5th Seminar

Process

- The execution of a program that allows to perform the appropriate actions specified in a program. It can be defined as an execution unit where a program runs. The OS helps to create, schedule, and terminate the processes which is used by CPU. The other processes created by the main process are called children processes.
- Properties:

- creation requires separate system calls for each process.
- is an isolated execution entity and does not share data and information.
- use the IPC mechanism for communication that significantly increases the number of system calls.
- process management takes more system calls.
- has its stack, heap memory with memory, and data map.

Thread

- An execution unit that is part of a process. A process can have multiple threads, all executing at the same time. It is a unit of execution in concurrent programming. A thread is lightweight and can be managed independently by a scheduler. It helps improve the application performance using parallelism.
- Multiple threads share information like data, code, files, etc. They can be implemented in three different ways:

- Kernel-level threads
- User-level threads
- Hybrid threads

Threads

- Properties of Threads:

- Single system call can create more than one thread
- Threads share data and information.
- Threads share instructions, global, and heap regions. However, they have their own registers and stack.
- Thread management consumes very few, or no system calls because of communication between threads that can be achieved using shared memory.

A thread is a path of execution within a process. A process can contain multiple threads.

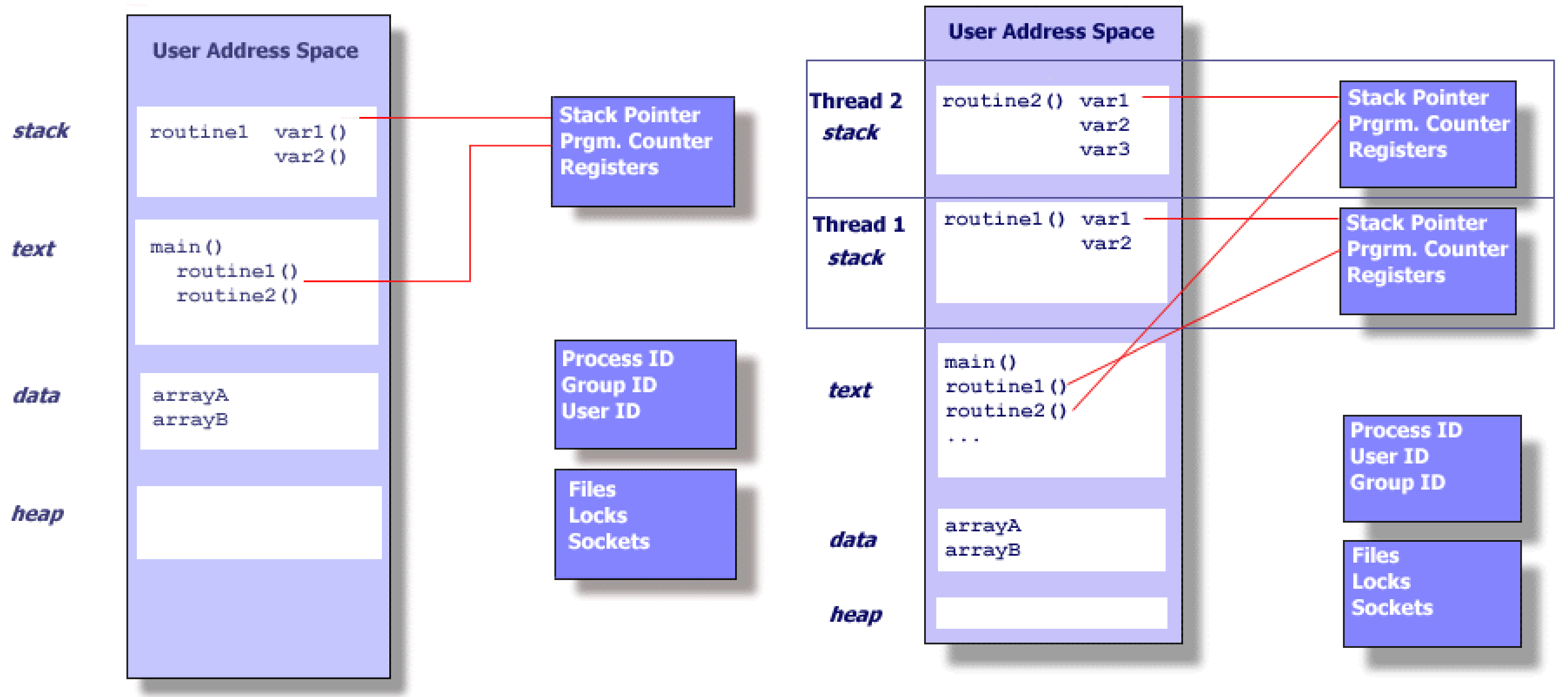
Why Multithreading?

- A thread is also known as lightweight process.
- The idea is to achieve parallelism by dividing a process into multiple threads.
- Examples:

- in a browser, multiple tabs can be different threads.
- MS Word uses multiple threads: one thread to format the text, another thread to process inputs

Process vs Thread

- The main difference is that threads within the same process run in a shared memory space, while processes run in separate memory spaces.
- Threads are not independent of one another like processes are, and as a result threads share with other threads their code section, data section, and OS resources (like open files and signals).
- Like a process, a thread has its own program counter (PC), register set, and stack space.


User Address Space
stack
routine1 var1() var2()
text
main() routine1() routine2()
data
arrayA arrayB
heap
Stack Pointer Prgm. Counter Registers
Process ID Group ID User ID
Files Locks Sockets
User Address Space
Thread 2 stack
routine2() var1 var2 var3
Thread 1 stack
routine1() var1 var2
text
main() routine1() routine2() ...
data
arrayA arrayB
heap
Stack Pointer Prgrm. Counter Registers
Stack Pointer Prgrm. Counter Registers
Process ID User ID Group ID
Files Locks Sockets

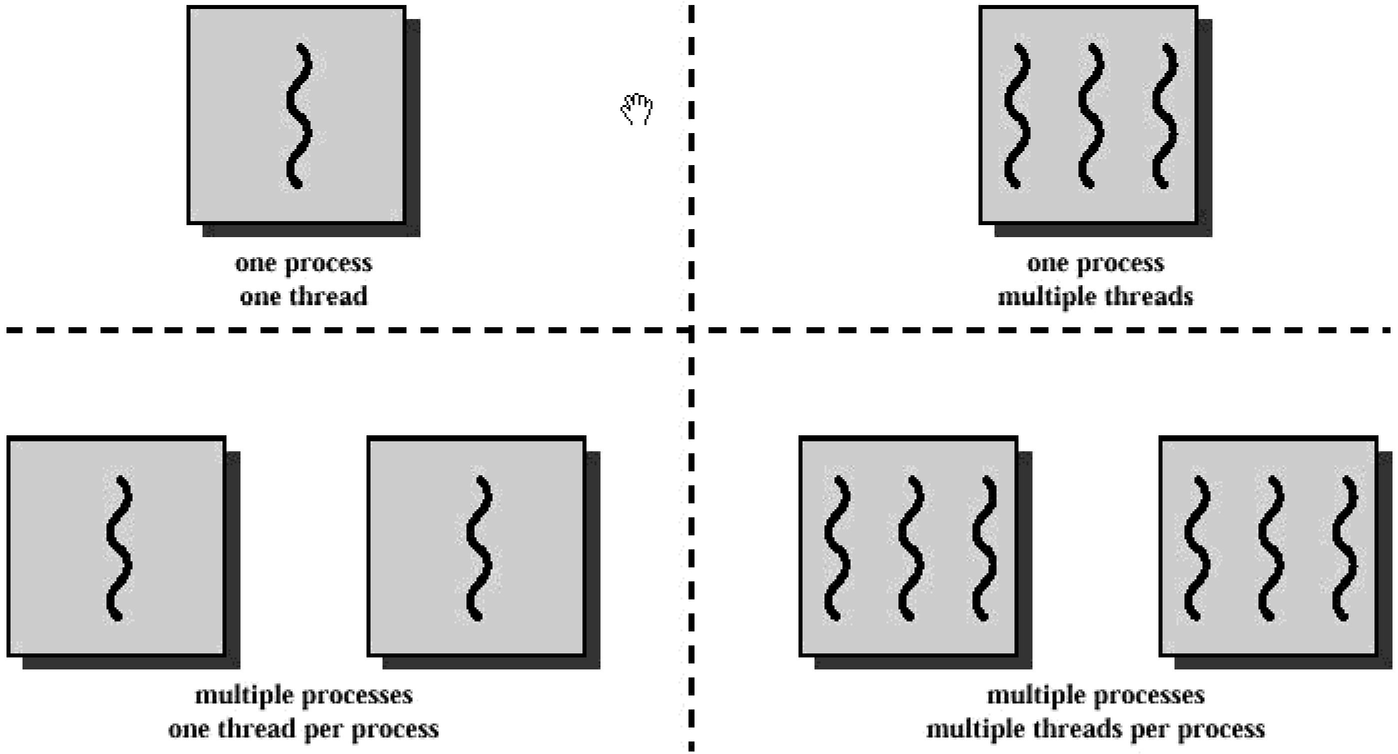
Advantages of Thread over Process

1. Responsiveness: If the process is divided into multiple threads, if one thread completes its execution, then its output can be immediately returned.
2. Faster context switch: Context switch time between threads is lower compared to processes context switch. Processes context switching requires more overhead from the CPU.
3. Effective utilization of multiprocessor system: If we have multiple threads in a single process, then we can schedule multiple threads on multiple processors. This will make process execution faster.
4. Resource sharing: Resources like code, data, and files can be shared among all threads within a process.

Note: stack and registers can’t be shared among the threads. Each thread has its own stack and registers.

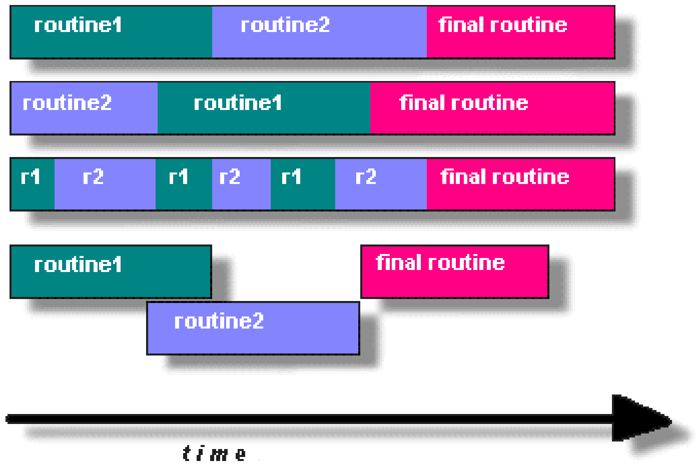
Advantages of Thread over Process

5. Communication: Communication between multiple threads is easier, as the threads share common address space, while in processes we have to follow some specific communication technique for communication between two processes.
6. Enhanced throughput of the system: If a process is divided into multiple threads, and each thread function is considered as one job, then the number of jobs completed per unit of time is increased, thus increasing the throughput of the system.


one process
one thread
one process
multiple threads
multiple processes
one thread per process
multiple processes
multiple threads per process

Multithreading

- in order for a program to take advantage of multithreading, it must be able to be organized into discrete, independent tasks which can execute concurrently. For example, if routine1 and routine2 can be interchanged, interleaved and/or overlapped in real time, they are candidates for threading.



POSIX Threads

- The functions part of the Pthreads API can be grouped into:
- Thread management: work directly on threads - creating, detaching, joining, etc
- Mutexes: deal with synchronization using "mutual exclusion". Mutex functions provided for creating, destroying, locking and unlocking mutexes.
- Condition variables: address communications between threads that share a mutex. Based on runtime conditions. Includes functions to create, destroy, wait and signal based upon specified variable values.
- Synchronization: manage read/write locks and barriers.

POSIX Threads

- The POSIX Threads Library:libpthread, <pthread.h>

Creating a (Default) Thread

- Uses the function pthread_create() to add a new thread of control to the current process. It is prototyped by:

```
int pthread_create(pthread_t *tid, const pthread_attr_t
*tattr, void*(*start_routine)(void *), void *arg);
```

Thread Creation

- The pthread_create() function is called with attr having the necessary state behavior. start_routine is the function with which the new thread begins execution. When start_routine returns, the thread exits with the exit status set to the value returned by start_routine.
- When pthread_create is successful, the ID of the thread created is stored in the location referred to as tid.
- Creating a thread using a NULL attribute argument has the same effect as using a default attribute; both create a default thread. When tattr is initialized, it acquires the default behavior.
- pthread_create() returns a zero and exits when it completes successfully. Any other returned value indicates that an error occurred.

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}
```

```
int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0; t<NUM_THREADS; t++){
        printf("In main: creating thread %ld\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void
*)t);
        if (rc){
        printf("ERROR; code from pthread_create() is %d\n", rc);
        exit(-1);
        }
   }
   /* Last thing that main() should do */
   pthread_exit(NULL); // or return 0;
}
```

Waiting for Thread Termination

- Use pthread_join function to wait for a thread to terminate. It is prototyped by:

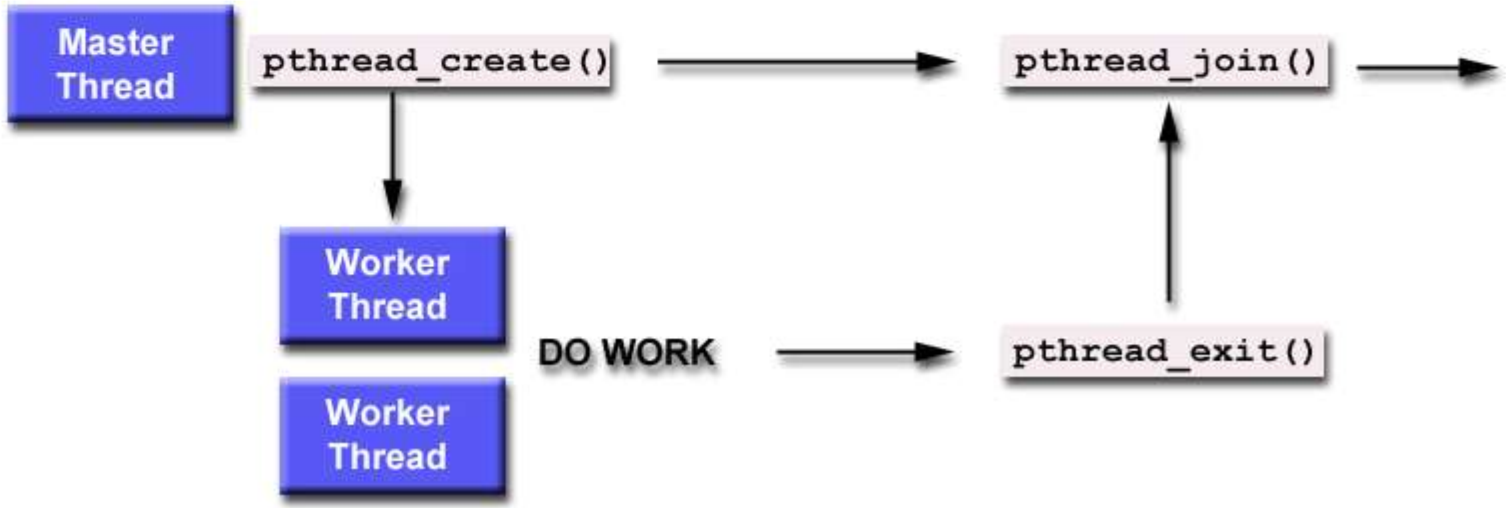
```
int pthread_join(thread_t tid, void **status);

#include <pthread.h>
pthread_t tid;
int ret;
int status;
/* waiting to join thread "tid" with status */
ret = pthread_join(tid, &status);
/* waiting to join thread "tid" without status */
ret = pthread_join(tid, NULL);
```

Waiting for Thread Termination

- The `pthread_join()` function blocks the calling thread until the specified thread terminates. The specified thread must be in the current process and must not be detached. When status is not NULL, it points to a location that is set to the exit status of the terminated thread when pthread_join() returns successfully.
- Multiple threads cannot wait for the same thread to terminate. If they try to, one thread returns successfully and the others fail with an error of ESRCH.
- After `pthread_join()` returns, any stack storage associated with the thread can be reclaimed by the application.

Waiting for Thread Termination



Thread modes

- A Thread can run in two modes:
  - Joinable Mode: can be joined
  - Detached Mode: can never be joined

Joinable Mode

- By default a thread runs in joinable mode. Joinable thread will not release any resource even after the end of thread function, until some other thread calls pthread_join() with its ID.

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **retval);
```

- pthread_join() is a blocking call, it will block the calling thread until the other thread ends. First parameter of pthread_join() is the ID of target thread.
        Second parameter of pthread_join() is address of (void *) i.e. (void **), it will point to the return value of thread function i.e. pointer to (void *).

Detached Mode

- A Detached thread automatically releases it allocated resources on exit. No other thread needs to join it. But by default all threads are joinable, so to make a thread detached we need to call pthread_detach() with thread id

```
#include <pthread.h>
int pthread_detach(pthread_t thread);

int err = pthread_detach(threadId);
```

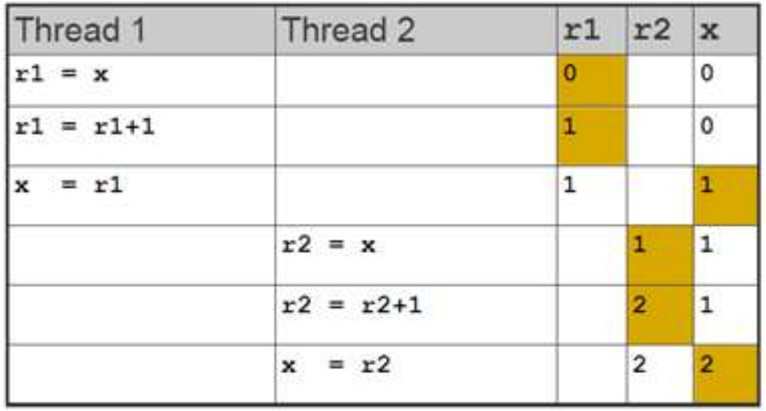
- A detached thread automatically release the resources on exit, therefore there is no way to determine its return value of detached thread function.
- pthread_detach() will return non zero value in case of error.

Pthread Mutex

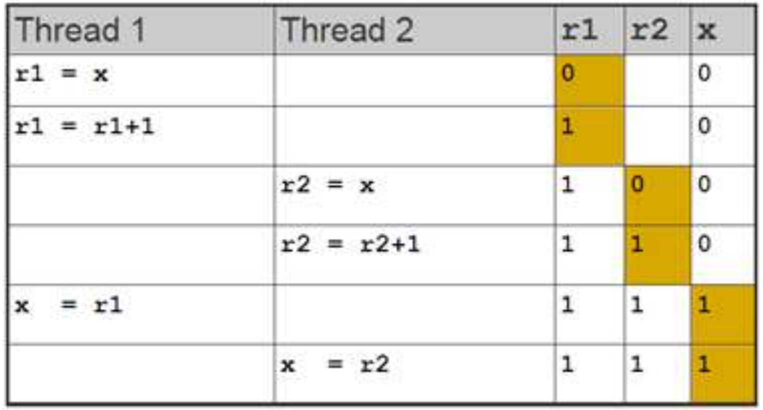
- Mutex stands for "mutual exclusion". Mutex are global variables used for implementing thread synchronization and protecting shared data for concurrent access.
- A mutex variable acts like a "lock" protecting access to a shared data resource. The basic concept of a mutex as used in Pthreads is that only one thread can lock (or own) a mutex variable at any given time. Thus, even if several threads try to lock a mutex only one thread will be successful. No other thread can own that mutex until the owning thread unlocks that mutex. Threads must "take turns" accessing protected data.
- Mutexes can be used to prevent "race" conditions.

Pthread Mutex

- Suppose we have a global variable `int x=0;`
- And two threads, each performing: `x++;`
- What is the final value of x?



Thread 1	Thread 2	r1	r2	x
`r1 = x`		0		0
`r1 = r1+1`		1		0
`x  = r1`		1		1
	`r2 = x`		1	1
	`r2 = r2+1`		2	1
	`x  = r2`		2	2



Thread 1	Thread 2	r1	r2	x
`r1 = x`		0		0
`r1 = r1+1`		1		0
	`r2 = x`	1	0	0
	`r2 = r2+1`	1	1	0
`x  = r1`		1	1	1
	`x  = r2`	1	1	1

Mutex

- Usually the action performed by a thread owning a mutex is to update a global variable. This is a way to ensure that when several threads update the same variable, the final value is the same as it would be if only one thread performed the update. The variables being updated belong to a "critical section".
- A typical sequence in the use of a mutex is as follows:

- Create and initialize a mutex variable
- Several threads attempt to lock the mutex
- Only one succeeds and that thread owns the mutex
- The owner thread performs some set of actions
- The owner unlocks the mutex
- Another thread acquires the mutex and repeats the process
- Finally the mutex is destroyed

Mutex

- When several threads compete for a mutex, the losers block at that call - an unblocking call is available with "trylock" instead of the "lock" call.
- When protecting shared data, it is the programmer's responsibility to make sure every thread that needs to use a mutex does so. For example, if 4 threads are updating the same data, but only one uses a mutex, the data can still be corrupted.

Mutex

- Mutex variables must be declared of type pthread_mutex_t, and must be initialized before usage. There are two ways to initialize a mutex variable:
- Statically, when it is declared. For example:

```
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
```

- Dynamically, with the `pthread_mutex_init()` routine. This method permits setting mutex object attributes, attr.
- The mutex is initially unlocked.
- The attr object is used to establish properties for the mutex object, and must be of type pthread_mutexattr_t if used (may be specified as NULL to accept defaults).

Mutex

- The Pthreads standard defines three optional mutex attributes:
  - Protocol: Specifies the protocol used to prevent priority inversions for a mutex.
  - Prioceiling: Specifies the priority ceiling of a mutex.
  - Process-shared: Specifies the process sharing of a mutex.
- Note that not all implementations may provide the three optional mutex attributes.
- The pthread_mutexattr_init() and pthread_mutexattr_destroy() routines are used to create and destroy mutex attribute objects respectively.
- `pthread_mutex_destroy()` should be used to free a mutex object which is no longer needed.

Mutex

- The `pthread_mutex_lock()` is used by a thread to acquire a lock on the specified mutex variable. If the mutex is already locked by another thread, this call will block the calling thread until the mutex is unlocked.
- `pthread_mutex_trylock()` will attempt to lock a mutex. If the mutex is already locked, the routine will return immediately with a "busy" error code. This routine may be useful in preventing deadlock conditions, as in a priority-inversion situation.
- `pthread_mutex_unlock()` will unlock a mutex if called by the owning thread. Required after a thread has completed its use of protected data if other threads are to acquire the mutex. An error will be returned if:

- If the mutex was already unlocked
- If the mutex is owned by another thread

Mutex

```
int x;
pthread_mutex_t m=PTHREAD_MUTEX_INITIALIZER;
void *f(void *arg){
        int v;
        v=*(int *)arg;
        pthread_mutex_lock(&m);
        x++;
        pthread_mutex_unlock(&m);
        printf("Suntem in thread, am primit argument %d\n", v);
        return NULL;
}
```

Mutex

```
int main(){
        pthread_t t;
        pthread_create(&t, NULL, f, (void *)&t);
        pthread_mutex_lock(&m);
        x++;
        pthread_mutex_unlock(&m);
        printf("Am creat un thread cu id-ul %d\n",t);
        pthread_join(t,NULL);
        printf("S-a terminat threadul, main se termina, x: %d\n",x);
        pthread_mutex_destroy(&m);
        return 0;
}
```

Conditional Variables

Condition variables provide yet another way for threads to synchronize. While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data. Without condition variables, the programmer would need to have threads continually polling (possibly in a critical section), to check if the condition is met. This can be very resource consuming since the thread would be continuously busy in this activity. A condition variable is a way to achieve the same goal without polling.
A condition variable is always used in conjunction with a mutex lock.

Conditional Variables

Main Thread

Declare and initialize global data/variables which require synchronization (such as "count")
Declare and initialize a condition variable object
Declare and initialize an associated mutex
Create threads A and B to do work

Thread A

Do work up to where a certain condition must occur (such as "count" must reach a specified value)
Lock associated mutex and check value of a global variable
Call pthread_cond_wait() to perform a blocking wait for signal from Thread-B. The call to pthread_cond_wait() automatically and atomically unlocks the associated mutex variable so that it can be used by Thread-B.
When signalled, wake up. Mutex is automatically locked.
Explicitly unlock mutex
Continue

Thread B

Do work
Lock associated mutex
Change the value of the global variable that Thread-A is waiting upon.
Check value of the global Thread-A wait variable. If it fulfills the desired condition, signal Thread-A.
Unlock mutex.
Continue

Main Thread

Join / Continue

Conditional Variables

- Condition variables must be declared with type pthread_cond_t, and must be initialized before they can be used. There are two ways to initialize a condition variable:

- Statically, when it is declared:

```
pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER;
```

- Dynamically, with the `pthread_cond_init()` routine. The ID of the created condition variable is returned to the calling thread through the condition parameter.

- `pthread_cond_destroy()` should be used to free a condition variable that is no longer needed.

Condition Variables

- `pthread_cond_wait()` blocks the calling thread until the specified condition is signalled.
  - Should be called while mutex is locked, and it will automatically release the mutex while it waits.
  - After signal is received and thread is awakened, mutex will be automatically locked for use by the thread. The programmer is then responsible for unlocking mutex when the thread is finished with it.
- Recommendation: Using a WHILE loop instead of an IF statement to check the waited for condition can help deal with several potential problems, such as:
  - If several threads are waiting for the same wake up signal, they will take turns acquiring the mutex, and any one of them can then modify the condition they all waited for.

Condition Variables

- The `pthread_cond_signal()` routine is used to signal (or wake up) another thread which is waiting on the condition variable. It should be called after mutex is locked, and must unlock mutex in order for `pthread_cond_wait()` routine to complete.
- The `pthread_cond_broadcast()` routine should be used instead of `pthread_cond_signal()` if more than one thread is in a blocking wait state.
- It is a logical error to call `pthread_cond_signal()` before calling `pthread_cond_wait()`.

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS  3
#define TCOUNT 10
#define COUNT_LIMIT 12

int     count = 0;
int     thread_ids[3] = {0,1,2};
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;
```

```
void *inc_count(void *t)
{
  int i;
  long my_id = (long)t;
  for (i=0; i<TCOUNT; i++) {
        pthread_mutex_lock(&count_mutex);
        count++;
        if (count == COUNT_LIMIT) {
          pthread_cond_signal(&count_threshold_cv);
          printf("thread %ld,count=%d Threshold reached.\n",my_id, count);
        }
        printf("thread %ld, count = %d, unlocking mutex\n",my_id, count);
        pthread_mutex_unlock(&count_mutex);
        /* Do some "work" so threads can alternate on mutex lock */
        sleep(1);
        }
  pthread_exit(NULL);
}
```

```
void *watch_count(void *t)
{
  long my_id = (long)t;

  printf("Starting watch_count(): thread %ld\n", my_id);

  pthread_mutex_lock(&count_mutex);
  while (count<COUNT_LIMIT) {
        pthread_cond_wait(&count_threshold_cv, &count_mutex);
        printf("thread %ld Condition signal received.\n", my_id);
        }
        count += 125;
        printf("thread %ld count now = %d.\n", my_id, count);
  pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}
```

```
int main (int argc, char *argv[])
{
  int i, rc;
  long t1=1, t2=2, t3=3;
  pthread_t threads[3];

  /* Initialize mutex and condition variable objects */
  pthread_mutex_init(&count_mutex, NULL);
  pthread_cond_init (&count_threshold_cv, NULL);
  pthread_create(&threads[0], NULL, watch_count, (void *)t1);
  pthread_create(&threads[1], NULL, inc_count, (void *)t2);
  pthread_create(&threads[2], NULL, inc_count, (void *)t3);

  /* Wait for all threads to complete */
  for (i=0; i<NUM_THREADS; i++) {
       pthread_join(threads[i], NULL);
  }
  printf ("Main(): Waited on %d  threads. Done.\n", NUM_THREADS);

  /* Clean up and exit */
  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_threshold_cv);
  pthread_exit(NULL);

}
```

RW Locks

- When sharing resources between multiple threads, it is safe to allow multiple readers to access the resource simultaneous as they would not alter any data but only one writer should be allowed to access the resource at any given time.
- The standard mutex, pthread_mutex_t, allows only one process to access the resource irrespective of whether the process is a reader or writer. The behavior even though safe, is not efficient, as the readers are made to wait even though they are not going to modify the data.
- To allow multiple readers to access a resource at the same time one can use pthread_rwlock_t. The pthread_rwlock_t (read,write lock) allows multiple readers to access the resource, but allows only one writer at any given time.
- A read/write lock variable has 3 values (states): Unlocked, Read Locked, Write Locked

RW Locks

Read Lock

- If the read/write lock is in the unlocked state, the read lock will complete (and the thread continues with the next instruction following the read lock command)
- The value (state) of the read/write lock is changed to read locked
- If the read/write lock is in the read locked state, the thread that executes the read lock command complete (and the thread continues with the next instruction following the read lock command).
- The value (state) of the read/write lock remains read locked, but a count is increased (so we know how many times a read lock operation has been performed)
- If the read/write lock is in the write locked state, the thread that executes the read lock command will block until the value (state) of the read/write lock becomes unlocked
- (When the state of the read/write lock does become unlocked, the read lock command will complete and change the state of the read/write lock to read locked)

RW Locks

Defining a read/write lock variable in Pthreads:

```
        pthread_rwlock_t   x;
```

Initializing a read/write lock variable:

After defining the read/write lock variable, it must be initialized:

```
        int pthread_rwlock_init(pthread_rwlock_t  *rwlock,
pthread_rwlockattr_t *attr);
```

rwlock: is the read/write lock that you want to initialize (pass the address !)

attr: is the set of initial property of the read/write lock.

The most common read/write lock is one where the lock is initially in the unlock.

This kind of mutex lock is created using the (default) attribute null:

```
        pthread_rwlock_init(&x, NULL); /*Default initialization
*/
```

RW Locks

Read lock a read/write lock:

```
        int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
```

NOTE: if a thread already holds a write lock on a read/write lock, and performs a pthread_rwlock_rdlock() on that lock, then the outcome is undefined

NOTE: A thread may hold multiple concurrent read locks on a read/write lock (that is, successfully call the pthread_rwlock_rdlock() function n times). If so, the thread must perform matching unlocks (that is, it must call the pthread_rwlock_unlock() function n times).

RW Locks

Write lock a read/write lock:

```
    int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
```

NOTE: if a thread already holds a read lock or write lock on a read/write lock, and performs a pthread_rwlock_wrlock() on that lock, then the outcome is undefined (in order words: do NOT try !)

```
pthread_rwlock_t rwlock_id;
int var_global  = 4;
void * Read_Lock_Thread()
{
       int ret, i=0;
    while(i <2 )
    {
         ret = pthread_rwlock_rdlock(&rwlock_id);
         if(ret != 0)
                perror("error in read lock taking");
         printf("READ : value of global variable is %d\n",
var_global);
         sleep(10);
         ret = pthread_rwlock_unlock(&rwlock_id);
         if(ret != 0)
                perror("read un-lock");
         i++;
         sleep(1);
    }
}
```

```
void * Write_Lock_Thread()
{
    int ret;
    ret = pthread_rwlock_wrlock(&rwlock_id);
    if(ret != 0)
        perror("error in write lock taking");
    printf(" WRITE : value of global variable is %d\n", var_global);
    var_global = 5;
    printf(" WRITE : value of global variable changed is %d\n",
var_global);
    ret = pthread_rwlock_unlock(&rwlock_id);
    if(ret != 0)
        perror("Write un-lock");
}
```

```
int main()
{
    int ret;
    pthread_t thread_id[2];
    // initializing read write lock
    ret = pthread_rwlock_init(&rwlock_id, NULL);
    if(ret != 0)
        perror("RWlock initialize ");
    // Creates thread which only read the value
    ret = pthread_create(&thread_id[0], NULL, Read_Lock_Thread, 
NULL);
    if(ret != 0)
        perror("Read lock thread ");
    // Creates thread which only Write the value
    ret = pthread_create(&thread_id[1], NULL, Write_Lock_Thread, 
NULL);
    if(ret != 0)
        perror("Write lock thread ");
    sleep(2);
```

```
        ret = pthread_rwlock_rdlock(&rwlock_id);
    if(ret != 0)
         perror("error in read lock taking");
    printf(" MAIN : value of global variable is %d\n", var_global);
    ret = pthread_rwlock_unlock(&rwlock_id);
    if(ret != 0)
         perror("Write un-lock");

    pthread_join(thread_id[0],NULL);
    pthread_join(thread_id[1],NULL);
}
```