**THREAD**

**1)What is a Thread?**

🡪 A **thread** is a sequence of such instructions within a program that can be executed independently of other code. A process can contain multiple threads. Some information of the program cannot be replicated, such as the stack (stack pointer to a different memory area per thread), registers and thread-specific data. This information sufficies to allow threads to be scheduled independently of the program's main thread and possibly one or more other threads within the program.

**2) Process vs Thread?**  
🡪 The primary 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). But, like process, a thread has its own program counter (PC), register set, and stack space.

**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 process context switch. Process 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 processor. 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.

5. **Communication:** Communication between multiple threads is easier, as the threads shares common address space. while in process we have to follow some specific communication technique for communication between two process.

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.

**Difference between Process and Thread:** 

|  |  |  |
| --- | --- | --- |
| S.NO | Process | Thread |
| 1. | Process means any program is in execution. | Thread means segment of a process. |
| 2. | Process takes more time to terminate. | Thread takes less time to terminate. |
| 3. | It takes more time for creation. | It takes less time for creation. |
| 4. | It also takes more time for context switching. | It takes less time for context switching. |
| 5. | Process is less efficient in term of communication. | Thread is more efficient in term of communication. |
| 6. | Process consume more resources. | Thread consume less resources. |
| 7. | Process is isolated. | Threads share memory. |
| 8. | Process is called heavy weight process. | Thread is called light weight process. |
| 9. | Process switching uses interface in operating system. | Thread switching does not require to call a operating system and cause an interrupt to the kernel. |
| 10. | If one process is blocked then it will not effect the execution of other process | Second thread in the same task couldnot run, while one server thread is blocked. |
| 11. | Process has its own Process Control Block, Stack and Address Space. | Thread has Parents’ PCB, its own Thread Control Block and Stack and common Address space. |

**3) Why Multithreading?/Why Thread is required?**  
A thread is also known as lightweight process. The idea is to achieve parallelism by dividing a process into multiple threads. For example, in a browser, multiple tabs can be different threads. MS word uses multiple threads, one thread to format the text, other thread to process inputs, etc.  
Advantages of threads are :-   
1) Thread creation is much faster.  
2) Context switching between threads is much faster.  
3) Threads can be terminated easily  
4) Communication between threads is faster.

**4) Creating Thread /PTHREAD:**

In a **Unix/Linux operating system**, the **C/C++ languages** provide the POSIX thread(pthread) standard API(Application program Interface) for all thread related functions. It allows us to create multiple threads for concurrent process flow. It is most effective on multiprocessor or multi-core systems where threads can be implemented on a kernel-level for achieving the speed of execution. Gains can also be found in uni-processor systems by exploiting the latency in IO or other system functions that may halt a process.We must include the pthread.h header file at the beginning of the script to use all the functions of the pthreads library. To execute the c file, we have to use the -pthread or -lpthread in the command line while compiling the file.

cc -pthread file.c or

cc -lpthread file.c

The **functions** defined in the **pthreads library** include:

1. **pthread\_create:** used to create a new thread

**Syntax:**

int pthread\_create(pthread\_t \* thread, const pthread\_attr\_t \* attr, void \* (\*start\_routine)(void \*),void \*arg);

**Parameters:**

* + **attr:** pointer to a structure that is used to define thread attributes like detached state, scheduling policy, stack address, etc. Set to NULL for default thread attributes.
  + **start\_routine:** pointer to a subroutine that is executed by the thread. The return type and parameter type of the subroutine must be of type void \*. The function has a single attribute but if multiple values need to be passed to the function, a struct must be used.
  + **arg:** pointer to void that contains the arguments to the function defined in the earlier argument

1. **pthread\_exit:** used to terminate a thread

**Syntax:**

void pthread\_exit(void \*retval);

**Parameters:** This method accepts a mandatory parameter **retval** which is the pointer to an integer that stores the return status of the thread terminated. The scope of this variable must be global so that any thread waiting to join this thread may read the return status.

**Note:** If we use exit() instead of **pthread\_exit()** to end a thread, the whole process with all associated threads will be terminated even if some of the threads may still be running.

1. **pthread\_join:** used to wait for the termination of a thread.

**Syntax:**

int pthread\_join(pthread\_t th,void \*\*thread\_return);

**Parameter:** This method accepts following parameters:

* 1. **th:** thread id of the thread for which the current thread waits.
  2. **thread\_return:** pointer to the location where the exit status of the thread mentioned in th is stored.

1. **pthread\_self:** used to get the thread id of the current thread.

**Syntax:**

pthread\_t pthread\_self(void);

1. **pthread\_equal:** compares whether two threads are the same or not. If the two threads are equal, the function returns a non-zero value otherwise zero.

**Syntax:**

int pthread\_equal(pthread\_t t1,pthread\_t t2);

**Parameters:** This method accepts following parameters:

* 1. t1: the thread id of the first thread
  2. t2: the thread id of the second thread

1. **pthread\_cancel:** used to send a cancellation request to a thread

**Syntax:**

int pthread\_cancel(pthread\_t thread);

**Parameter:** This method accepts a mandatory parameter **thread** which is the thread id of the thread to which cancel request is sent.

1. **pthread\_detach:** used to detach a thread. A detached thread does not require a thread to join on terminating. The resources of the thread are automatically released after terminating if the thread is detached.

**Syntax:**

int pthread\_detach(pthread\_t thread);

**Parameter:** This method accepts a mandatory parameter **thread** which is the thread id of the thread that must be detached.

**5)Types of Threads** **(detached thread vs Joinable (non-detached) thread )**  
With every thread some resources are associated like stack and thread local storage etc. When a thread exits ideally these resources should be reclaimed by process automatically. But that doesn’t happens always. It depends on which mode thread is running. A Thread can run in two modes i.e.

* Joinable Mode
* Detached Mode

## **Joinable Thread**

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.pthread\_join() is a blocking call, it will block the calling thread until the other thread ends.

## **Detached thread (Non-Joinable Thread)**

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.  
Also, as 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.

**6)Thread attributes:**

Attributes are a way to specify behavior that is different from the default. When a thread is created with pthread\_create() or when a synchronization variable is initialized, an attribute object can be specified.

**Note:** however the default attributes are usually sufficient for most applications.

**Important Note**: Attributes are specified **only at thread creation time**; they **cannot** be altered while the thread is **being used**.

Thus three functions are usually called in sequence

* **Thread attibute intialisation** -- pthread\_attr\_init() create a default pthread\_attr\_t tattr
* **Thread attribute value change (unless defaults appropriate)** -- a variety of pthread\_attr\_\*() functions are available to set individual attribute values for the pthread\_attr\_t tattr structure. (see below).
* **Thread creation** -- a call to pthread\_create() with approriate attribute values set in a pthread\_attr\_t tattr structure.

The following code fragment should make this point clearer:

#include <pthread.h>

pthread\_attr\_t tattr;

pthread\_t tid;

void \*start\_routine;

void arg

int ret;

/\* initialized with default attributes \*/

ret = pthread\_attr\_init(&tattr);

/\* call an appropriate functions to alter a default value \*/

ret = pthread\_attr\_\*(&tattr,SOME\_ATRIBUTE\_VALUE\_PARAMETER);

/\* create the thread \*/

ret = pthread\_create(&tid, &tattr, start\_routine, arg);

In order to save space, code examples mainly focus on the attribute setting functions and the intializing and creation functions are omitted. These **must** of course be present in all actual code fragments.

An attribute object is opaque, and cannot be directly modified by assignments. A set of functions is provided to initialize, configure, and destroy each object type. Once an attribute is initialized and configured, it has process-wide scope. The suggested method for using attributes is to configure all required state specifications at one time in the early stages of program execution. The appropriate attribute object can then be referred to as needed. Using attribute objects has two primary advantages:

* First, it adds to code portability. Even though supported attributes might vary between implementations, you need not modify function calls that create thread entities because the attribute object is hidden from the interface. If the target port supports attributes that are not found in the current port, provision must be made to manage the new attributes. This is an easy porting task though, because attribute objects need only be initialized once in a well-defined location.
* Second, state specification in an application is simplified. As an example, consider that several sets of threads might exist within a process, each providing a separate service, and each with its own state requirements. At some point in the early stages of the application, a thread attribute object can be initialized for each set. All future thread creations will then refer to the attribute object initialized for that type of thread. The initialization phase is simple and localized, and any future modifications can be made quickly and reliably.

Attribute objects require attention at process exit time. When the object is initialized, memory is allocated for it. This memory must be returned to the system. The pthreads standard provides function calls to destroy attribute objects.

# Initializing Thread Attributes

The function pthread\_attr\_init() is used to initialize object attributes to their default values. The storage is allocated by the thread system during execution.

The function is prototyped by:

int pthread\_attr\_init(pthread\_attr\_t \*tattr);

An example call to this function is:

#include <pthread.h>

pthread\_attr\_t tattr;

int ret;

/\* initialize an attribute to the default value \*/

ret = pthread\_attr\_init(&tattr);

The default values for attributes (tattr) are:

|  |  |  |
| --- | --- | --- |
| Attribute | Value | Result |
| scope | PTHREAD\_SCOPE\_PROCESS | New thread is unbound - not permanently attached to LWP. |
| detachstate | PTHREAD\_CREATE\_JOINABLE | Exit status and thread are preserved after the thread terminates. |
| stackaddr | NULL | New thread has system-allocated stack address. |
| stacksize | 1 megabyte | New thread has system-defined stack size. |
| priority |  | New thread inherits parent thread priority. |
| inheritsched | PTHREAD\_INHERIT\_SCHED | New thread inherits parent thread scheduling priority. |
| schedpolicy | SCHED\_OTHER | New thread uses Solaris-defined fixed priority scheduling; threads run until preempted by a higher-priority thread or until they block or yield. |

#### Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred.

## **Destroy Attributes**

### pthread\_attr\_destroy(3THR)

Use pthread\_attr\_destroy(3THR) to remove the storage allocated during initialization. The attribute object becomes invalid.

Prototype:

int pthread\_attr\_destroy(pthread\_attr\_t \*tattr);

#include <pthread.h>

pthread\_attr\_t tattr;

int ret;

/\* destroy an attribute \*/

ret = pthread\_attr\_destroy(&tattr);

#### Return Values

**pthread\_attr\_destroy()** returns zero after completing successfully. Any other returned value indicates that an error occurred.

## **SCOPE**

## **Set Scope**

### pthread\_attr\_setscope(3THR)

Use [pthread\_attr\_setscope(3THR)](https://docs.oracle.com/docs/cd/E19455-01/806-0630/6j9vkb8ff/index.html) to create a bound thread (PTHREAD\_SCOPE\_SYSTEM) or an unbound thread (PTHREAD\_SCOPE\_PROCESS).

The **pthread\_attr\_setscope**() function sets the contention scope

attribute of the thread attributes object referred to by *attr* to

the value specified in *scope*. The contention scope attribute

defines the set of threads against which a thread competes for

resources such as the CPU. POSIX.1 specifies two possible values

for *scope*:

**PTHREAD\_SCOPE\_SYSTEM**

The thread competes for resources with all other threads

in all processes on the system that are in the same

scheduling allocation domain (a group of one or more

processors). **PTHREAD\_SCOPE\_SYSTEM** threads are scheduled

relative to one another according to their scheduling

policy and priority.

**PTHREAD\_SCOPE\_PROCESS**

The thread competes for resources with all other threads

in the same process that were also created with the

**PTHREAD\_SCOPE\_PROCESS** contention scope.

**PTHREAD\_SCOPE\_PROCESS** threads are scheduled relative to

other threads in the process according to their scheduling

policy and priority. POSIX.1 leaves it unspecified how

these threads contend with other threads in other process

on the system or with other threads in the same process

that were created with the **PTHREAD\_SCOPE\_SYSTEM** contention

scope.

POSIX.1 requires that an implementation support at least one of

these contention scopes. Linux supports **PTHREAD\_SCOPE\_SYSTEM**,

but not **PTHREAD\_SCOPE\_PROCESS**.

On systems that support multiple contention scopes, then, in

order for the parameter setting made by **pthread\_attr\_setscope**()

to have effect when calling [pthread\_create(3)](https://man7.org/linux/man-pages/man3/pthread_create.3.html), the caller must

use [pthread\_attr\_setinheritsched(3)](https://man7.org/linux/man-pages/man3/pthread_attr_setinheritsched.3.html) to set the inherit-scheduler

attribute of the attributes object *attr* to

**PTHREAD\_EXPLICIT\_SCHED**.

The **pthread\_attr\_getscope**() function returns the contention scope

attribute of the thread attributes object referred to by *attr* in

the buffer pointed to by *scope*.

**Note -**

Both thread types are accessible only within a given process.

**Prototype**:

int pthread\_attr\_setscope(pthread\_attr\_t \*tattr,int scope);

#include <pthread.h>

pthread\_attr\_t tattr;

int ret;

/\* bound thread \*/

ret = pthread\_attr\_setscope(&tattr, PTHREAD\_SCOPE\_SYSTEM);

/\* unbound thread \*/

ret = pthread\_attr\_setscope(&tattr, PTHREAD\_SCOPE\_PROCESS);

Notice that there are three function calls in this example: one to initialize the attributes, one to set any variations from the default attributes, and one to create the pthreads.

#include <pthread.h>

pthread\_attr\_t attr;

pthread\_t tid;

void start\_routine;

void arg;

int ret;

/\* initialized with default attributes \*/

ret = pthread\_attr\_init (&tattr);

/\* BOUND behavior \*/

ret = pthread\_attr\_setscope(&tattr, PTHREAD\_SCOPE\_SYSTEM);

ret = pthread\_create (&tid, &tattr, start\_routine, arg);

#### Return Values

**pthread\_attr\_setscope()** returns zero after completing successfully. Any other returned value indicates that an error occurred.

## **Get Scope**

### pthread\_attr\_getscope(3THR)

Use [pthread\_attr\_getscope(3THR)](https://docs.oracle.com/docs/cd/E19455-01/806-0630/6j9vkb8f6/index.html) to retrieve the thread scope, which indicates whether the thread is bound or unbound.

**Prototype**:

int pthread\_attr\_getscope(pthread\_attr\_t \*tattr, int \*scope);

#include <pthread.h>

pthread\_attr\_t tattr;

int scope;

int ret;

/\* get scope of thread \*/

ret = pthread\_attr\_getscope(&tattr, &scope);

#### Return Values

**pthread\_attr\_getscope()** returns zero after completing successfully. Any other returned value indicates that an error occurred

**2)DETACH STATE:**

## **Set Detach State**

### pthread\_attr\_setdetachstate(3THR)

When a thread is created detached (PTHREAD\_CREATE\_DETACHED), its thread ID and other resources can be reused as soon as the thread terminates. Use [pthread\_attr\_setdetachstate(3THR)](https://docs.oracle.com/docs/cd/E19455-01/806-0630/6j9vkb8fa/index.html) when the calling thread does not want to wait for the thread to terminate.

When a thread is created nondetached (PTHREAD\_CREATE\_JOINABLE), it is assumed that you will be waiting for it. That is, it is assumed that you will be executing a **pthread\_join(3T)()** on the thread.

Whether a thread is created detached or nondetached, the process does not exit until all threads have exited. See ["Finishing Up"](https://docs.oracle.com/cd/E19455-01/806-5257/6je9h032i/index.html#tlib-12602) for a discussion of process termination caused by premature exit from **main()**.

**Prototype**:

int pthread\_attr\_setdetachstate(pthread\_attr\_t \*tattr,int detachstate);

#include <pthread.h>

pthread\_attr\_t tattr;

int ret;

/\* set the thread detach state \*/

ret = pthread\_attr\_setdetachstate(&tattr,PTHREAD\_CREATE\_DETACHED);

**Note -**

When there is no explicit synchronization to prevent it, a newly created, detached thread can die and have its thread ID reassigned to another new thread before its creator returns from **pthread\_create()**.

For nondetached (PTHREAD\_CREATE\_JOINABLE) threads, it is very important that some thread join with it after it terminates--otherwise the resources of that thread are not released for use by new threads. This commonly results in a memory leak. So when you do not want a thread to be joined, create it as a detached thread.

##### Example 3-1 Creating a Detached Thread

#include <pthread.h>

pthread\_attr\_t tattr;

pthread\_t tid;

void \*start\_routine;

void arg

int ret;

/\* initialized with default attributes \*/

ret = **pthread\_attr\_init()**(&tattr);

ret = **pthread\_attr\_setdetachstate()**(&tattr,PTHREAD\_CREATE\_DETACHED);

ret = **pthread\_create()**(&tid, &tattr, start\_routine, arg);

#### Return Values

**pthread\_attr\_setdetachstate()** returns zero after completing successfully. Any other returned value indicates that an error occurred.

## **Get Detach State**

### pthread\_attr\_getdetachstate(3THR)

Use [pthread\_attr\_getdetachstate(3THR)](https://docs.oracle.com/docs/cd/E19455-01/806-0630/6j9vkb8f1/index.html) to retrieve the thread create state, which can be either detached or joined.

**Prototype**:

int pthread\_attr\_getdetachstate(const pthread\_attr\_t \*tattr,

int \*detachstate;

#include <pthread.h>

pthread\_attr\_t tattr;

int detachstate;

int ret;

/\* get detachstate of thread \*/

ret = pthread\_attr\_getdetachstate (&tattr, &detachstate);

#### Return Values

**pthread\_attr\_getdetachstate()** returns zero after completing successfully. Any other returned value indicates that an error occurred.

**3)** **STACKADDR:**

## About Stacks

Typically, thread stacks begin on page boundaries and any specified size is rounded up to the next page boundary. A page with no access permission is appended to the top of the stack so that most stack overflows result in sending a SIGSEGV signal to the offending thread. Thread stacks allocated by the caller are used as is.

When a stack is specified, the thread should also be created PTHREAD\_CREATE\_JOINABLE. That stack cannot be freed until the pthread\_join(3T) call for that thread has returned, because the thread's stack cannot be freed until the thread has terminated. The only reliable way to know if such a thread has terminated is through pthread\_join(3T).

Generally, you do not need to allocate stack space for threads. The threads library allocates 1 Mbyte of virtual memory for each thread's stack with no swap space reserved. (The library uses the MAP\_NORESERVE option of **mmap()** to make the allocations.)

Each thread stack created by the threads library has a red zone. The library creates the red zone by appending a page to the top of a stack to catch stack overflows. This page is invalid and causes a memory fault if it is accessed. Red zones are appended to all automatically allocated stacks whether the size is specified by the application or the default size is used.

**Note -**

Because runtime stack requirements vary, you should be absolutely certain that the specified stack will satisfy the runtime requirements needed for library calls and dynamic linking.

There are very few occasions when it is appropriate to specify a stack, its size, or both. It is difficult even for an expert to know if the right size was specified. This is because even a program compliant with ABI standards cannot determine its stack size statically. Its size is dependent on the needs of the particular runtime environment in which it executes.

### **Building Your Own Stack**

When you specify the size of a thread stack, be sure to account for the allocations needed by the invoked function and by each function called. The accounting should include calling sequence needs, local variables, and information structures.

Occasionally you want a stack that is a bit different from the default stack. An obvious situation is when the thread needs more than one megabyte of stack space. A less obvious situation is when the default stack is too large. You might be creating thousands of threads and not have enough virtual memory to handle the gigabytes of stack space that this many default stacks require.

The limits on the maximum size of a stack are often obvious, but what about the limits on its minimum size? There must be enough stack space to handle all of the stack frames that are pushed onto the stack, along with their local variables, and so on.

You can get the absolute minimum limit on stack size by calling the macro PTHREAD\_STACK\_MIN, which returns the amount of stack space required for a thread that executes a NULL procedure. Useful threads need more than this, so be very careful when reducing the stack size.

#include <pthread.h>

pthread\_attr\_t tattr;

pthread\_t tid;

int ret;

int size = PTHREAD\_STACK\_MIN + 0x4000;

/\* initialized with default attributes \*/

ret = pthread\_attr\_init(&tattr);

/\* setting the size of the stack also \*/

ret = pthread\_attr\_setstacksize(&tattr, size);

/\* only size specified in tattr\*/

ret = pthread\_create(&tid, &tattr, start\_routine, arg);

When you allocate your own stack, be sure to append a red zone to its end by calling mprotect(2).

## Set Stack Address

### pthread\_attr\_setstackaddr(3THR)

pthread\_attr\_setstackaddr(3THR) sets the thread stack address.

The **stackaddr** attribute defines the base of the thread's stack. If this is set to non-null (NULL is the default) the system initializes the stack at that address.

**Prototype**:

int pthread\_attr\_setstackaddr(pthread\_attr\_t \*tattr,void \*stackaddr);

#include <pthread.h>

pthread\_attr\_t tattr;

void \*base;

int ret;

base = (void \*) malloc(PTHREAD\_STACK\_MIN + 0x4000);

/\* setting a new address \*/

ret = pthread\_attr\_setstackaddr(&tattr, base);

In the previous example, base contains the address for the stack that the new thread uses. If base is NULL, then pthread\_create(3T) allocates a stack for the new thread with at least PTHREAD\_STACK\_MIN bytes.

#### Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred.

This example shows how to create a thread with a custom stack address.

#include <pthread.h>

pthread\_attr\_t tattr;

pthread\_t tid;

int ret;

void \*stackbase;

stackbase = (void \*) malloc(size);

/\* initialized with default attributes \*/

ret = pthread\_attr\_init(&tattr);

/\* setting the base address in the attribute \*/

ret = pthread\_attr\_setstackaddr(&tattr, stackbase);

/\* only address specified in attribute tattr \*/

ret = pthread\_create(&tid, &tattr, func, arg);

This example shows how to create a thread with both a custom stack address and a custom stack size.

#include <pthread.h>

pthread\_attr\_t tattr;

pthread\_t tid;

int ret;

void \*stackbase;

int size = PTHREAD\_STACK\_MIN + 0x4000;

stackbase = (void \*) malloc(size);

/\* initialized with default attributes \*/

ret = pthread\_attr\_init(&tattr);

/\* setting the size of the stack also \*/

ret = pthread\_attr\_setstacksize(&tattr, size);

/\* setting the base address in the attribute \*/

ret = pthread\_attr\_setstackaddr(&tattr, stackbase);

/\* address and size specified \*/

ret = pthread\_create(&tid, &tattr, func, arg);

## Get Stack Address

### pthread\_attr\_getstackaddr(3THR)

pthread\_attr\_getstackaddr(3THR) returns the thread stack address set by **pthread\_attr\_setstackaddr()**.

**Prototype**:

int pthread\_attr\_getstackaddr(pthread\_attr\_t \*tattr,void \* \*stackaddr);

#include <pthread.h>

pthread\_attr\_t tattr;

void \*base;

int ret;

/\* getting a new address \*/

ret = pthread\_attr\_getstackaddr (&tattr, \*base);

#### Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred.

**4)STACK SIZE**

## **Set Stack Size**

### pthread\_attr\_setstacksize(3THR)

pthread\_attr\_setstacksize(3THR) sets the thread stack size.

The **stacksize** attribute defines the size of the stack (in bytes) that the system will allocate. The size should not be less than the system-defined minimum stack size. See "About Stacks" for more information.

**Prototype**:

int pthread\_attr\_setstacksize(pthread\_attr\_t \*tattr, int size);

#include <pthread.h>

pthread\_attr\_t tattr;

int size;

int ret;

size = (PTHREAD\_STACK\_MIN + 0x4000);

/\* setting a new size \*/

ret = pthread\_attr\_setstacksize(&tattr, size);

In the example above, size contains the number of bytes for the stack that the new thread uses. If **size** is zero, a default size is used. In most cases, a zero value works best.

PTHREAD\_STACK\_MIN is the amount of stack space required to start a thread. This does not take into consideration the threads routine requirements that are needed to execute application code.

#### Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred.

## **Get Stack Size:**

### pthread\_attr\_getstacksize(3THR)

pthread\_attr\_getstacksize(3THR) returns the stack size set by **pthread\_attr\_setstacksize()**.

**Prototype**:

int pthread\_attr\_getstacksize(pthread\_attr\_t \*tattr, size\_t \*size);

#include <pthread.h>

pthread\_attr\_t tattr;

int size;

int ret;

/\* getting the stack size \*/

ret = pthread\_attr\_getstacksize(&tattr, &size);

#### Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred.

**5)PRIORITY**

## **Set Scheduling Parameters**

### pthread\_attr\_setschedparam(3THR)

[pthread\_attr\_setschedparam(3THR)](https://docs.oracle.com/docs/cd/E19455-01/806-0630/6j9vkb8fd/index.html) sets the scheduling parameters.

Scheduling parameters are defined in the param structure; only priority is supported. Newly created threads run with this priority.

* SCHED\_FIFO

First-In-First-Out; threads scheduled to this policy, if not preempted by a higher priority, will proceed until completion. Threads that have a contention scope of system (PTHREAD\_SCOPE\_SYSTEM) are in real-time (RT) scheduling class and the calling process must have an effective user id of 0. SCHED\_FIFO for threads that have a contention scope of process (PTHREAD\_SCOPE\_PROCESS) is based on the TS scheduling class.

* SCHED\_RR

Round-Robin; threads scheduled to this policy, if not pre-empted by a higher priority, will execute for a time period determined by the system. Threads that have a contention scope of system (PTHREAD\_SCOPE\_SYSTEM) are in real-time (RT) scheduling class and the calling process must have an effective user id of 0. SCHED\_RR for threads that have a contention scope of process (PTHREAD\_SCOPE\_PROCESS) is based on the TS scheduling class.

Prototype:

int pthread\_attr\_setschedparam(pthread\_attr\_t \*tattr,

const struct sched\_param \*param);

#include <pthread.h>

pthread\_attr\_t tattr;

int newprio;

sched\_param param;

newprio = 30;

/\* set the priority; others are unchanged \*/

param.sched\_priority = newprio;

/\* set the new scheduling param \*/

ret = pthread\_attr\_setschedparam (&tattr, &param);

#### Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred.

You can manage pthreads priority two ways. You can set the priority attribute before creating a child thread, or you can change the priority of the parent thread and then change it back.

## **Get Scheduling Parameters**

### pthread\_attr\_getschedparam(3THR)

[pthread\_attr\_getschedparam(3THR)](https://docs.oracle.com/docs/cd/E19455-01/806-0630/6j9vkb8f4/index.html) returns the scheduling parameters defined by pthread\_attr\_setschedparam().

Prototype:

int pthread\_attr\_getschedparam(pthread\_attr\_t \*tattr,

const struct sched\_param \*param);

#include <pthread.h>

pthread\_attr\_t attr;

struct sched\_param param;

int ret;

/\* get the existing scheduling param \*/

ret = pthread\_attr\_getschedparam (&tattr, &param);

#### Return Values

pthread\_attr\_setschedparam() returns zero after completing successfully. Any other returned value indicates that an error occurred.

#### **Creating a Thread With a Specified Priority**

You can set the priority attribute before creating the thread. The child thread is created with the new priority that is specified in the sched\_param structure (this structure also contains other scheduling information).

It is always a good idea to get the existing parameters, change the priority, xxx the thread, and then reset the priority.

[Example 3-2](https://docs.oracle.com/cd/E19455-01/806-5257/6je9h032l/index.html#attrib-37680) shows an example of this.

##### Example 3-2 Creating a Prioritized Thread

#include <pthread.h>

#include <sched.h>

pthread\_attr\_t tattr;

pthread\_t tid;

int ret;

int newprio = 20;

sched\_param param;

/\* initialized with default attributes \*/

ret = pthread\_attr\_init (&tattr);

/\* safe to get existing scheduling param \*/

ret = pthread\_attr\_getschedparam (&tattr, &param);

/\* set the priority; others are unchanged \*/

param.sched\_priority = newprio;

/\* setting the new scheduling param \*/

ret = pthread\_attr\_setschedparam (&tattr, &param);

/\* with new priority specified \*/

ret = pthread\_create (&tid, &tattr, func, arg);

**6) INHERITEDSCHED**

## **Set Inherited Scheduling Policy**

### pthread\_attr\_setinheritsched(3THR)

Use pthread\_attr\_setinheritsched(3THR) to set the inherited scheduling policy.

An inherit value of PTHREAD\_INHERIT\_SCHED (the default) means that the scheduling policies defined in the creating thread are to be used, and any scheduling attributes defined in the **pthread\_create()** call are to be ignored. If PTHREAD\_EXPLICIT\_SCHED is used, the attributes from the **pthread\_create()** call are to be used.

**Prototype**:

int pthread\_attr\_setinheritsched(pthread\_attr\_t \*tattr, int inherit);

#include <pthread.h>

pthread\_attr\_t tattr;

int inherit;

int ret;

/\* use the current scheduling policy \*/

ret = pthread\_attr\_setinheritsched(&tattr, PTHREAD\_EXPLICIT\_SCHED);

#### Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred.

## **Get Inherited Scheduling Policy**

### pthread\_attr\_getinheritsched(3THR)

pthread\_attr\_getinheritsched(3THR) returns the scheduling policy set by **pthread\_attr\_setinheritsched()**.

**Prototype**:

int pthread\_attr\_getinheritsched(pthread\_attr\_t \*tattr, int \*inherit);

#include <pthread.h>

pthread\_attr\_t tattr;

int inherit;

int ret;

/\* get scheduling policy and priority of the creating thread \*/

ret = pthread\_attr\_getinheritsched (&tattr, &inherit);

#### Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred.

## **7)SCHEDPOLICY**

## **Set Scheduling Policy**

### pthread\_attr\_setschedpolicy(3THR)

Use [pthread\_attr\_setschedpolicy(3THR)](https://docs.oracle.com/docs/cd/E19455-01/806-0630/6j9vkb8fe/index.html) to set the scheduling policy. The POSIX draft standard specifies scheduling policy attributes of SCHED\_FIFO (first-in-first-out), SCHED\_RR (round-robin), or SCHED\_OTHER (an implementation-defined method).

* SCHED\_FIFO

First-In-First-Out; threads scheduled to this policy, if not preempted by a higher priority, will proceed until completion. Threads whose contention scope is system (PTHREAD\_SCOPE\_SYSTEM) are in real-time (RT) scheduling class and the calling process must have an effective user id of 0. SCHED\_FIFO for threads that have a contention scope of process (PTHREAD\_SCOPE\_PROCESS) is based on the TS scheduling class.

* SCHED\_RR

Round-Robin; threads scheduled to this policy, if not preempted by a higher priority, will execute for a time period determined by the system. Threads whosethat have a contention scope of system (PTHREAD\_SCOPE\_SYSTEM) are in real-time (RT) scheduling class and the calling process must have an effective user id of 0. SCHED\_RR for threads that have a contention scope of process (PTHREAD\_SCOPE\_PROCESS) is based on the TS scheduling class.

SCHED\_FIFO and SCHED\_RR are optional in POSIX, and are supported for real time bound threads only.

Currently, only the Solaris SCHED\_OTHER, time-sharing, default value is supported in pthreads. For a discussion of scheduling, see the section ["Scheduling"](https://docs.oracle.com/cd/E19455-01/806-5257/6je9h032e/index.html#mtintro-69291).

Prototype:

int pthread\_attr\_setschedpolicy(pthread\_attr\_t \*tattr, int policy);

#include <pthread.h>

pthread\_attr\_t tattr;

int policy;

int ret;

/\* set the scheduling policy to SCHED\_OTHER \*/

ret = pthread\_attr\_setschedpolicy(&tattr, SCHED\_OTHER);

#### Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred.

## **Get Scheduling Policy**

### pthread\_attr\_getschedpolicy(3THR)

Use [pthread\_attr\_getschedpolicy(3THR)](https://docs.oracle.com/docs/cd/E19455-01/806-0630/6j9vkb8f5/index.html) to retrieve the scheduling policy. Currently, only the Solaris-based SCHED\_OTHER default value is supported in pthreads.

Prototype:

int pthread\_attr\_getschedpolicy(pthread\_attr\_t \*tattr, int \*policy);

#include <pthread.h>

pthread\_attr\_t tattr;

int policy;

int ret;

/\* get scheduling policy of thread \*/

ret = pthread\_attr\_getschedpolicy (&tattr, &policy);

#### Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred.

**7)** **Access single variable between two threads and their issue (race condition):**

🡪

A race condition occurs when two or more threads can access shared data and they try to change it at the same time. Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are "racing" to access/change the data.

Problems often occur when one thread does a "check-then-act" (e.g. "check" if the value is X, then "act" to do something that depends on the value being X) and another thread does something to the value in between the "check" and the "act". E.g:

if (x == 5) // The "Check"

{

y = x \* 2; // The "Act"

// If another thread changed x in between "if (x == 5)" and "y = x \* 2" above,

// y will not be equal to 10.

}

The point being, y could be 10, or it could be anything, depending on whether another thread changed x in between the check and act. You have no real way of knowing.

In order to prevent race conditions from occurring, you would typically put a lock around the shared data to ensure only one thread can access the data at a time. This would mean something like this:

// Obtain lock for x

if (x == 5)

{

y = x \* 2; // Now, nothing can change x until the lock is released.

// Therefore y = 10

}

// release lock for x

A deadlock occurs when two threads each lock a different variable at the same time and then try to lock the variable that the other thread already locked. As a result, each thread stops executing and waits for the other thread to release the variable. Because each thread is holding the variable that the other thread wants, nothing occurs, and the threads remain deadlocked.

**8)** **Protect shared variable from race condition (mutex/conditional variable)**

**🡪**

Avoiding Race Conditions:

Critical Section:

To avoid race condition we need Mutual Exclusion. Mutual Exclusion is someway of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same things.

The difficulty above in the printer spooler occurs because process B started using one of the shared

variables before process A was finished with it.

That part of the program where the shared memory is accessed is called the critical region or critical

section. If we could arrange matters such that no two processes were ever in their critical regions at the

same time, we could avoid race conditions. Although this requirement avoids race conditions, this is not

sufficient for having parallel processes cooperate correctly and efficiently using shared data.

(Rules for avoiding Race Condition) Solution to Critical section problem:

1. No two processes may be simultaneously inside their critical regions. (Mutual Exclusion)

2. No assumptions may be made about speeds or the number of CPUs.

3. No process running outside its critical region may block other processes.

4. No process should have to wait forever to enter its critical region.

The common mechanism used to allow safe access to shared resource is called **mutex**, which is short for mutual exclusion. Any process entering a critical section will first acquire the mutex. Other processes wanting access to the critical section will get blocked. The owning process will give up the mutex when leaving the critical section.

What separate mutexes from semaphores is the concept of ownership. A mutex can be released only by its owner. This is not so with semaphores that can be used for signalling state change and synchronization from one thread to another. While it's possible to use semaphores to protect critical sections, they come with their own set of problems such as accidental release, recursive deadlock, task-death deadlock, and priority inversion. Mutexes are therefore a better solution.

Usually, however, semaphores are used in a mutex mode. To make this common case a little easier, the kernel has provided a set of helper functions and macros. Thus, a mutex can be declared and initialized with one of the following:

DECLARE\_MUTEX(name);

DECLARE\_MUTEX\_LOCKED(name);

Here, the result is a semaphore variable (called name) that is initialized to 1 (with DECLARE\_MUTEX) or 0 (with DECLARE\_MUTEX\_LOCKED). In the latter case, the mutex starts out in a locked state; it will have to be explicitly unlocked before any thread will be allowed access.

If the mutex must be initialized at runtime (which is the case if it is allocated dynamically, for example), use one of the following:

void init\_MUTEX(struct semaphore \*sem);

void init\_MUTEX\_LOCKED(struct semaphore \*sem);

mutexes - Mutual exclusion lock: Block access to variables by other threads. This enforces exclusive access by a thread to a variable or set of variables.

Mutexes:

Mutexes are used to prevent data inconsistencies due to race conditions. A race condition often occurs when two or more threads need to perform operations on the same memory area, but the results of computations depends on the order in which these operations are performed. Mutexes are used for serializing shared resources. Anytime a global resource is accessed by more than one thread the resource should have a Mutex associated with it. One can apply a mutex to protect a segment of memory ("critical region") from other threads. Mutexes can be applied only to threads in a single process and do not work between processes as do semaphores.

Condition Variables:

A condition variable is a variable of type pthread\_cond\_t and 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.

Functions used in conjunction with the condition variable:

Creating/Destroying:

pthread\_cond\_init

pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIZER;

pthread\_cond\_destroy

Waiting on condition:

pthread\_cond\_wait

pthread\_cond\_timedwait - place limit on how long it will block.

Waking thread based on condition:

pthread\_cond\_signal

pthread\_cond\_broadcast - wake up all threads blocked by the specified condition variable.

In concurrent programming, a monitor is a synchronization construct that allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become false. A lock or mutex (from mutual exclusion) is a synchronization mechanism for enforcing limits on access to a resource in an environment where there are many threads of execution. A lock is designed to enforce a mutual exclusion concurrency control policy.

Condition variables are synchronization primitives that enable threads to wait until a particular condition occurs. Condition variables are user-mode objects that cannot be shared across processes.

Condition variables enable threads to atomically release a lock and enter the sleeping state. They can be used with critical sections or slim reader/writer (SRW) locks. Condition variables support operations that "wake one" or "wake all" waiting threads. After a thread is woken, it re-acquires the lock it released when the thread entered the sleeping state.

A **conditional variable** in operating system programming is a special kind of variable that is used to determine if a certain condition has been met or not. It is used to communicate between threads when certain conditions become true.

A conditional variable is like a queue. A thread stops its execution and enters the queue if the specified condition is not met. Once another thread makes that condition true, it sends a signal to the leading thread in the queue to continue its execution.

There are two types of actions that can be performed with condition variables:

* wait
* signal

We use the *wait* instruction in a thread if we want to halt the execution of that thread till a certain condition is met.

We use the *signal* instruction if we want to continue executing the leading thread in the waiting queue.