Department of Electrical And Computer Engineering
Istanbul , Turkey
ECE519 Advanced Operating Systems

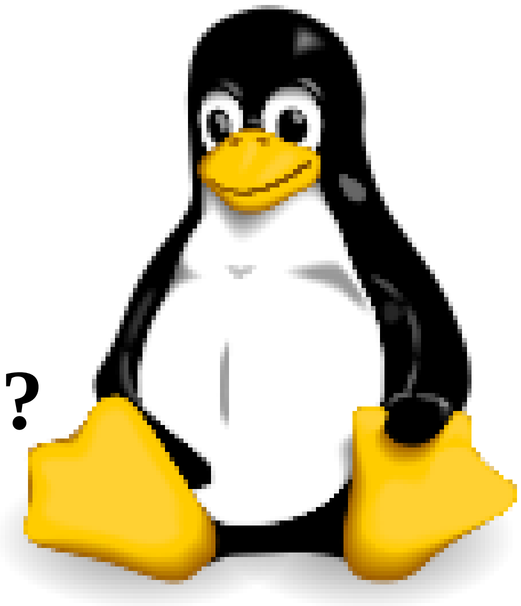
Linux Kernel Concurrency Mechanisms

By
Mabruka Khlifa Karkeb
Student Id: 163103069
Prof. Dr. Hasan Hüseyin

Spring 2017

Overview

- What is Concurrency?
- Causes of Concurrency
- What is Kernel ?
  - Why do we update Kernel ?
- Critical Sections
- Linux Kernel Concurrency Mechanism
- Reference

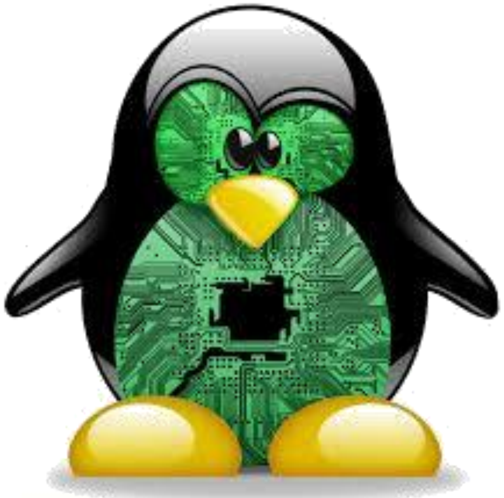


What is Concurrency?

Concurrency is the tendency for things to happen at the same time in a system. It is a natural phenomenon, of course. In the real world, at any given time, many things are happening simultaneously. When we design software to monitor and control real-world systems, we must deal with this natural concurrency.

Causes of Concurrency

- Pseudo concurrency .
- True concurrency .



Sources of Concurrency

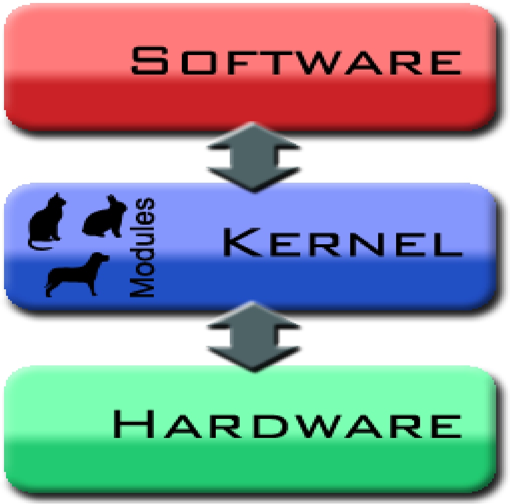
- Interrupts
- Softirqs and tasklets
- Kernel preemption
- Sleeping
- SMP

What is Kernel ?

A kernel is a central component of an operating system. It acts as an interface between the user applications and the hardware.

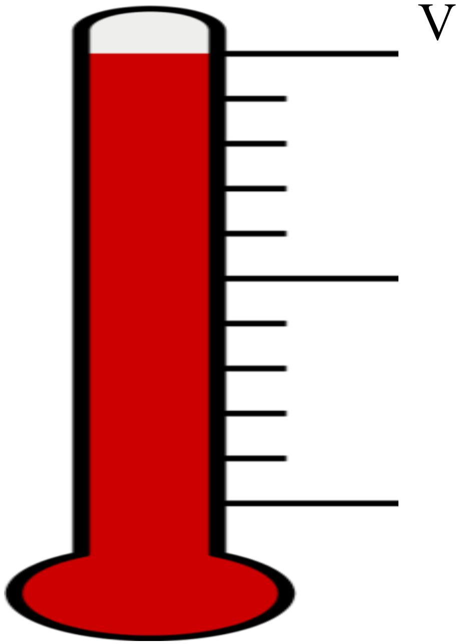
The main tasks of the kernel are :

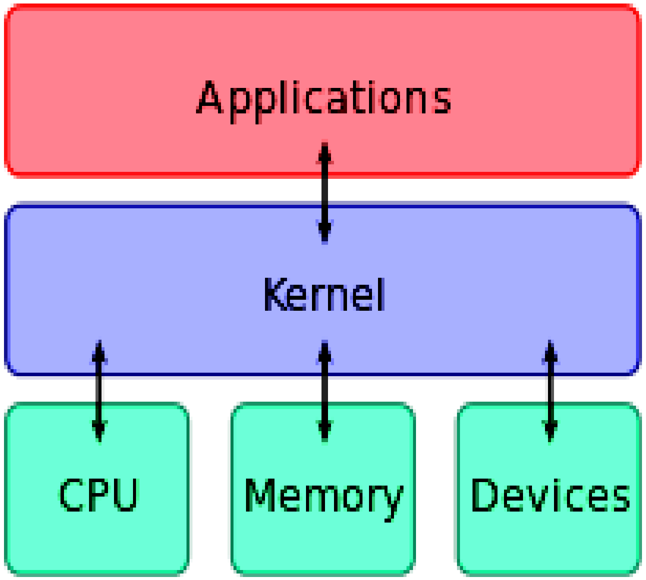
- Process management
- Device management
- Memory management
- Interrupt handling
- I/O communication
- File system...etc..



Why do we update Kernel ?

The developers are modifying the Kernel, to improve the way the devices are handled and the device is disconnected, allowing better performance for these devices and solving some problems such as heat up the device.





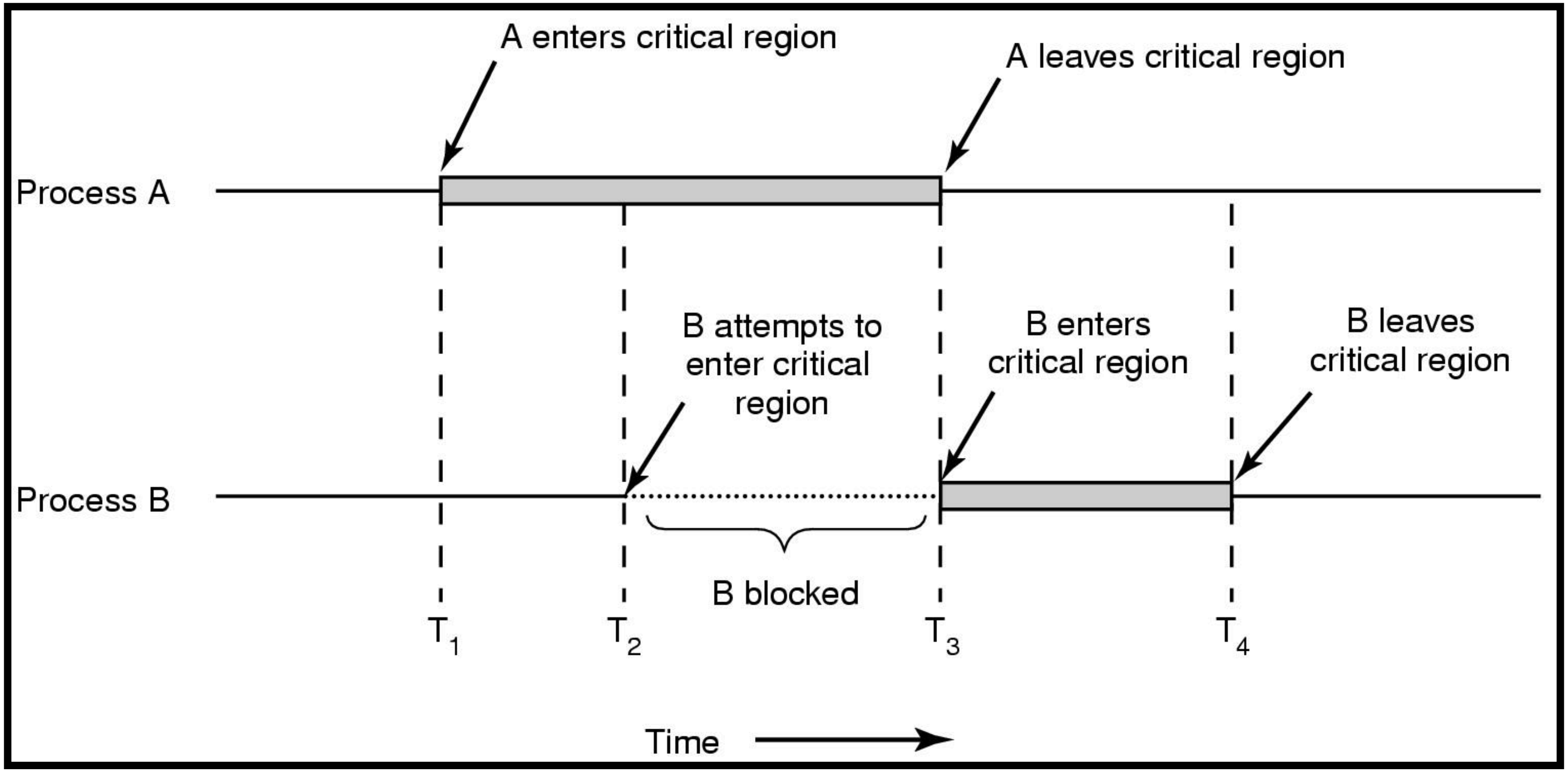
Critical Sections

A critical region is a section of code that is always executed under mutual exclusion . It shift the responsibility for enforcing mutual exclusion from the programmer.

(where it resides when semaphores are used) to the compiler.

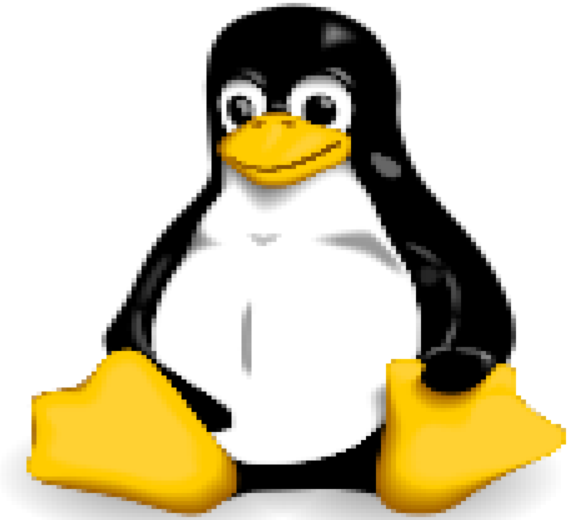
Critical Sections

Code that accesses shared resource.



Outline

- ➢ **Linux Kernel Concurrency Mechanism**
  - ➢ **Kernel Synchronization Techniques**
  - ➢ **Atomic Operations**
  - ➢ **Spinlocks**
  - ➢ **Semaphores**
  - ➢ **Barriers**



Linux Kernel Concurrency Mechanism

Linux includes all of the concurrency mechanisms found in other UNIX systems, such as SVR4, including pipes, messages, shared memory, and signals. In addition , Linux 2.6 includes a rich set of concurrency mechanisms specifically intended for use when a thread is executing in kernel mode. That is, these are mechanisms used within the kernel to provide concurrency in the execution of kernel code.

Kernel Synchronization Techniques

Technique	**Description**	**Scope**
Per-CPU vars	Each CPU has data	All CPUs
Atomic operation	Atomic operation	All CPUs
Memory barrier	Avoid re-ordering	Local or All CPUs
Spin lock	Lock w/ busy wait	All CPUs
Semaphore	Lock w/ sleep wait	All CPUs
Seqlocks	Lock on access ctr	All CPUs
Interrupt disabling	cli on a single CPU	Local CPU
SoftIRQ disabling	Forbid SoftIRQs	Local CPU
Read-copy-update (RCU)	Lock-free access to shared data via ptrs.	All CPUs

Atomic Operations

- Linux provides a set of operations that guarantee atomic operations on a variable. These operations can be used to avoid simple race conditions. An atomic operation executes without interruption and without interference.

Type of Atomic Operations

- Have Two types of atomic operations are defined in Linux:

Integer Operations

- Integer operations which operate on an integer variable, typically used to implement counters.

Bitmap Operations

- Bitmap operations which operate on one bit in a bitmap. operate on one of a sequence of bits at an arbitrary memory location indicated by a pointer variable

Type of Atomic Operations

Two types of atomic operations are defined in Linux

Integer Operations

At declaration: initialize an atomic_t to i	ATOMIC_INIT (int i)
Read integer value of v	int atomic_read(atomic_t *v)
Set the value of v to integer i	void atomic_set(atomic_t *v, int i)
Add i to v	void atomic_add(int i, atomic_t *v)
Subtract i from v	void atomic_sub(int i, atomic_t *v)
Add 1 to v	void atomic_inc(atomic_t *v)
Subtract 1 from v	void atomic_dec(atomic_t *v)

Bitmap Operations

Set bit nr in the bitmap pointed to by addr	**void set_bit(int nr, void *addr)**
Clear bit nr in the bitmap pointed to by addr	**void clear_bit(int nr, void *addr)**
Invert bit nr in the bitmap pointed to by addr	**void change_bit(int nr, void *addr)**
Set bit nr in the bitmap pointed to by addr; return the old bit value	**int test_and_set_bit(int nr, void *addr)**
Clear bit nr in the bitmap pointed to by addr; return the old bit value	**int test_and_clear_bit(int nr, void *addr)**

Spin Lock

The most common technique used for protecting a critical section in Linux is the spinlock. Only one thread at a time can acquire a spinlock. Any other thread attempting to acquire the same lock will keep trying (spinning) until it can acquire the lock. In essence, a spinlock is built on an integer location in memory that is checked by each thread before it enters its critical section

Using a Spin Lock

The basic form of use of a spinlock is the following

```
spin_lock(&lock)
/* critical section */
spin_unlock(&lock)
```

Inside a Spin Lock

1. Disables kernel pre-emption.
2. Atomic test-and-sets lock.
3. If old value positive

   Lock acquired.
4. Else

   Enables pre-emption.

   If break_lock is 0, sets to 1 to indicate a task is waiting.

   Busy wait loop

   ```
   while (spin_is_locked(lock))
         cpu_relax(); # pause instruction on P4
   ```

   Goto 1.

Spin Lock Functions

`spin_lock_init(spinlock_t *lock)`

Initialize spin lock to 1 (unlocked).

`spin_lock(spinlock_t *lock)`

Spin until lock becomes 1, then set to 0 (locked).

`spin_lock_irqsave(spinlock_t *l, u flags)`

Like `spin_lock()` but disables and saves interrupts.

Always use an IRQ disabling variant in interrupt context.

`spin_unlock(spinlock_t *lock)`

Set spin lock to 1 (unlocked).

`spin_lock_irqrestore(spinlock_t *l, u flags)`

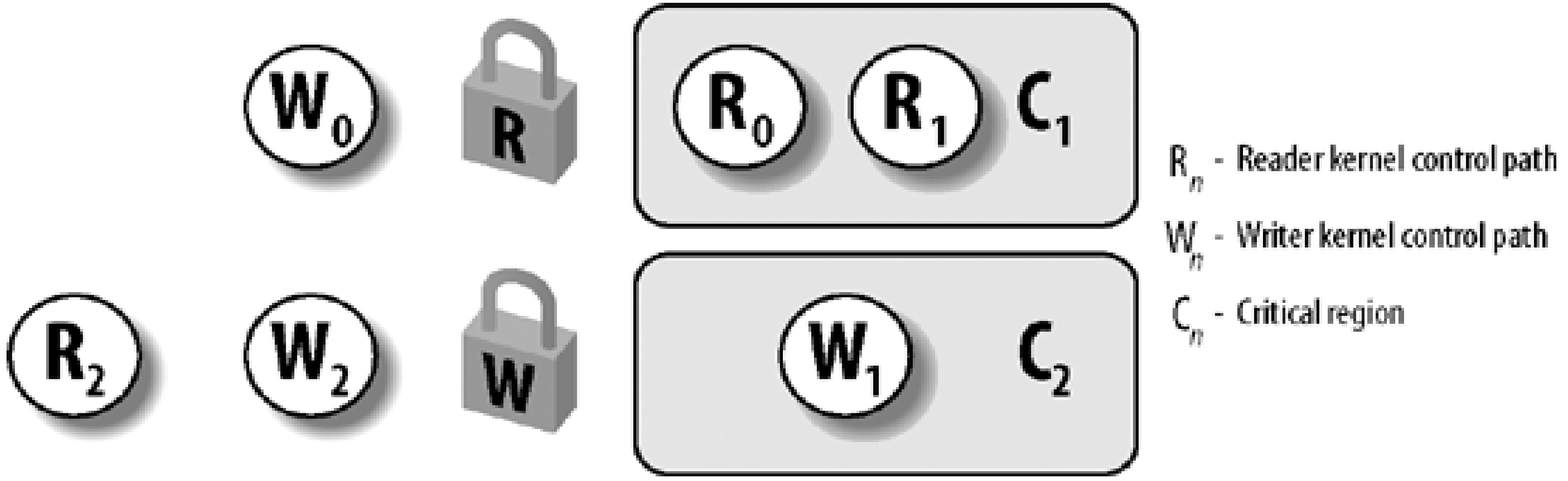
Like spin_lock(), but restores interrupt status.

`spin_trylock(spinlock_t *lock)`

Set lock to 0 if unlocked and return 1; return 0 if locked.

Read/Write Spinlocks

- Multiple readers can acquire lock simultaneously.
- Only one writer can have the lock at a time.
- Increase concurrency by allowing many readers.
- Example use: task list



Spin Lock

Advantage And Disadvantage

The spinlock is **easy** to implement but has the **disadvantage** that locked-out threads continue to execute in a busy-waiting mode.

Semaphores

At the user level, Linux provides a semaphore interface corresponding to that in UNIX SVR4. Internally, Linux provides an implementation of semaphores for its own use. That is, code that is part of the kernel can invoke kernel semaphores. These kernel semaphores cannot be accessed directly by the user program via system calls. They are implemented as functions within the kernel and are thus more efficient than user-visible semaphores.

Semaphores

Types of semaphore

Linux provides three types of semaphore facilities in the kernel:

- Binary semaphores.
- Counting semaphores.
- Reader-writer semaphores

Linux Traditional Semaphores Functions

```
DECLARE_MUTEX(sem);
```

Static declares a mutex semaphore.

```
void init_MUTEX(struct semaphore *sem);
```

Dynamic declaration of a mutex semaphore.

```
void down(struct semaphore *sem);
```

Decrements semaphore and sleeps.

```
int down_interruptible(struct semaphore *sem);
```

Same as `down()` but returns on user interrupt.

```
int down_trylock(struct semaphore *sem);
```

Same as `down()` but returns immediately if not available.

```
void up(struct semaphore *sem);
```

Releases semaphore.

Read/Write Semaphores

The reader-writer semaphore divides users into readers and writers; it allows multiple concurrent readers (with no writers) but only a single writer (with no concurrent readers).

Read/Write Semaphores

Table shows the basic reader-writer semaphore operations

Initalizes the dynamically created semaphore with a count of 1	void init_rwsem(struct rw_semaphore, *rwsem)
Down operation for readers	void down_read(struct rw_semaphore, *rwsem)
Up operation for readers	void up_read(struct rw_semaphore, *rwsem)
Down operation for writers	void down_write(struct rw_semaphore, *rwsem)
Up operation for writers	void up_write(struct rw_semaphore, *rwsem)

Spin Locks VS Semaphores

- Spin Locks
  - Busy waits waste CPU cycles.
  - Can use in interrupt context, as does not sleep.
  - Cannot use when code sleeps while holding lock.
  - Use for locks held a short time.
- Semaphores
  - Context switch on sleep is expensive.
  - Sleeps, so cannot use in interrupt context.
  - Can use when code sleeps while holding lock.
  - Use for locks that held a long time.

Barriers provide Ordering

- Optimization Barriers
  - Prevent compiler from re-ordering instructions.
  - Compiler doesn't know when interrupts or other processors may read/write your data.
  - Kernel provides `barrier()` macro.
- Memory Barriers
  - Read/write barriers prevent loads/stores from being re-ordered across barrier.
  - Kernel provides `rmb()`, `wmb()` macros.

All syncronization primitives act as barriers.

Some Reference

- http://www.linux-mag.com/id/2316/
- http://ww2.cs.fsu.edu/~stanovic/teaching/ldd_summer_2014/syllabus.html

Operating Systems INTERNALS AND DESIGN PRINCIPLES 7 EDITION By William Stallings

Thank you
for listening

Up INDEX

OUT OF SHOW

Critical Sections

1. No two processes may be simultaneously inside their critical sections.
2. No assumptions may be made about speed or number of CPUs.
3. No process running outside its critical section may block other processes from entering the critical section.
4. No process should have to wait forever to enter its critical section.

Causes of Concurrency

- **Pseudo concurrency :**

Two things do not actually happen at the same time but interleave with each other, which may be caused by preemption or signal.

- **True concurrency**

A symmetrical multiprocessing machine, two processes can actually be executed in a critical region at the exact same time.

Spin Locks

- If lock "open"
  - Sets lock bit with atomic test-and-set.
  - Continues into critical section.
- else lock "closed"
  - Code "spins" in busy wait loop until available.
  - Waits are typically much less than 1ms.
  - Kernel-preemption can run other processes while task is busy waiting.

Spin Lock State.

`spinlock_t`

`slock`

Spin lock state.

1 is unlocked.

< 1 is locked.

`break_lock`

Flag signals that task is busy waiting for this lock.

Why do we need atomicity?

Problem: Two processes incrementing i.

Uniprocessor Version

A: read i(7)

A: incr i(7 -> 8)

B: read i(8)

B: incr i(8 -> 9)

Process A	**Process B**
read i(7)	read i(7)
incr i(7 -> 8)	-
-	incr i(7 -> 8)
write i(8)	-
	write i(8)

Atomicity doesn't provide Ordering

One atomic order of operations:

Process A	Process B
atomic_inc i (7->8)	-
	atomic_inc i (8->9)

Another atomic order of operations:

Process A	Process B
-	atomic_inc i (7->8)
atomic_inc i (8->9)	

Atomic Operations

Atomic operations are indivisible.

Process A	Process B
atomic_inc i (7->8)	-
	atomic_inc i (8->9)

Provided by `atomic_t` in the kernel.

x86 assembly: lock byte preceding opcode makes atomic.

Atomic Operations

`atomic_t` guarantees atomic operations

```
atomic_t v;
atomic_t u = ATOMIC_INIT(0);
```

Atomic operations

```
atomic_set(&v, 4);
atomic_add(2, &v);
atomic_inc(&v);
printk("%d\n", atomic_read(&v));
atomic_dec_and_test(&v);
```

Read/Write Semaphores

One writer or many readers can hold lock.

```
static DECLARE_RWSEM(my_rwsem);

down_read(&my_rwsem);
/* critical section (read only) */
up_read(&my_rwsem);
down_write(&my_rwsem);
/* critical section (read/write) */
up_write(&my_rwsem);
```

Linux Semaphores

```
#include <asm/semaphore.h>
struct semaphore sem;

init_MUTEX(&sem);

if (down_interruptible(&sem))
   return -ERESTARTSYS; /* user interrupt */
/*
 * critical section
 */
up(&sem);
```

Semaphores

Down (P): Request to enter critical region.

If S > 0, decrements S, enters region.

Else process sleeps until semaphore is released.

Up (V): Request to exit critical region.

Increments S.

If S > 0, wakes sleeping processes.

Semaphores

If semaphore “open”

Task acquires semaphore.

else

Task placed on wait queue and sleeps.

Task awakened when semaphore released.

Semaphores

Integer value S with atomic access.

If S>0, semaphore prevents access.

Using a semaphore for mutual exclusion:

```
down(S);
/* critical section */
up(S);
```

What is Kernel ?

The kernel is the central module of an operating system (OS). It is the part of the operating system that loads first, and it remains in main memory. Because it stays in memory, it is important for the kernel to be as small as possible while still providing all the essential services required by other parts of the operating system and applications.