program to prevent data conflicts.

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.

Kernal synchornization
- The kernel is like a server handling requests from processes and devices.
- These requests run concurrently and can create race conditions.
- This chapter covers synchronization techniques.
- It explores multiprocessor architecture and mutual exclusion in SMP Linux.

Kernel control path
- Kernel functions are executed in response to requests from User Mode processes or external devices via interrupts.
- These sequences of instructions in Kernel Mode are called "kernel control paths."
- Kernel control paths handle system calls, exceptions, or interrupts.
- Unlike processes, kernel control paths lack descriptors and are managed through instruction sequences.
- In some cases, the CPU runs a kernel control path sequentially, but it can interleave them during context switches or interrupts.
- Interleaving is crucial for multiprocessing and improves controller throughput.
- When interleaving, it's essential to protect critical data structures to prevent corruption.

Synchronization Techniques
- Race conditions and critical regions apply to kernel control paths as they do to processes.
- Race conditions arise when interleaved control paths affect the outcome of computations.
- Critical regions are sections of code that must be executed entirely by one control path before another can enter.
- Four synchronization techniques are discussed: nonpreemptability, atomic operations, interrupt disabling, and locking.

Nonpreemptability of Processes in Kernel Mode
- Linux's Kernel Mode is non-preemptive, meaning a running process in Kernel Mode won't be replaced by a higher-priority process unless it voluntarily gives up control.
- Interrupts or exceptions can interrupt a Kernel Mode process, but once the handler finishes, the original kernel control path resumes.
- Kernel control paths handling interrupts or exceptions can only be interrupted by other such control paths.
- Nonblocking system call control paths are atomic with respect to each other, simplifying kernel function implementations.
- When a process in Kernel Mode voluntarily yields the CPU, it must ensure data structures remain consistent.
- Upon resuming, the process must recheck previously accessed data structures that may have changed due to other control paths running the same code for different processes.

Atomic operations:
- Atomic operations are operations that execute in a single, uninterrupted assembly language instruction.
- They help prevent race conditions by ensuring that operations are completed without interruption at the chip level.
- Examples of atomic operations include instructions that make zero or one memory access.
- Read/modify/write instructions like "inc" or "dec" are atomic if no other processor accesses the memory between the read and write.
- Instructions prefixed with "lock" (0xf0) are atomic even on multiprocessor systems, as they lock the memory bus until the instruction is finished.

- Instructions prefixed with "rep" (0xf2, 0xf3) are not atomic because they check for interrupts between iterations.
- In the Linux kernel, special functions are used to ensure atomicity, with a lock byte added for multiprocessor systems.

Table 11-1. Atomic Operations in C

Function	Description
`atomic_read(v)`	Return `*v`
`atomic_set(v,i)`	Set `*v` to `i`.
`atomic_add(i,v)`	Add `i` to `*v`.
`atomic_sub(i,v)`	Subtract `i` from `*v`.
`atomic_inc(v)`	Add 1 to `*v`.
`atomic_dec(v)`	Subtract 1 from `*v`.
`atomic_dec_and_test(v)`	Subtract 1 from `*v` and return 1 if the result is non-null, otherwise.
`atomic_inc_and_test_greater_zero(v)`	Add 1 to `*v` and return 1 if the result is positive, otherwise.
`atomic_clear_mask(mask,addr)`	Clear all bits of `addr` specified by `mask`.
`atomic_set_mask(mask,addr)`	Set all bits of `addr` specified by `mask`.

Interrupt disabling:

- Critical sections in code require more than atomic operations and typically include an atomic operation as their core.
- Interrupt disabling is a mechanism to create critical sections in the kernel, allowing a control path to run without interruption from hardware interrupts.
- However, interrupt disabling alone doesn't prevent all forms of control path interleaving.
- Examples of scenarios where interrupt disabling may not be sufficient include "Page fault" exceptions or when the schedule() function is invoked.
- In the Linux kernel, interrupt disabling is achieved using assembly instructions like cli and sti, or via macros like __cli and __sti.
- To save and restore the state of the IF flag in the eflags register, the kernel uses macros like __save_flags and __restore_flags.
- These macros help maintain proper interrupt handling, ensuring that interrupts are only enabled if they were enabled before the critical section.
- Linux provides additional synchronization macros, especially important on multiprocessor systems, but they are somewhat redundant on uniprocessor systems.

Table 11-2. Interrupt Disabling/Enabling Macros on a Uniprocessor System	
Macro	**Description**
`spin_lock_init(lck)`	No operation
`spin_lock(lck)`	No operation
`spin_unlock(lck)`	No operation
`spin_unlock_wait(lck)`	No operation
`spin_trylock(lck)`	Return always 1
`spin_lock_irq(lck)`	`__cli( )`
`spin_unlock_irq(lck)`	`__sti( )`
`spin_lock_irqsave(lck, flags)`	`__save_flags(flags); __cli( )`
`spin_unlock_irqrestore(lck, flags)`	`__restore_flags(flags)`
`read_lock_irq(lck)`	`__cli( )`
`read_unlock_irq(lck)`	`__sti( )`
`read_lock_irqsave(lck, flags)`	`__save_flags(flags); __cli( )`
`read_unlock_irqrestore(lck, flags)`	`__restore_flags(flags)`
`write_lock_irq(lck)`	`__cli( )`
`write_unlock_irq(lck)`	`__sti( )`
`write_lock_irqsave(lck, flags)`	`__save_flags(flags); __cli( )`
`write_unlock_irqrestore(lck, flags)`	`__restore_flags(flags)`

- `add_wait_queue()` and `remove_wait_queue()` functions protect the wait queue list with `write_lock_irqsave()` and `write_unlock_irqrestore()` functions.

- `setup_x86_irq()` adds a new interrupt handler for a specific IRQ and uses `spin_lock_irqsave()` and `spin_unlock_irqrestore()` to protect the handler list.

- `run_timer_list()` function safeguards dynamic timer data structures with `spin_lock_irq()` and `spin_unlock_irq()`.

- `handle_signal()` function protects the blocked field of the current process with `spin_lock_irq()` and `spin_unlock_irq()`.

Interrupt disabling is used in these functions for simplicity but is recommended for short critical regions, as it blocks communication between I/O devices and the CPU. Longer critical regions should be implemented using locking mechanisms.

locking through kernel semaphores:

- Locking is a common synchronization technique in the kernel. It's used when a control path needs to access a shared resource or enter a critical region.
- Kernel semaphores are one way to implement locking. They ensure that only one control path can access a resource at a time.
- Semaphores have fields like "count" to indicate resource availability and "wait" to store sleeping processes.
- When a process wants to acquire a semaphore-protected resource, it calls `down()`. This function decrements the count and suspends if the resource is busy.
- Waking up a process from suspension is controlled by the "waking" field, ensuring only one process succeeds in acquiring the resource.
- The `up()` function is used to release a semaphore lock. It increments the count, potentially waking up waiting processes.
- There are variations of `down()` like `down_trylock()` and `down_interruptible()` for specific use cases.
- Semaphores are mainly used to prevent race conditions during I/O disk operations and when interrupt handlers access kernel data structures.
- Spin locks are preferred in multiprocessor systems for more complex synchronization scenarios.

Different types of semaphores and avoiding deadlocks:

Semaphore Types:

1. **Slab Cache List Semaphore:** Protects the list of slab cache descriptors. It prevents race conditions when functions like `kmem_cache_create()` modify the list while others like `kmem_cache_shrink()` scan it. This semaphore is mainly relevant in multiprocessor systems.

2. **Memory Descriptor Semaphore:** Found in each `mm_struct` memory descriptor, this semaphore prevents race conditions that could occur when multiple lightweight processes share the same memory descriptor. For example, it ensures data consistency during memory region creation or extension using `do_mmap()`.

3. **Inode Semaphore:** In the context of filesystem handling, each inode data structure has an `i_sem` semaphore. It guards against race conditions when multiple processes access and modify files on disk simultaneously.

Avoiding Deadlocks:

- Deadlocks can occur when two or more semaphores are used because different paths may end up waiting for each other to release a semaphore.

- Linux typically avoids deadlock issues with semaphore requests because most kernel control paths acquire only one semaphore at a time.

- In cases where the kernel needs to acquire two semaphores, it follows a specific order: semaphores are requested in the order of their addresses, preventing potential deadlocks. This approach is used, for example, in `rmdir()` and `rename()` system calls when working with two inodes.

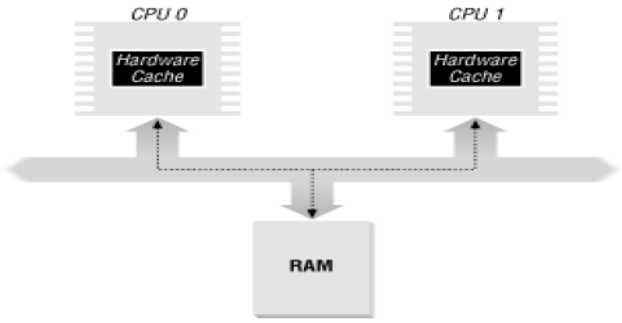
SMP architecture:

Symmetrical Multiprocessing (SMP):
- SMP is a multiprocessor architecture where all CPUs cooperate on an equal basis, without one being designated as a master CPU.
- The number of CPUs in an SMP system can vary, but it introduces challenges related to cache systems and cache synchronization.

Common Memory:
- SMP systems share the same memory, allowing multiple CPUs to access RAM concurrently.
- Memory arbiters control access to RAM chips, granting access to CPUs when the chips are available and delaying access when busy.

Hardware Cache Synchronization:
- Each CPU in an SMP system has its own local hardware cache.
- Cache synchronization ensures consistency between the cache and RAM. When a CPU modifies its cache, it checks if the same data is in another CPU's cache and notifies it to update with the correct value (cache snooping).
- These cache synchronization tasks are handled at the hardware level and are not a concern for the kernel.

Figure 11-1. The caches in a dual processor



Overall, an SMP kernel remains the same regardless of the number of CPUs in the system, simplifying kernel design and abstracting many hardware complexities.

In SMP (Symmetrical Multiprocessing) systems, several CPUs work together on an equal basis. Here are some key points related to SMP:

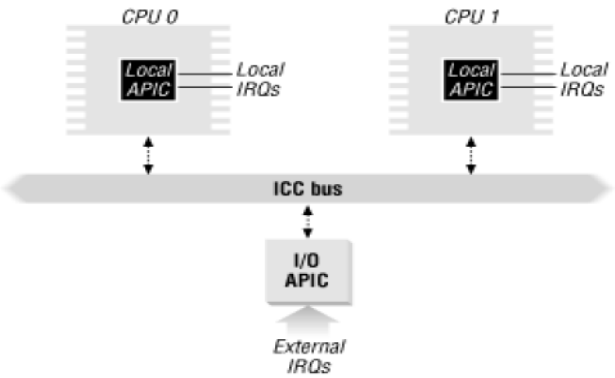
1. **Common Memory:** SMP systems share the same memory, and multiple CPUs can access RAM concurrently. Memory arbiters manage access to RAM chips to prevent conflicts.

2. **Hardware Cache Synchronization:** Each CPU has its own local hardware cache. Cache synchronization ensures that the cache contents remain consistent with RAM. This is done through cache snooping and is handled at the hardware level.

3. **SMP Atomic Operations:** Standard read-modify-write instructions are not atomic on multiprocessor systems. To ensure atomicity, the `lock` instruction prefix is used to lock the memory bus during such operations.

4. **Distributed Interrupt Handling:** SMP systems use the I/O APIC (I/O Advanced Programmable Interrupt Controller) to deliver interrupts to any CPU in the system. Each CPU has its own Local APIC, and an ICC (Interrupt Controller Communication) bus connects them to the I/O APIC. Interrupt requests can be distributed using fixed mode or lowest-priority mode. Interprocessor interrupts allow CPUs to send interrupts to each other.

Figure 11-2. APIC system



SMP systems introduce complexity due to cache synchronization and interrupt distribution, but the Linux kernel abstracts most of these hardware complexities, providing a unified kernel regardless of the number of CPUs in the system.

Kernel semaphores are represented as a struct with three fields:

1. `count`: An integer value that indicates whether a resource is available. If `count` is greater than 0, the resource is free and available. If `count` is less than or equal to 0, the semaphore is busy, and the absolute value of `count` represents the number of processes waiting for the resource. Zero means the resource is in use, but no other process is waiting for it.

2. `wait`: Stores the address of a wait queue list, containing sleeping processes waiting for the resource. If `count` is greater than or equal to 0, the wait queue is empty.

3. `waking`: Ensures that only one of the sleeping processes waiting for the resource succeeds in acquiring it when the resource becomes available.

These fields collectively help manage access to shared resources in the kernel, ensuring proper synchronization and control over resource availability.

Up()

```
function up(semaphore):
   semaphore.count++
   if semaphore.count <= 0:
      semaphore.waking++
      wake_up_one_process(semaphore.wait)
```

Process Scheduling :
Linux uses process scheduling to switch between multiple processes quickly, achieving apparent simultaneous execution. This involves choosing when and which process to run.
Scheduling Policy:
Linux scheduling balances objectives like fast response time, good throughput, and avoiding starvation. It's based on time-sharing, dividing CPU time into slices for each process.

Scheduling Policy:
- Linux ranks processes by priority and adjusts them dynamically based on their CPU usage.
- Processes are categorized as "I/O-bound" (waiting for I/O) or "CPU-bound" (needing a lot of CPU time).
Linux's scheduling ensures efficient use of CPU resources for various types of processes.

Processes in Linux can be classified into three categories based on their behavior:
1. **Interactive Processes:** These constantly interact with users, waiting for input like keypresses and mouse actions. To maintain responsiveness, the average delay should be between 50 and 150 ms with limited variance. Examples include command shells, text editors, and graphical applications.

2. **Batch Processes:** These run in the background without user interaction. They don't require immediate responsiveness and are often lower-priority. Examples include programming language compilers, database search engines, and scientific computations.

3. **Real-Time Processes:** Real-time processes have strict scheduling requirements. They should never be blocked by lower-priority processes, need a quick response time, and minimal variation in response time. Examples include video and sound applications, robot controllers, and sensor data collection.

While these classifications are somewhat independent, Linux prioritizes I/O-bound processes to ensure responsiveness for interactive applications. It explicitly recognizes real-time processes but doesn't distinguish between interactive and batch processes in scheduling.

Process Preemtation
In Linux, processes are preemptive, meaning they can be interrupted and switched out for another process in the TASK_RUNNING state with higher priority or when their time quantum expires. This ensures responsiveness and efficient use of CPU time.
- Preemption occurs when a higher-priority process becomes runnable.
- For example, if an interactive text editor has a higher priority than a compiler, it can be suspended during pauses but quickly resumed when the user presses a key.
- Preempted processes remain in TASK_RUNNING state but don't use the CPU.
Linux's kernel is not preemptive in Kernel Mode, simplifying synchronization. Preemption only happens in User Mode.

The duration of a quantum (time slice) is crucial:
- Too short quantum leads to high overhead from frequent task switches.
- Too long quantum can make processes seem non-concurrent, but it doesn't typically affect interactive applications' response time due to their higher priority.
- However, excessively long quantum can make the system appear unresponsive, especially when a CPU-bound process delays an interactive one.

The Scheduling Algorithm
The Linux scheduling algorithm divides CPU time into epochs, each with a specific time quantum for every process. Here's how it works:

1. **Time Quantum and Epochs:** Within each epoch, each process gets a time quantum, which is the maximum CPU time it can use. Different processes can have different quantum durations. When a process exhausts its quantum, it gets preempted, and another runnable process is chosen.

2. **Repeating Epochs:** A process can be selected multiple times in one epoch if it doesn't use up its entire quantum. The epoch ends when all runnable processes have used up their quantum. Then, the scheduler recalculates quantum durations for all processes, and a new epoch begins.

3. **Base Time Quantum:** Each process has a base time quantum, which is assigned by the scheduler when it exhausts its quantum in the previous epoch. Users can modify this base quantum using system calls like nice() and setpriority(). New processes inherit their parent's base quantum.

4. **Priority Types:** There are two types of priorities:
   - **Static Priority:** Assigned by users to real-time processes, ranging from 1 to 99. It remains fixed and is not changed by the scheduler.
   - **Dynamic Priority:** Applies to conventional processes. It's the sum of the base time quantum (base priority) and the remaining CPU time ticks in the current epoch.

5. **Priority Hierarchy:** Real-time processes always have higher priority than conventional ones. The scheduler runs conventional processes only when no real-time process is in a TASK_RUNNING state.

Data Structures Used by the Scheduler
The Linux scheduler uses various data structures and fields to manage processes and decide which one to run next. Here's an overview of these data structures and fields used by the scheduler:

1. **Process Lists:** Processes are linked together using process descriptors, and runnable processes are linked in a runqueue list. The `init_task` process descriptor acts as a list header.

2. **Fields in Process Descriptors:** Each process descriptor includes several fields related to scheduling:

   - `need_resched`: A flag checked to decide if scheduling needs to occur.
   - `policy`: Specifies the scheduling class (SCHED_FIFO, SCHED_RR, SCHED_OTHER).
   - `rt_priority`: The static priority of real-time processes.
   - `priority`: The base time quantum or priority of the process.
   - `counter`: The number of ticks of CPU time left before the quantum expires.

3. **Base Time Quantum:** Processes have a base time quantum, inherited from their parent or modified using system calls like `nice()` and `setpriority()`.

4. **Scheduling Priority:** Processes have static (real-time) or dynamic (conventional) priorities. Real-time processes always have higher priority.

5. **Scheduling Algorithm Actions:**

- **Direct Invocation:** The scheduler is invoked directly when a process must be blocked immediately due to resource unavailability.
- **Lazy Invocation:** The scheduler can be invoked lazily by setting the `need_resched` field of the current process.

6. **Schedule Function:** The `schedule()` function is responsible for selecting the next process to run based on priority and other factors.

7. **Actions Performed by `schedule()`:**
- It runs functions left by other kernel control paths in various queues.
- It executes active unmasked bottom halves.
- It selects the next process to run by comparing priorities.
- If the previous process used up its quantum, it assigns a new quantum and repositions it in the runqueue.
- If the previous process had pending signals and was in an interruptible state, it wakes it up.
- If the previous process is not in the TASK_RUNNING state, it removes it from the runqueue.
- If no suitable process is found, a new epoch begins with fresh quantum assignments for all processes.

8. **Context Switch:** If a different process is selected, a process switch occurs, and the `context_swtch` statistic is updated.

Overall, the scheduler manages processes' quantum durations, priorities, and selects the next process to run based on a set of rules and criteria.

System Calls Related to Real-Time Processes

The provided text describes several system calls related to real-time processes and their scheduling policies in Linux. Here's a summary of these system calls:

1. **sched_getscheduler():**
- This system call queries the scheduling policy currently applied to the specified process (identified by its PID).
- If the PID is set to 0, it retrieves the policy of the calling process.
- Returns the scheduling policy (SCHED_FIFO, SCHED_RR, or SCHED_OTHER).

2. **sched_setscheduler():**
- This system call sets both the scheduling policy and the associated parameters for the specified process (identified by its PID).
- If the PID is set to 0, it sets the scheduling parameters of the calling process.
- It checks the validity of the policy and static priority specified and the permissions of the caller (requires CAP_SYS_NICE capability or superuser rights).
- Sets the policy and real-time priority fields of the process descriptor.
- If the process is currently runnable, it may need to be moved in the runqueue.
- It sets `need_resched` for the current process, ensuring the scheduler is invoked.

3. **sched_yield():**
- This system call allows a process to voluntarily relinquish the CPU without being suspended.
- The process remains in a TASK_RUNNING state but is placed at the end of the runqueue list, giving other processes with the same dynamic priority a chance to run.
- Mainly used by SCHED_FIFO processes.

- Sets the SCHED_YIELD flag in the policy field if the process is SCHED_OTHER.
- Sets `need_resched` for the current process.
- Moves the process to the end of the runqueue list.

4. **sched_get_priority_min() and sched_get_priority_max():**
- These system calls return the minimum and maximum real-time static priority values that can be used with the specified scheduling policy.
- Always returns 1 for minimum priority if the current process is a real-time process.
- Always returns 99 for maximum priority if the current process is a real-time process.

5. **sched_rr_get_interval():**
- This system call is supposed to retrieve the round-robin time quantum for a named real-time process.
- However, it does not operate as expected and always returns a 150-millisecond value in the timespec structure.
- This system call is effectively unimplemented in Linux.

These system calls provide ways for processes to query and modify their scheduling policies and priorities, as well as to yield the CPU voluntarily when needed.

Memory Addressing

Memory Management:
Memory management in Unix is complex.
Unix uses virtual memory for various benefits.
Virtual memory acts as a middleman between apps and hardware.
It allows running big apps, sharing memory, and simplifies coding.
Virtual memory uses virtual addresses, translated to physical by the CPU.
RAM is divided between kernel and virtual memory usage.
Balancing memory types is crucial, and fragmentation is an issue.

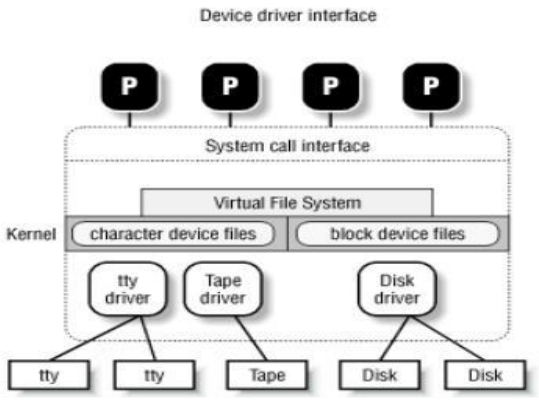
RAM (Random Access Memory):
RAM is computer memory used for active tasks.
It's divided into two parts: kernel and virtual memory.
Kernel part stores the core of the operating system.
The rest is used by applications for their data and processes.
RAM is faster than storage but volatile (loses data when powered off).
Balancing memory usage is crucial, and fragmentation can be a problem.

Kernel Memory Allocator (KMA):
KMA handles memory requests from different parts of the system.
It's crucial for speed because all kernel subsystems use it.
Good KMA features include speed, minimal wasted memory, reduced fragmentation, and cooperation with other memory management subsystems.
Different KMA techniques include resource map allocator, power-of-two freelists, McKusick-Karels allocator, buddy system, Mach's zone allocator, Dynix allocator, and Solaris's slab allocator.

Process Virtual Address Space:
In Unix, a process has a virtual address space that includes program code, data, stack, libraries, and dynamic memory. Recent Unix systems use demand paging, which means pages are loaded into physical memory only when needed. When a process accesses a missing page, the system allocates a page and fills it with the required data. This helps manage memory efficiently and allows processes to start even if not all pages are in physical memory.

Swapping and Caching:
Unix uses swapping to manage memory by moving pages between disk and RAM when needed.
Caching stores frequently used data in RAM to speed up access.

Device Drivers:
Device Drivers:
1. Kernel modules for I/O devices.
2. Each has a unique interface.
3. Benefits: encapsulation, vendor-friendly, uniform.
4. Load/unload without rebooting.
5. Simplifies device management.



Memory Addresses:
Memory address is how we access memory cell content.
Three kinds of addresses:

Logical Address:
Used in machine language instructions to specify operand or instruction address.
Comprises a segment and an offset, denoting the distance from segment start to the actual address.

Linear Address:
A single 32-bit unsigned integer.
Addresses up to 4GB, or 4,294,967,296 memory cells.

Physical Addresses:
Used to address memory cells in memory chips.
Correspond to electrical signals sent from the processor to the memory bus.
Represented as 32-bit unsigned integers.

Segmentation in Hardware:
Intel microprocessors perform address translation in two ways: real mode and protected mode.
Real mode is used for compatibility with older models and bootstrapping.
The focus is on protected mode, which provides advanced memory protection and security features.

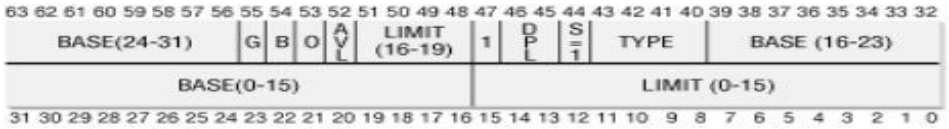
Segmentation Registers:
Logical address = Segment Selector (16-bit) + Offset (32-bit) within the segment.
Processors have segmentation registers: cs, ss, ds, es, fs, and gs.
These registers hold Segment Selectors.
Programs can reuse these registers for different purposes by saving content in memory and restoring it later.

Specific Registers:
cs: Code Segment Register, points to the segment with program instructions.
ss: Stack Segment Register, points to the segment with the current program stack.
ds: Data Segment Register, points to the segment with static and external data.

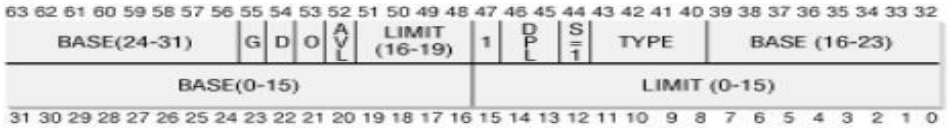
Segment Descriptors:
Each segment has an 8-byte Segment Descriptor describing its characteristics.
Segment Descriptors are stored in the Global Descriptor Table (GDT) or the Local Descriptor Table (LDT).
Typically, there's only one GDT, but each process can have its LDT.

The GDT's address in memory is stored in the gdtr processor register, while the currently used LDT's address is in the ldtr processor register.

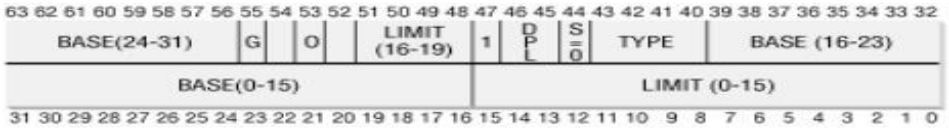
Data Segment Descriptor



Code Segment Descriptor



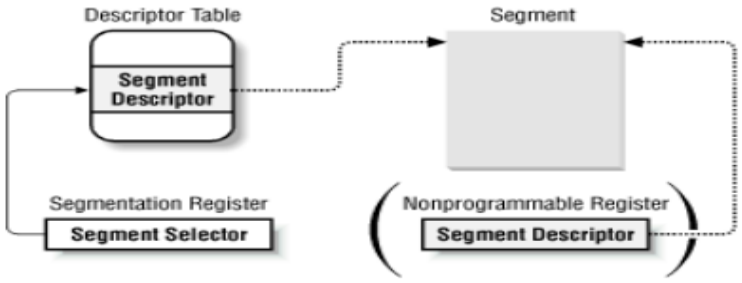
System Segment Descriptor



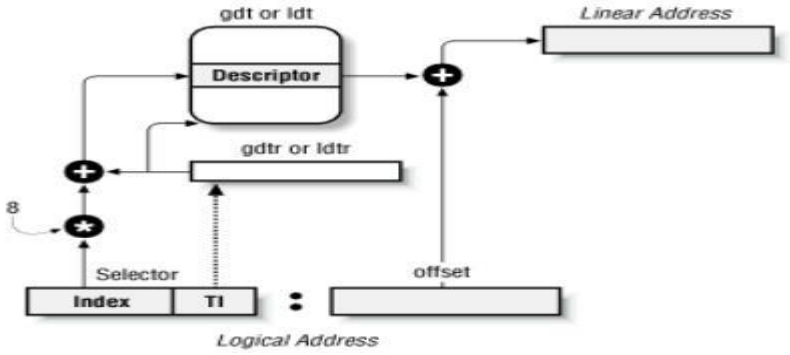
Segment Descriptor Fields:
Each Segment Descriptor has these fields:
32-bit Base: Linear address of the segment's first byte.
G Granularity Flag: Indicates segment size (bytes or multiples of 4096).
20-bit Limit: Denotes segment length in bytes (1B to 1MB or 4KB to 4GB).
S System Flag: System or normal segment (kernel data or code).
4-bit Type: Characterizes segment type and access rights.
D or B Flag: Depends on code or data, indicating 32-bit or 16-bit addresses.
Reserved bit: Always set to 0.
AVL (Available) Flag: Optionally used by the operating system but ignored in Linux.

Segment Selector and Segment Descriptor:
Intel processors use Segment Selectors as memory segment pointers. Loading a Selector into a segmentation register fetches its Segment Descriptor into a CPU register. This speeds up address translation for that segment without constantly accessing the GDT or LDT in memory, making memory operations more efficient.

A 13-bit index in the Segment Selector points to a specific entry in either the GDT or LDT. The TI flag (Table Indicator) distinguishes between the GDT (TI=0) and LDT (TI=1). The RPL (Requestor Privilege Level), a 2-bit field, indicates the CPU's current privilege level when loading the Selector. The Segment Descriptor's position in the GDT or LDT is calculated by multiplying the top 13 bits of the Selector by 8.

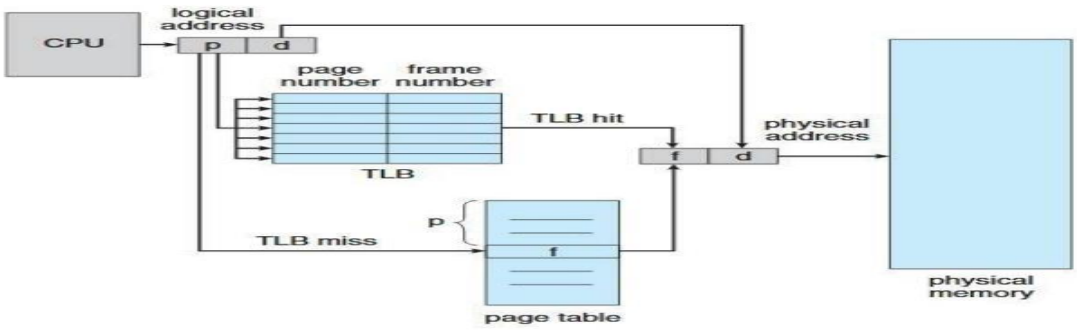


Segmentation Unit:

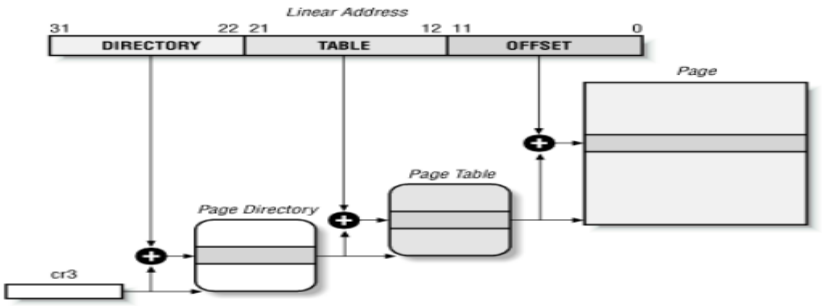


- Translates logical addresses to linear addresses.
- Determines Descriptor Table (GDT or LDT) based on the TI field in Segment Selector.
- Computes Descriptor address by multiplying index by 8 and adding to gdtr/ldtr register.
- Adds logical address offset to the Segment Descriptor's Base to get linear address.

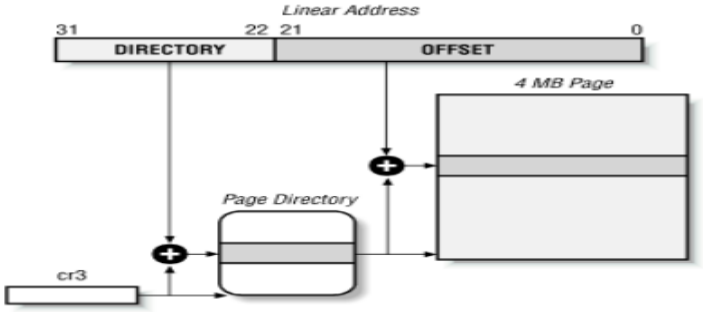
Paging in Hardware:
- Hardware translates linear addresses to physical ones.
- Checks access rights, generates page fault if invalid.
- Linear addresses grouped into fixed-size pages.
- Each page corresponds to a page frame in RAM.
- Page tables map linear to physical addresses.
- Enabled by setting the PG flag in the cr0 register.



Regular Paging:
- Intel processors from i80386 use 4KB pages.
- Linear address (32 bits) is divided into three fields:
  - Directory: Top 10 bits.
  - Table: Middle 10 bits.
  - Offset: Bottom 12 bits.
- Translation occurs in two steps with Page Directory and Page Table.
- Physical address of Page Directory is in cr3 processor register.
- Directory field points to the Page Table entry.
- Table field determines the Page Table entry containing the page frame's physical address.
- Offset field specifies the position within the page frame, as each page is 4096 bytes.



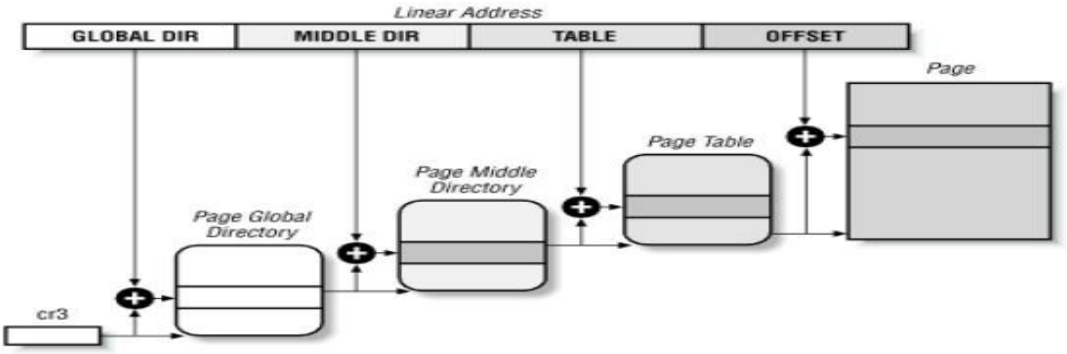
Extended Paging:
Extended Paging (Intel Processors):
- Introduced in Pentium and later CPUs.
- Enables 4KB or 4MB page sizes.
- Set PageSize flag in Page Directory entry.
- Splits linear address into Directory (top 10 bits) and Offset (remaining 22 bits).
- Coexists with regular paging, enabled by PSE flag in cr4 register.
- Translates large contiguous address ranges, conserving memory by eliminating intermediates.



Hardware Protection Scheme:
In hardware protection, the paging unit and segmentation unit employ distinct schemes. Two privilege levels are defined for pages and Page Tables, governed by the User/Supervisor flag. When the flag is 0, the page is only accessible when the Current Privilege Level (CPL) is less than 3. When the flag is 1, the page is always accessible. Pages also have two access rights (Read and Write). If the Read/Write flag in a Page Directory or Page Table entry is 0, the corresponding Page Table or page can only be read; otherwise, it can be both read and written to.

Three-Level Paging:
Introduced for 64-bit architectures.
Typically, 16 KB page size is chosen, which results in a 14-bit Offset field.
The remaining 50 bits of the linear address are distributed between the Table and Directory fields.
In a three-level paging scheme, 30 bits of the address are split into three 10-bit fields.
Page Tables have 210 (1024) entries at each level.
Allows addressing large memory spaces efficiently.

Linux and Paging:
Linux relies heavily on paging for process management.
Assigns a different physical address space to each process, ensuring protection against addressing errors.
Distinguishes between pages (groups of data) and page frames (physical addresses in main memory).
Enables features like virtual memory, allowing pages to be stored in different page frames, swapped to disk, and reloaded in different page frames.
Paging is a fundamental aspect of Linux memory management.



The Linear Address field
PAGE_SHIFT: Specifies page size, usually 4096 bytes (4 KB).
PMD_SHIFT: Defines the size of an area mapped by a Page Middle Directory entry, often 4 MB.
PGDIR_SHIFT: Specifies the size of an area mapped by a Page Global Directory entry, also typically 4 MB.
And these macros tell us:
PTRS_PER_PTE: The number of entries in a Page Table, usually 1024.
PTRS_PER_PMD: Entries in a Page Middle Directory, typically 1.
PTRS_PER_PGD: Entries in a Page Global Directory, usually 1024.