Linux Internals & Networking

System programming using Kernel interfaces

Team Emertxe

Threads

Threads

- Threads, like processes, are a mechanism to allow a program to do more than one thing at a time
- As with processes, threads appear to run concurrently
- The Linux kernel schedules them asynchronously, interrupting each thread from time to time to give others a chance to execute
- Threads are a finer-grained unit of execution than processes
- That thread can create additional threads; all these threads run the same program in the same process
- But each thread may be executing a different part of the program at any given time

Threads
Advantages

- Takes less time to create a new thread in an existing process than to create a brand new process
- Switching between threads is faster than a normal context switch
- Threads enhance efficiency in communication between different executing programs
- No kernel involved

Threads
Compilation

.Use the following command to compile the programs using thread libraries

```
.$ gcc -o <output_file> <input_file.c> -lpthread
```

Threads
Creation

.The **pthread_create** function creates a new thread

Function	Meaning
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg)	✓A pointer to a pthread_t variable, in which the thread ID of the new thread is stored ✓A pointer to a thread attribute object. If you pass NULL as the thread attribute, a thread will be created with the default thread attributes ✓A pointer to the thread function. This is an ordinary function pointer, of this type: void* (*) (void*) ✓A thread argument value of type void *. Whatever you pass is simply passed as the argument to the thread function when thread begins executing

Threads
Creation

- A call to **pthread_create** returns immediately, and the original thread continues executing the instructions following the call
- Meanwhile, the new thread begins executing the thread function
- Linux schedules both threads asynchronously
- Programs must not rely on the relative order in which instructions are executed in the two threads

Threads
Joining

- It is quite possible that output created by a thread needs to be integrated for creating final result
- So the main program may need to wait for threads to complete actions
- The pthread_join() function helps to achieve this purpose

Function	Meaning
int pthread_join(pthread_t thread, void **value_ptr)	✓Thread ID of the thread to wait ✓Pointer to a void* variable that will receive thread finished value ✓If you don't care about the thread return value, pass NULL as the second argument.

Threads
Passing Data

- The thread argument provides a convenient method of passing data to threads
- Because the type of the argument is **void***, though, you can't pass a lot of data directly via the argument
- Instead, use the thread argument to pass a pointer to some structure or array of data
- Define a structure for each thread function, which contains the "parameters" that the thread function expects
- Using the thread argument, it's easy to reuse the same thread function for many threads. All these threads execute the same code, but on different data

Threads
Return Values

- If the second argument you pass to **pthread_join** is non-null, the thread's return value will be placed in the location pointed to by that argument
- The thread return value, like the thread argument, is of type **void***
- If you want to pass back a single int or other small number, you can do this easily by casting the value to **void*** and then casting back to the appropriate type after calling **pthread_join**

Threads
Attributes

- Thread attributes provide a mechanism for fine-tuning the behaviour of individual threads
- Recall that **pthread_create** accepts an argument that is a pointer to a thread attribute object
- If you pass a null pointer, the default thread attributes are used to configure the new thread
- However, you may create and customize a thread attribute object to specify other values for the attributes

Threads
Attributes

- There are multiple attributes related to a
- particular thread, that can be set during creation
- Some of the attributes are mentioned as follows:
  - Detach state
  - Priority
  - Stack size
  - Name
  - Thread group
  - Scheduling policy
  - Inherit scheduling

Threads
Joinable and Detached

- A thread may be created as a ***joinable thread*** (the default) or as a ***detached thread***
- A joinable thread, like a process, is not automatically cleaned up by GNU/Linux when it terminates
- Thread's exit state hangs around in the system (kind of like a zombie process) until another thread calls **pthread_join** to obtain its return value. Only then are its resources released
- A detached thread, in contrast, is cleaned up automatically when it terminates
- Because a detached thread is immediately cleaned up, another thread may not synchronize on its completion by using **pthread_join** or obtain its return value

Threads
Creating a Detached Thread

.In order to create a detached thread, the thread attribute needs to be set during creation

.Two functions help to achieve this

Function	Meaning
int pthread_attr_init(pthread_attr_t *attr)	✓Initializing thread attribute ✓Pass pointer to pthread_attr_t type ✓Reurns integer as pass or fail
int pthread_attr_setdetachstate (pthread_attr_t *attr, int detachstate);	✓Pass the attribute variable ✓Pass detach state, which can take .PTHREAD_CREATE_JOINABLE .PTHREAD_CREATE_DETACHED

Threads
ID

.Occasionally, it is useful for a sequence of code to determine which thread is executing it.

. Also sometimes we may need to compare one thread with another thread using their IDs

. Some of the utility functions help us to do that

Function	Meaning
pthread_t pthread_self()	✓Get self ID
int pthread_equal(pthread_t threadID1, pthread_t threadID2);	✓Compare threadID1 with threadID2 ✓If equal return non-zero value, otherwise return zero

Synchronization - Concepts

Synchronization
why?

- In a multi-tasking system the most critical resource is CPU. This is shared between multiple tasks / processes with the help of ‘scheduling’ algorithm
- When multiple tasks are running simultaneously:
  - Either on a single processor, or on
  - A set of multiple processors
- They give an appearance that:
  - For each process, it is the only task in the system.
  - At a higher level, all these processes are executing efficiently.
  - Process sometimes exchange information:
  - They are sometimes blocked for input or output (I/O).
- Whereas multiple processes run concurrently in a system by communicating, exchanging information with others all the time. They also have very close dependency with various I/O devices and peripherals.

Synchronization
why?

- Considering resources are lesser and processes are more, there is a contention going between multiple processes

- Hence resource needs to be shared between multiple processes. This is called as ‘Critical section’.

- Access / Entry to critical section is determined by scheduling, however exit from critical section needs to be done when activity is completed properly

- Otherwise it will lead to a situation called ‘Race condition’

Synchronization
why?

- Synchronization is defined as a mechanism which ensures that two or more concurrent processes do not simultaneously execute some particular program segment known as critical section
- When one process starts executing the critical section (serialized segment of the program) the other process should wait until the first process finishes
- If not handled properly, it may cause a race condition where, the values of variables may be unpredictable and vary depending on the timings of context switches of the processes
- If any critical decision to be made based on variable values (ex: real time actions – like medical system), synchronization problem will create a disaster as it might trigger totally opposite action than what was expected

Synchronization

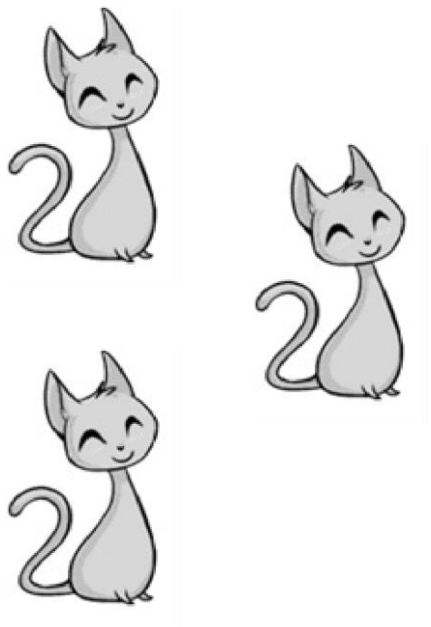
Race Condition in Embedded Systems

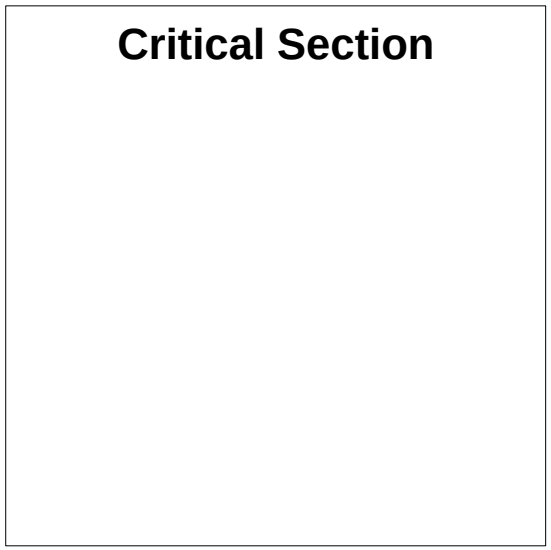
- Embedded systems are typically lesser in terms of resources, but having multiple processes running. Hence they are more prone to synchronization issues, thereby creating race conditions
- Most of the challenges are due to shared data condition. Same pathway to access common resources creates issues
- Debugging race condition and solving them is a very difficult activity because you cannot always easily re-create the problem as they occur only in a particular timing sequence
- Asynchronous nature of tasks makes race condition simulation and debugging as a challenging task, often spend weeks to debug and fix them

Synchronization
Critical Section

.The way to solve race condition is to have the critical section access in such a way that only one process can execute at a time

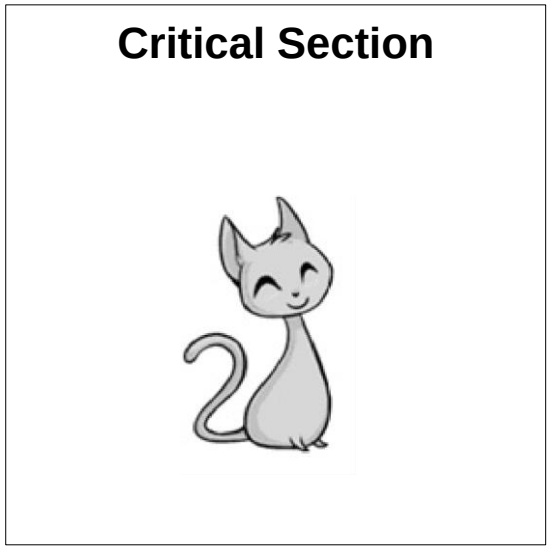
.If multiple process try to enter a critical section, only one can run and the others will sleep (means getting into blocked / waiting state)





Synchronization
Critical Section

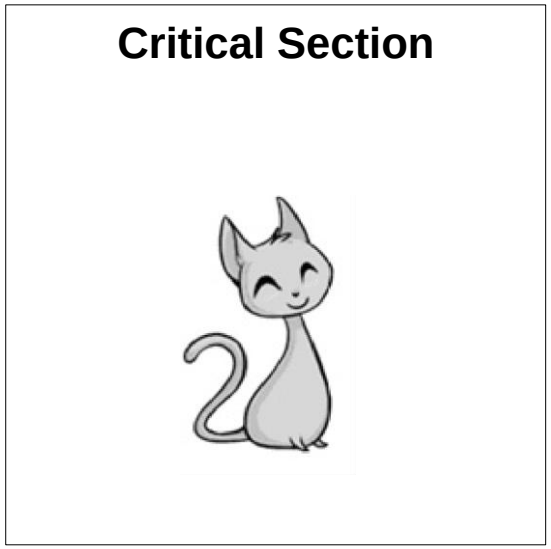
.Only one process can enter the critical section; the other two have to sleep. When a process sleeps, its execution is paused and the OS will run some other task



Synchronization
Critical Section

.Once the process in the critical section exits, another process is woken up and allowed to enter the critical section. This is done based on the existing scheduling algorithm

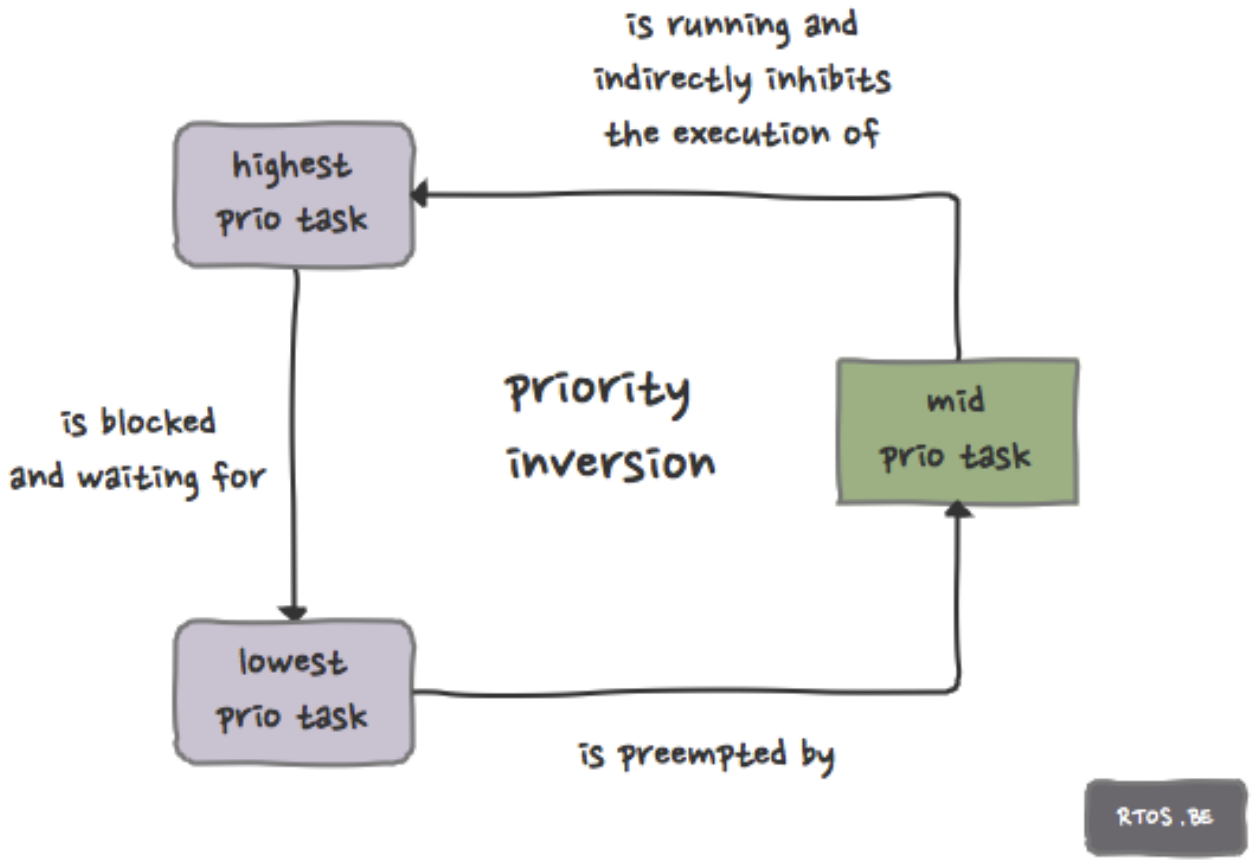
.It is important to keep the code / instructions inside a critical section as small as possible (say similar to ISR) to handle race conditions effectively



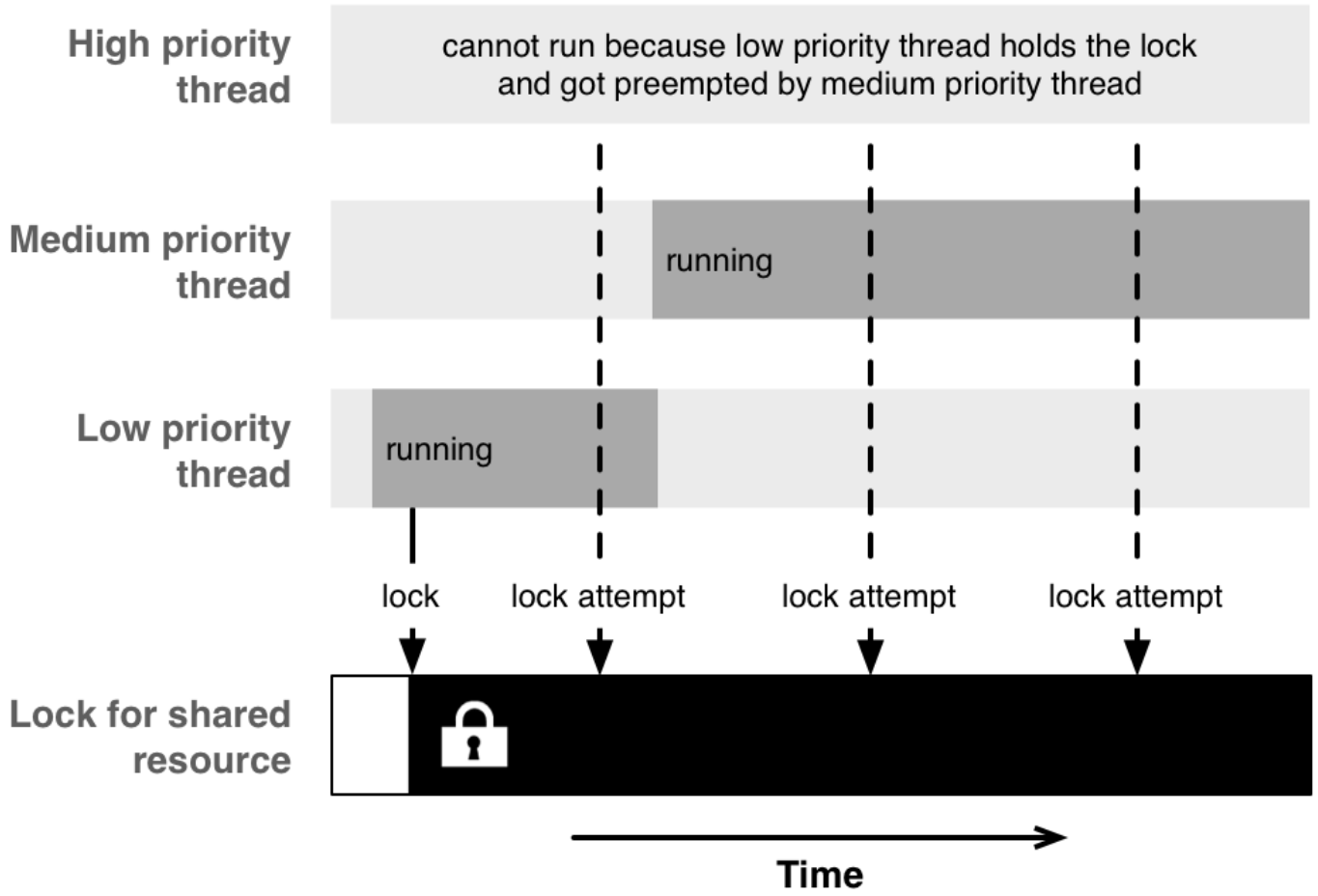
Synchronization
Priority Inversion

- One of the most important aspect of critical section is to ensure whichever process is inside it, has to complete the activities at one go. They should not be done across multiple context switches. This is called Atomicity
- Assume a scenario where a lower priority process is inside the critical section and higher priority process tries to enter
- Considering atomicity the higher priority process will be pushed into blocking state. This creates some issue with regular priority algorithm
- In this juncture if a medium priority tasks gets scheduled, it will enter into the critical section with higher priority task is made to wait. This scenario is further creating a change in priority algorithm
- This is called as ‘Priority Inversion’ which alters the priority schema

Synchronization
Priority Inversion



Synchronization
Priority Inversion



Quick refresher

.Before moving onto exploring various solutions for critical section problem, ensure we understand these terminologies / definitions really well.

- Difference between scheduling & Synchronization
- Shared data problem
- Critical section
- Race condition
- Atomicity
- Priority inversion

Critical Section
Solutions

Solution to critical section should have following three aspects into it:

.Mutual Exclusion: If process P is executing in its critical section, then no other processes can be executing in their critical sections

.Progress: If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely

.Bounded Waiting: A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

Critical Section
Solutions

.There are multiple algorithms (ex: Dekker's algorithm) to implement solutions that is satisfying all three conditions. From a programmers point of view they are offered as multiple solutions as follows:

- Locks / Mutex
- Readers–writer locks
- Recursive locks
- Semaphores
- Monitors
- Message passing
- Tuple space

.Each of them are quite detailed in nature, in our course two varieties of solutions are covered they are Mutex and Semaphores

.Let us look into them in detail!

Critical Section

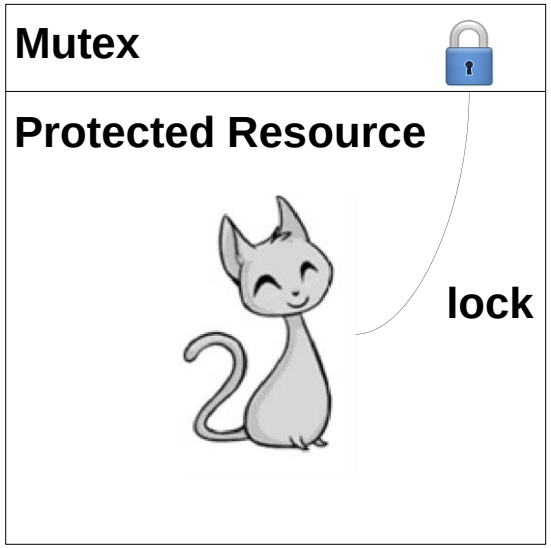
Solutions - Mutual Exclusion

- A Mutex works in a critical section while granting access
- You can think of a Mutex as a token that must be grabbed before execution can continue.

Critical Section

Solutions - Mutual Exclusion

.During the time that a task holds the mutex, all other tasks waiting on the mutex sleep.



Critical Section
Solutions - Mutual Exclusion

.Once a task has finished using the shared resource, it releases the mutex. Another task can then wake up and grab the mutex.

release

Critical Section

Solutions - Mutual Exclusion – Locking / Blocking

- A process may attempt to get a Mutex by calling a **lock** method. If the Mutex was unlocked (means already available), it becomes locked (unavailable) and the function returns immediately
- If the Mutex was locked by another process, the locking function **blocks** execution and returns only eventually when the Mutex is **unlocked** by the other process
- More than one process may be blocked on a locked Mutex at one time
- When the Mutex is unlocked, only one of the blocked process is unblocked and allowed to lock the Mutex. Other tasks stay blocked.

Critical Section
Semaphores

- A semaphore is a counter that can be used to synchronize multiple processes. Typically semaphores are used where multiple units of a particular resources are available

- Each semaphore has a counter value, which is a non-negative integer. It can take any value depending on number of resources available

- The ‘lock’ and ‘unlock’ mechanism is implemented via ‘wait’ and ‘post’ functionality in semaphore. Where the wait will decrement the counter and post will increment the counter

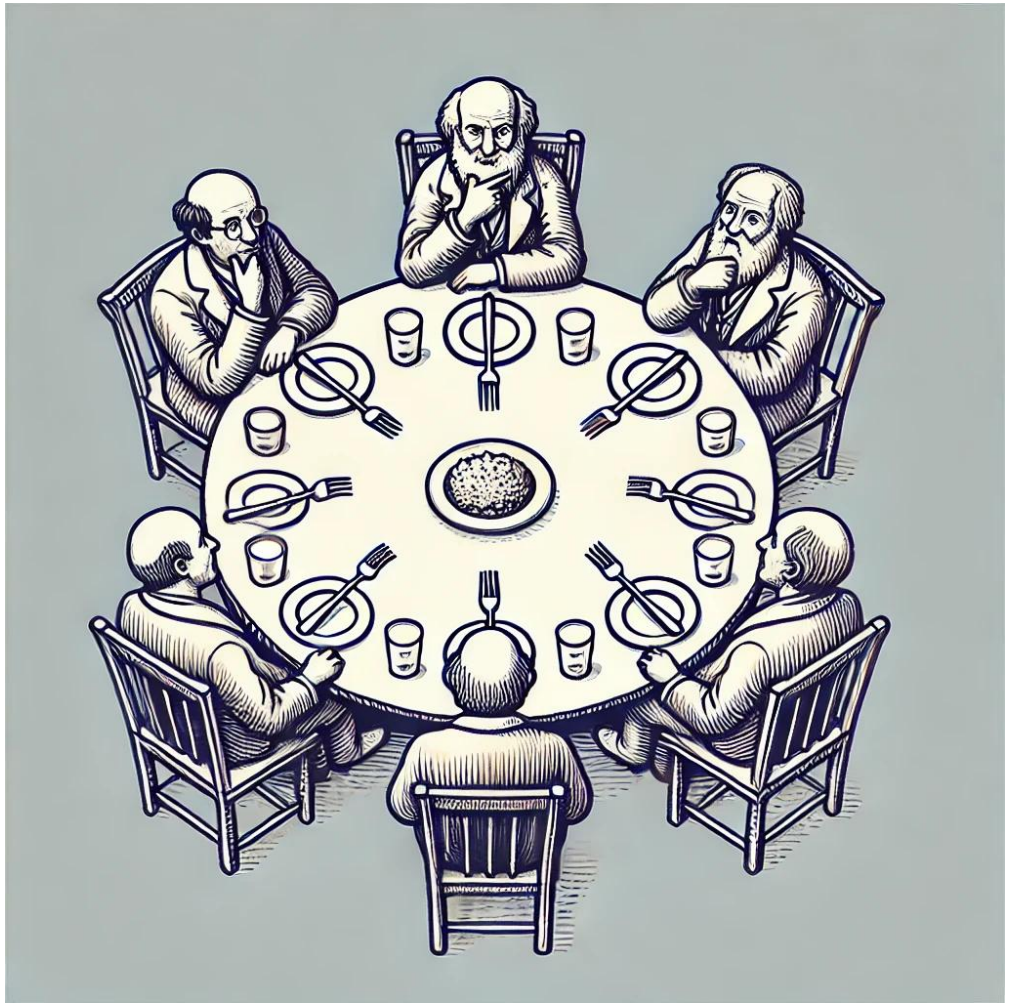
- When the counter value becomes zero that means the resources are no longer available hence remaining processes will get into blocked state

Critical Section
Sleeping barber problem

A lazy barber who sleeps and gets up on his own will :)

Critical Section
Dining philosophers’ problem

The **Dining Philosophers Problem** is a classical synchronization problem introduced by **Edsger Dijkstra** to illustrate **deadlock**, **resource sharing**, and **concurrency** in operating systems.



Critical Section

Semaphores – 2 basic operations

.Wait operation:

–Decrements the value of the semaphore by 1

–If the value is already zero, the operation blocks until the value of the semaphore becomes positive

–When the semaphore's value becomes positive, it is decremented by 1 and the wait operation returns

.Post operation:

–Increments the value of the semaphore by 1

–If the semaphore was previously zero and other threads are blocked in a wait operation on that semaphore

–One of those threads is unblocked and its wait operation completes (which brings the semaphore's value back to zero)

Critical Section
Mutex & Semaphores

- Semaphores which allow an arbitrary resource count (say 25) are called **counting semaphores**
- Semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available) are called **binary semaphores**
- A Mutex is essentially the same thing as a binary semaphore, however the differences between them are in how they are used
- While a binary semaphore may be used as a Mutex, a Mutex is a more specific use-case, in that only the process that locked the Mutex is supposed to unlock it
- This constraint makes it possible to implement some additional features in Mutexes

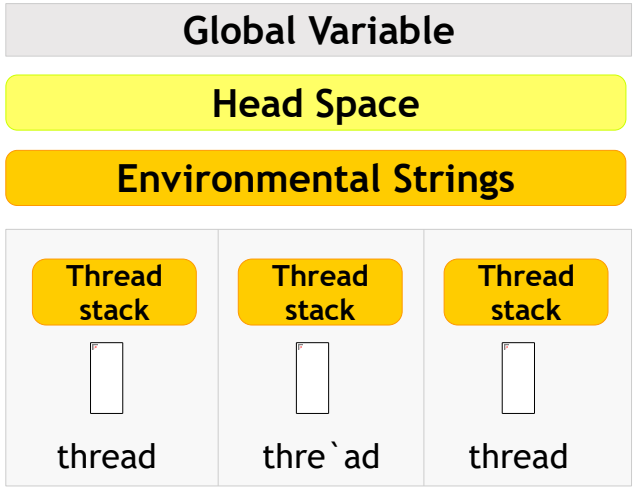
Critical Section

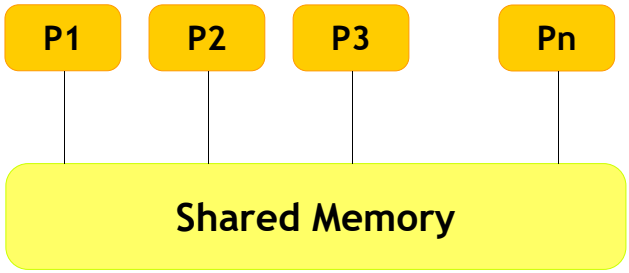
Practical Implementation

.The problem is critical section / race condition is common in multi-threading and multi-processing environment. Since both of them offer concurrency & common resource facility, it will raise to race conditions

.However the common resource can be different. In case of multiple threads a common resource can be a data segment / global variable which is a shared resource between multiple threads

.In case of multiple processes a common resource can be a shared memory





EMERTXE

Synchronization
Treads - Mutex

- pthread library offers multiple Mutex related library functions
- These functions help to synchronize between multiple threads

Function	Meaning
int pthread_mutex_init(pthread_mutex_t *mutex const pthread_mutexattr_t *attribute)	✓Initialize mutex variable ✓mutex: Actual mutex variable ✓attribute: Mutex attributes ✓RETUR: Success (0)/Failure (Non zero)
int pthread_mutex_lock(pthread_mutex_t *mutex)	✓Lock the mutex ✓mutex: Mutex variable ✓RETURN: Success (0)/Failure (Non-zero)
int pthread_mutex_unlock(pthread_mutex_t *mutex)	✓Unlock the mutex ✓Mutex: Mutex variable ✓RETURN: Success (0)/Failure (Non-zero)
int pthread_mutex_destroy(pthread_mutex_t *mutex)	✓Destroy the mutex variable ✓Mutex: Mutex variable ✓RETURN: Success (0)/Failure (Non-zero)

Synchronization

Treads – Semaphores – 2 basic operations

•Wait operation:

–Decrements the value of the semaphore by 1

–If the value is already zero, the operation blocks until the value of the semaphore becomes positive

–When the semaphore's value becomes positive, it is decremented by 1 and the wait operation returns

•Post operation:

–Increments the value of the semaphore by 1

–If the semaphore was previously zero and other threads are blocked in a wait operation on that semaphore

–One of those threads is unblocked and its wait operation completes (which brings the semaphore's value back to zero)

Synchronization
Treads - Semaphores

- pthread library offers multiple Semaphore related library functions
- These functions help to synchronize between multiple threads

Function	Meaning
int sem_init (sem_t *sem, int pshared, unsigned int value)	✓sem: Points to a semaphore object ✓pshared: Flag, make it zero for threads ✓value: Initial value to set the semaphore ✓RETURN: Success (0)/Failure (Non zero)
int sem_wait(sem_t *sem)	✓Wait on the semaphore **(Decrements count)** ✓sem: Semaphore variable ✓RETURN: Success (0)/Failure (Non-zero)
int sem_post(sem_t *sem)	✓Post on the semaphore **(Increments count)** ✓sem: Semaphore variable ✓RETURN: Success (0)/Failure (Non-zero)
int sem_destroy(sem_t *sem)	✓Destroy the semaphore ✓No thread should be waiting on this semaphore ✓RETURN: Success (0)/Failure (Non-zero)

Thank You