PTHREADS

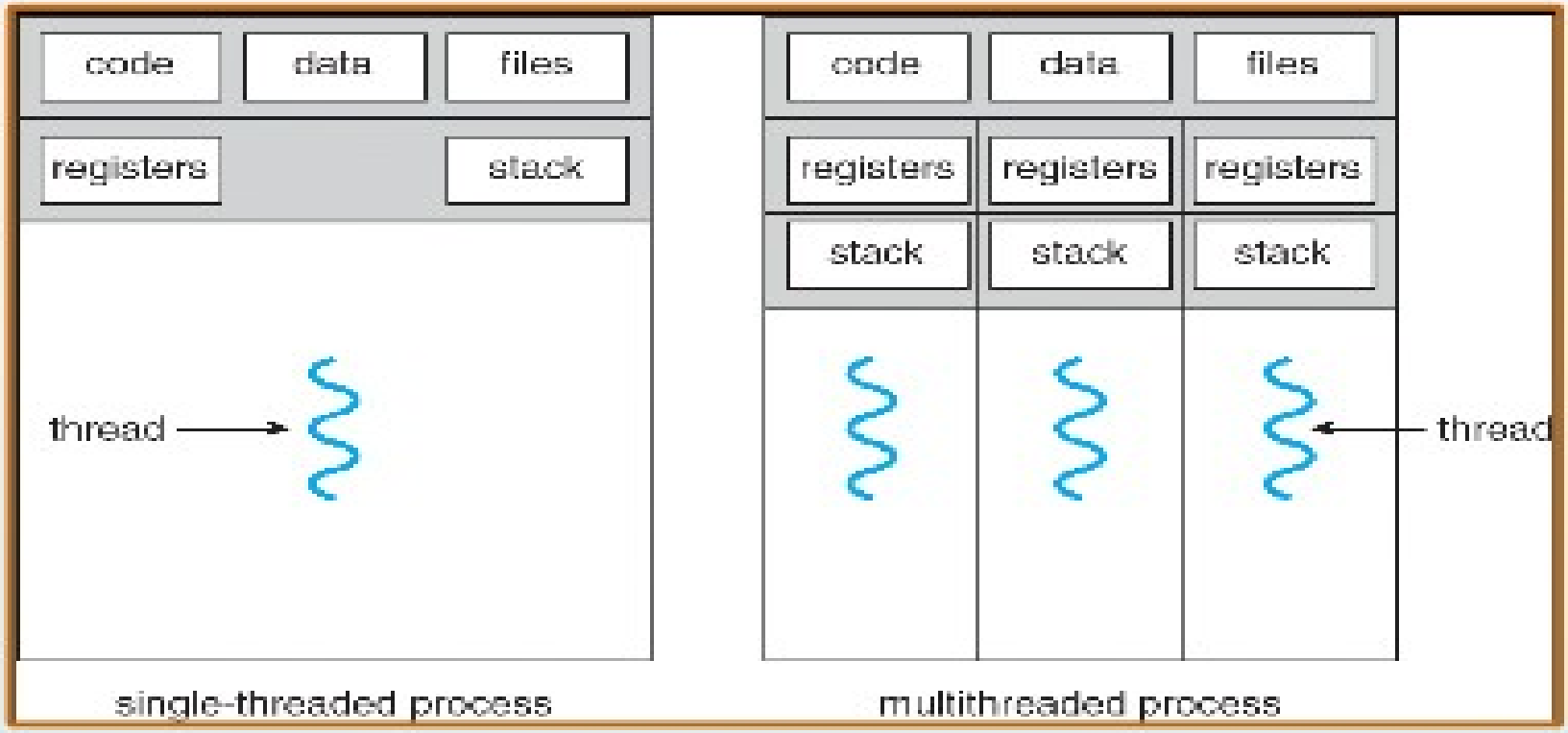
BY
MALLESH MEEROLLA

Basic information of thread

What is Thread?

- ‘Light Weight Process’
- Stream of instruction that can be scheduled as an independent unit.
- Exists within the process and uses or shares process resources.

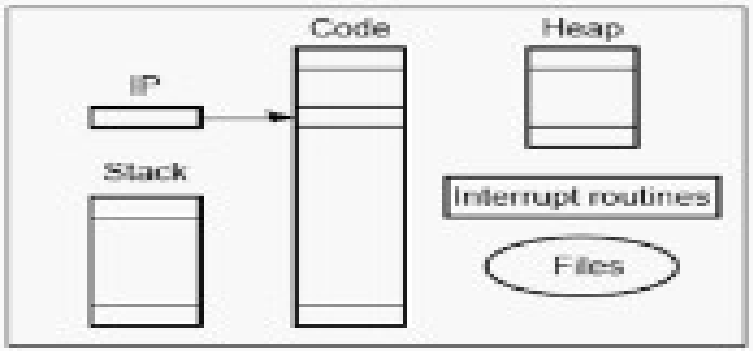
Single and Multithreaded Processes



Contd....

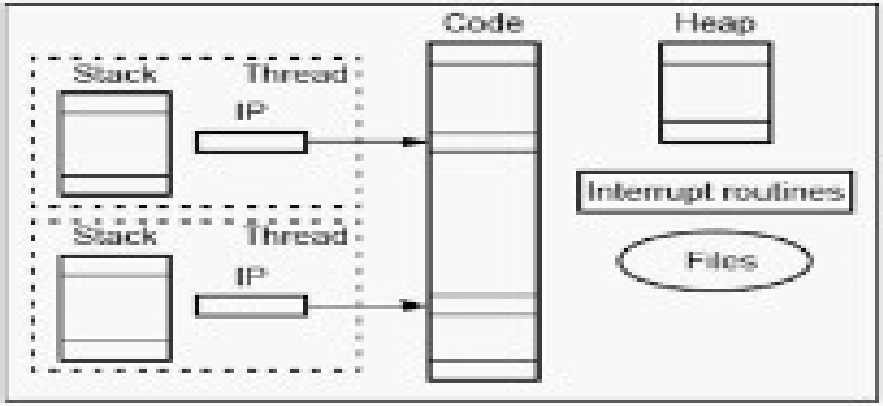
Threads vs Processes

"heavyweight" process - completely separate program with its own variables, stack, and memory allocation.



(a) Process

Threads - shares the same memory space and global variables between routines.



(b) Threads

Process vs Threads

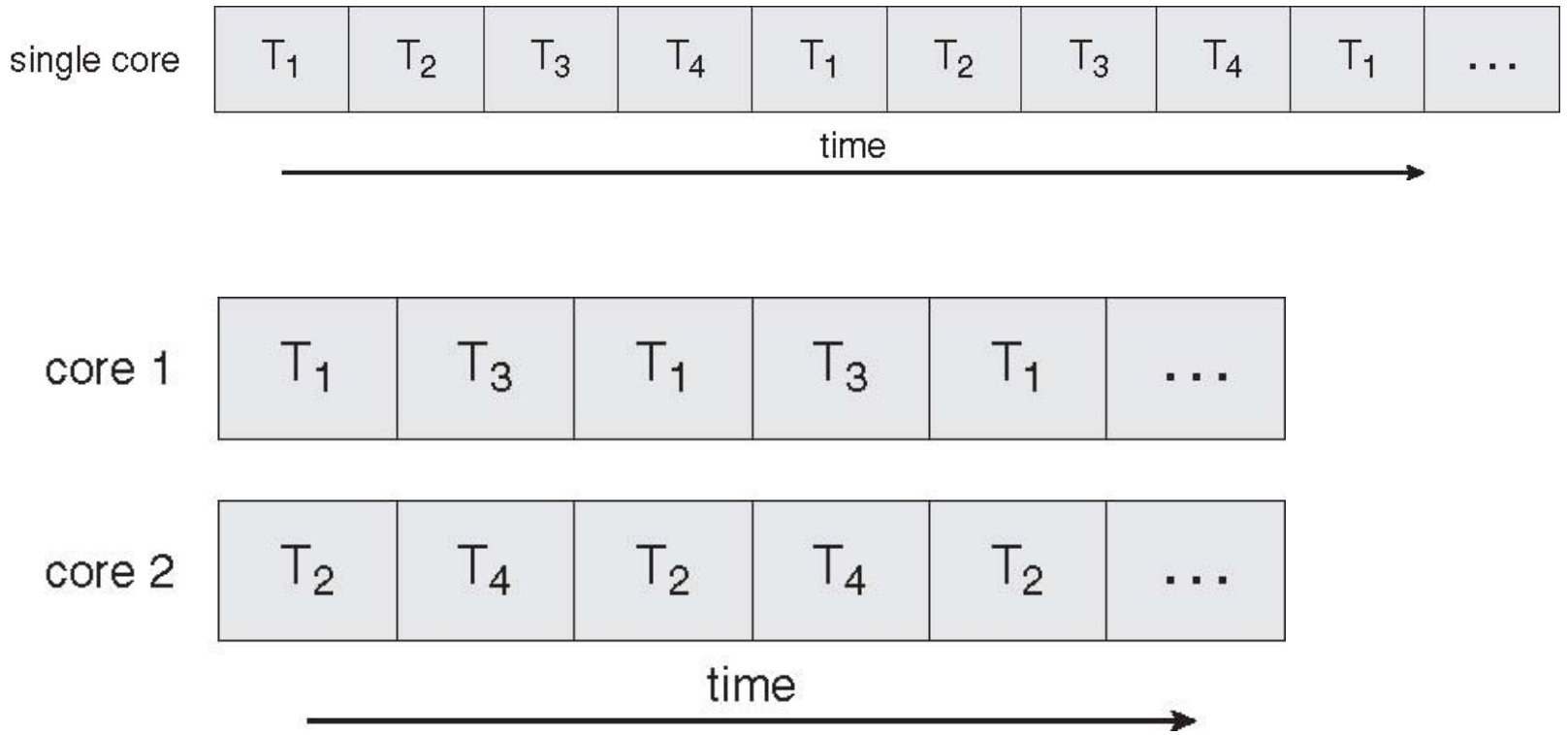
Advantages of Threads

- The overhead for creating a thread is significantly less than that for creating a process (~ 2 milliseconds for threads)
- switching between threads requires the OS to do much less work than switching between processes.
- multitasking, i.e., one process serves multiple clients.

Drawback of Threads

- Writing multithreaded programs require more careful thought
- More difficult to debug than single threaded programs
- For single processor machines, creating several threads in a program may not necessarily produce an increase in performance (only so many CPU cycles to be had)

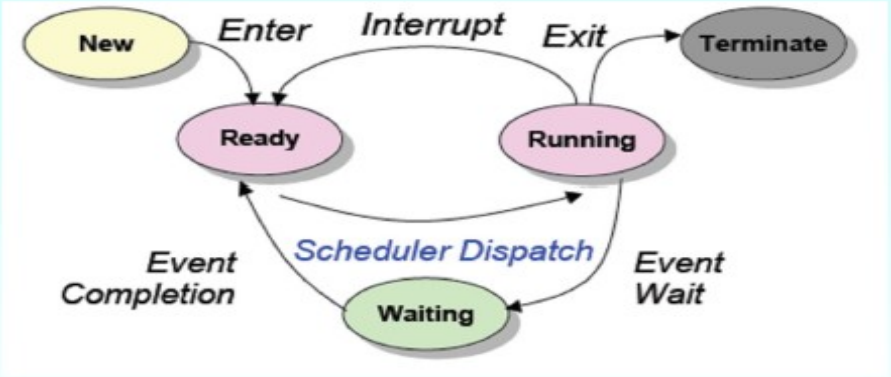
Concurrent Execution on a Single-core V/s Parallel Execution on a Multicore System



System overview of Threads……

State Diagram for a Thread

- Threads Creation
- Four Stages of Thread
  - Life Cycle
    - Ready
    - Running
    - Waiting (blocked)
    - Termination



Thread lifecycle

What is Pthreads or POSIX Threads?

What is Pthreads?

- Historically, hardware vendors have implemented their own proprietary versions of threads.
- Its difficult for programmers to develop portable threaded applications.
- So, A standardized programming interface was required.
- For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995). Implementations that adhere to this standard are referred to as POSIX threads, or Pthreads

Pthreads

- POSIX threads or Pthreads is a portable threading library which provides consistent programming interface across multiple operating systems.
- It is set of C language programming types and procedure calls , implemented with pthread.h file and a thread library.
- Set of threading interfaces developed by IEEE committee in charge of specifying a portable OS Interface.
- Library that has standardized functions for using threads across different platform.

Multi thread programs using pthread library

Simple thread program with pthread library api's

- Include thread library
  - #include <pthread.h>
- Create a thread(s) and assign sub programs in main program
  - pthread_create(thread, attr, start_routine, arg);
- Join threads (main thread wait for all threads to complete)
  - pthread_join(thread, status);
- Exit threads
  - pthread_exit(status);

Pthread API’s

Thread Creation

- **pthread_create**
  - When a new thread is created it runs concurrently with the creating process.
  - When creating a thread you indicate which function the thread should execute.
  - **1st argument**: Each thread is identified by a threadID of type pthread_t . Pointer to a pthread_t variable , in which threadID is stored.
  - **2nd argument**: Pointer to thread attribute object. If NULL is passed thread will be created with default thread attributes.see below slide for all possible thread attributes
  - **3rd argument**: pointer to the thread function. Ordinary function pointer of type void* (*) (void*).
  - **Last argument** value of type void*,passed as argument to the thread function.

Settable properties of thread attribute object

property	function
attribute objects	pthread_attr_destroy
	pthread_attr_init
detach state	pthread_attr_getdetachstate
	pthread_attr_setdetachstate
stack	pthread_attr_getguardsize
	pthread_attr_setguardsize
	pthread_attr_getstack
	pthread_attr_setstack
scheduling	pthread_attr_getinheritsched
	pthread_attr_setinheritsched
	pthread_attr_getschedparam
	pthread_attr_setschedparam
	pthread_attr_getschedpolicy

Thread exit and Thread join and etc

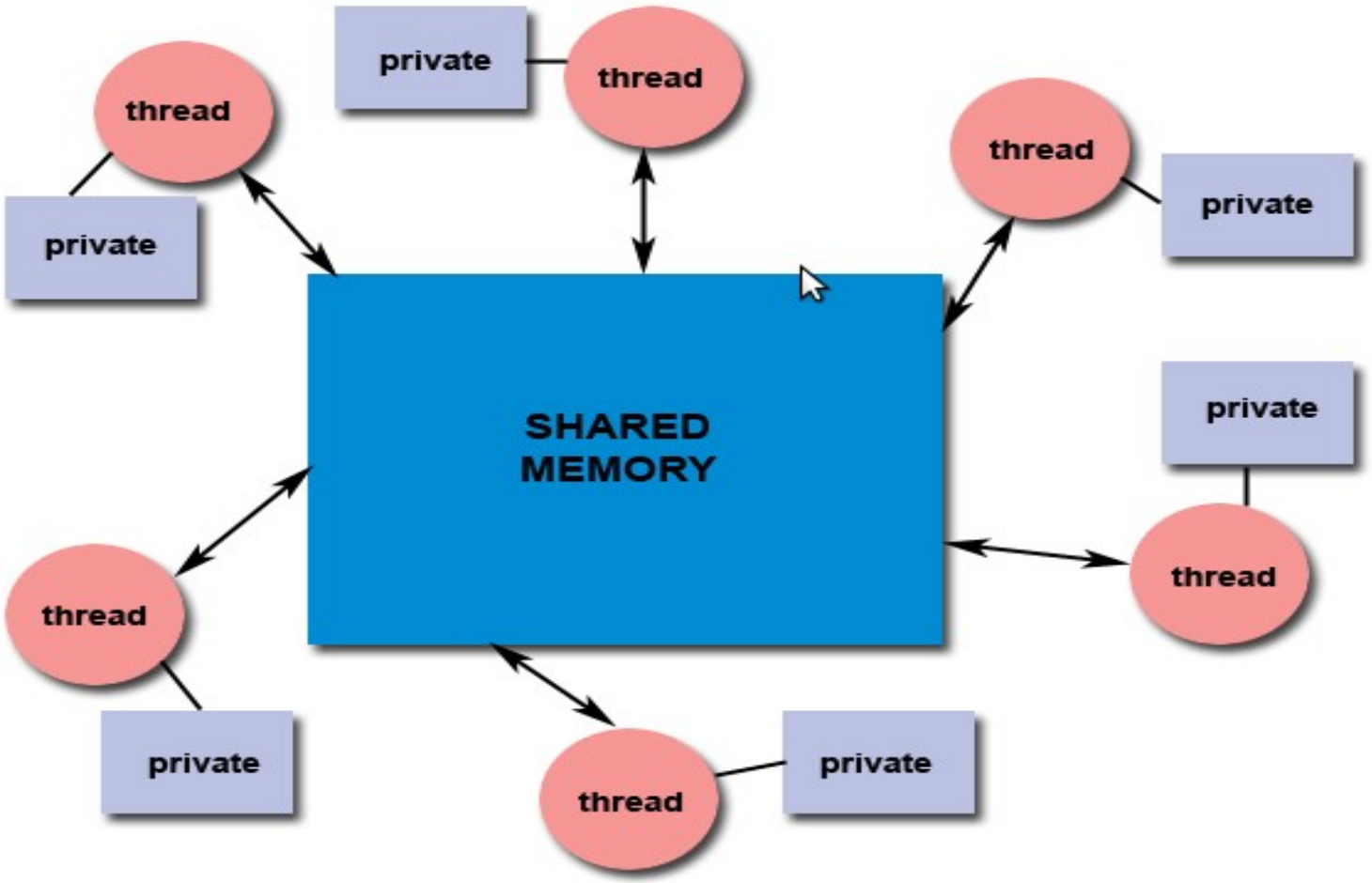
- pthread_join
  - The ThreadID of the thread to wait for
  - Pointer to the void* variable that will receive the finished thread’s return value.
  - Failure to join threads memory and other resource leaks until the process ends
- pthread_exit
  - Thread’s return value.
  - A thread can explicitly exit by using this API.
- pthread_cancel
  - One thread can request that another exit with pthread_cancel
  - int pthread_cancel(pthread_t thread);
- pthread_self
  - Returns the thread identifier for the calling thread

Single thread program vs multi thread program

```c
// example single threaded program

#include<stdio.h>

// function
void  printmsg(char *msg)
{
        printf("%s",msg);
        return;

}

int main()
{

printmsg("helloworld\n");   //subprogram1

printmsg("byeworld\n");     //subprogram2

return 0;

}
```

```c
//example multi threaded program

#include<pthread.h>   // to use apis provided by Pthreads library
#include<stdio.h>

// function
void  * printmsg(char *msg)
{
        printf("%s",msg);
        return;
}

int main()
{

//declare 2 child threads
pthread_t pthread1,pthread2;

/* To initializes
the thread,
it's attributes,
the address of the routine the thread has to start executing,
the parameters for that routine.
*/
pthread_create(&pthread1,NULL,printmsg,(void*)"helloworld");
pthread_create(&pthread2,NULL,printmsg,(void*)"byeworld");

//Just as pthread_create() splits our single thread into two threads, pthread_join()
merges two threads into a single thread.
pthread_join(pthread1,NULL);
pthread_join(pthread2,NULL);

// terminate thread
pthread_exit("thank u");

printf("\n");

return 0;
}
```

Racecondition and synchronization



Race condition

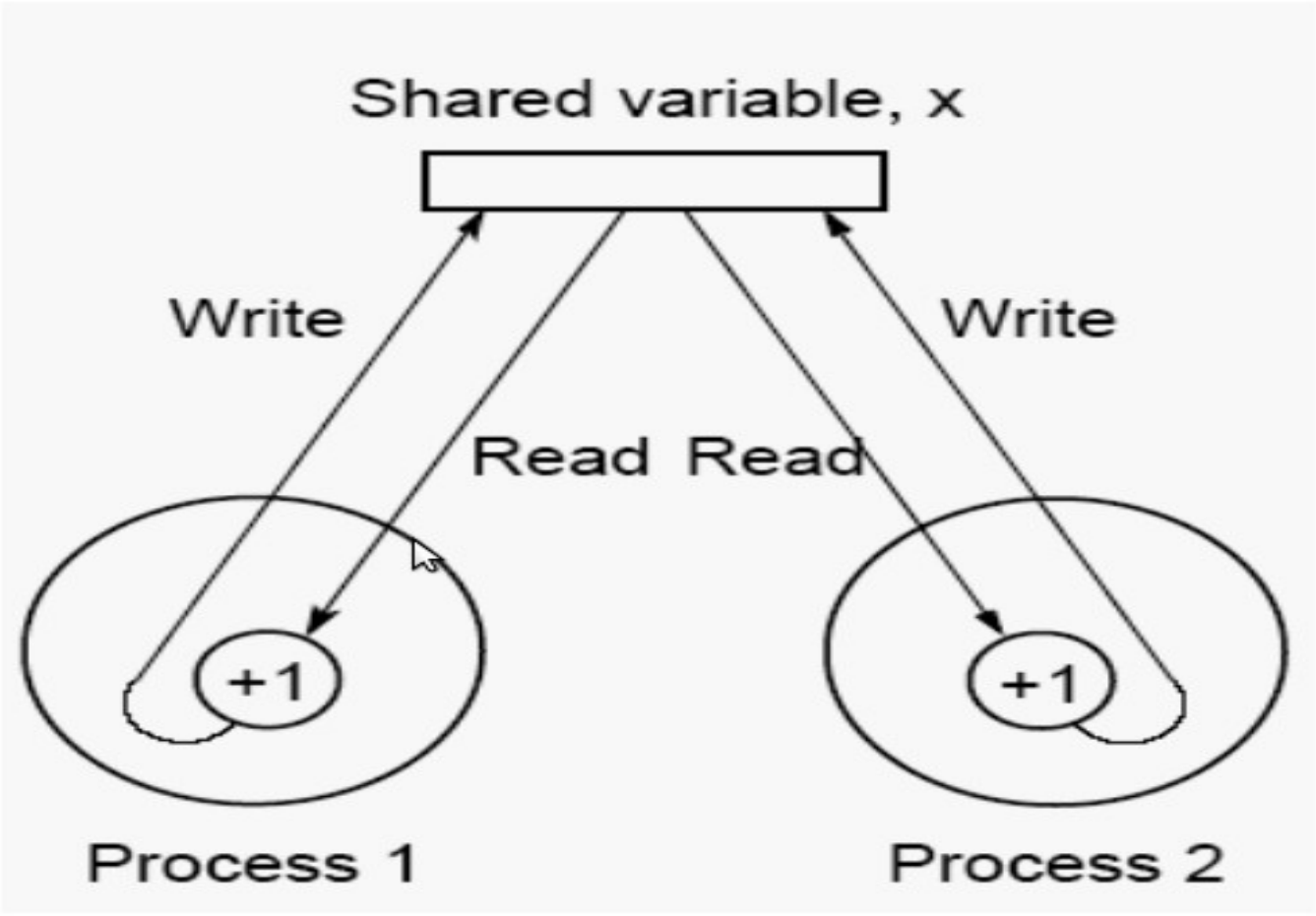
Accessing Shared Data

- The programmer must carefully control access to shared data.
- Consider two processes which both increment the variable X:

Instruction	Process 1	Process 2
`x = x + 1;`	`read x`	`read x`
	`compute x + 1`	`compute x + 1`
	`write to x`	`write to x`

Time

Race Condition



Race Condition

Critical Sections

- Critical sections provide a way to make sure only one process accesses a resource at a time.
- The critical section is the code that accesses the resource. They must be arranged such that only one section can be executed at a time.
- This mechanism is known as mutual exclusion.

Race Condition

Locks

- A lock is the simplest mechanism for ensuring mutual exclusion of critical sections.
- It can only have two values:
  - 1 – a process has entered the critical section.
  - 0 – there is no process in the critical section.
- To enter the critical section you must acquire the lock.
- On leaving the critical section you release the lock. This is done by changing the value back to 0.

Race Condition

Acquiring the Lock

- It must be guaranteed that only one thread can acquire the lock.
- This C code is not adequate:

```
if (lock == 0) {
    // lock is free
    lock = 1;
}
```

- Multiple threads could be executing the if statement simultaneously.

Race Condition

Pthread synchronization API's..(mutex)

- **pthread_mutex_init():**
  - pthread_mutex_init(mutex , attr)
  - initializes the mutex and sets its attributes
- **pthread_mutex_destroy():**
  - pthread_mutex_destroy(mutex)
  - destroy the mutex
- **pthread_mutex_lock():**
  - pthread_mutex_lock(mutex)
  - Locks a mutex.
  - If the mutex is already locked, the calling thread blocks until the mutex becomes available.
- **pthread_mutex_trylock():**
  - pthread_mutex_trylock(mutex)
  - Tries to lock a Mutex. If the mutex object referenced by mutex is currently locked by any thread, the call returns immediately.

Mutex

- **pthread_mutex_unlock():**
  - pthread_mutex_unlock(mutex):
  - Unlocks a Mutex.

Synchronization with semaphore

- **sem_t sem;**
  - Declare semaphore variable
- **sem_init(&sem,0,1);**
  - Initialize and Set the semaphore value
- **sem_wait(&sem);**
  - Lock semaphore
- **sem_post(&sem);**
  - Unlock semaphore

Mutex

Conditional variablea

- A condition variable is a variable of type pthread_cond_t
- It is used with the appropriate functions for waiting and later, process continuation.
- The condition variable mechanism allows threads to suspend execution and relinquish the processor until some condition is true.
- A condition variable must always be associated with a mutex to avoid a race condition created by one thread preparing to wait and another thread which may signal the condition before the first thread actually waits on it resulting in a deadlock.
- The thread will be perpetually waiting for a signal that is never sent. Any mutex can be used, there is no explicit link between the mutex and the condition variable

Creating and Destroying Condition Variables

- Creating/Destroying:
  - pthread_cond_init (dynamic way)
  - pthread_cond_t cond = PTHREAD_COND_INITIALIZER; (static way)
  - pthread_cond_destroy
- Waiting on condition:
  - pthread_cond_wait - unlocks the mutex and waits for the condition variable cond to be signaled.
  - pthread_cond_timedwait - place limit on how long it will block.
- Waking thread based on condition:
  - pthread_cond_signal - restarts one of the threads that are waiting on the condition variable cond.
  - pthread_cond_broadcast - wake up all threads blocked by the specified condition variable.