# Introduction into Multicore Programming

Károly Bósa
(Karoly.Bosa@jku.at)

Research Institute for Symbolic Computation
(RISC)

# Topic

**Multithreading**

- What is a thread?
- User- and kernel- level thread models
- Thread Context
- Hardware threads
- Process vs. threads
- Thread attributes
- The architecture of a thread
- Creating Thread
- Joining and detaching thread
- Thread Id
- Cancellation and cancelability state
- Thread scheduling and priorities
- Contention scope

# What Is a Thread?

- A thread is a sequence or stream of executable code within a process that is scheduled for execution by the operating system on a processor or core.

- Threads execute independent concurrent tasks of a program.

- All processes have a primary thread.

- A process with multiple threads is *multithreaded.* Its each thread executes independently and concurrently with its own sequence of instructions.

- Threads use minimal resources shared in the address space of a single process as compared to an application, which uses multiple processes.

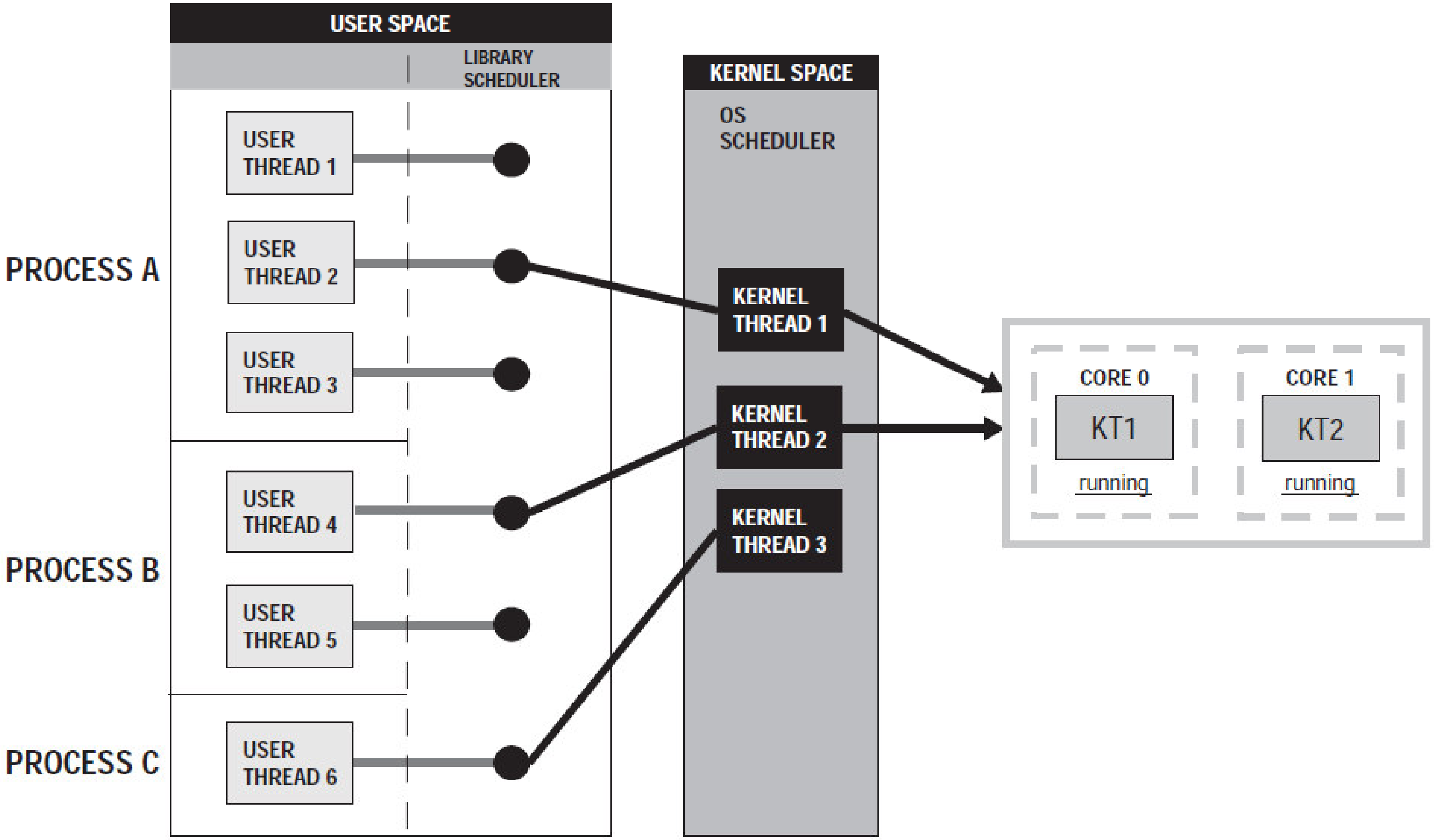
# User - and Kernel - Level Threads

**Karoly.Bosa@jku.at**

- There are three *implementation models* for threads:
    - User- or application - level threads
    - Kernel-level threads (different from kernel threads(!) )
    - Hybrid of user- and kernel-level threads
- The differences between them are the mode they exist in and the ability of the threads to be assigned to a processor.

# User-Level Thread Model

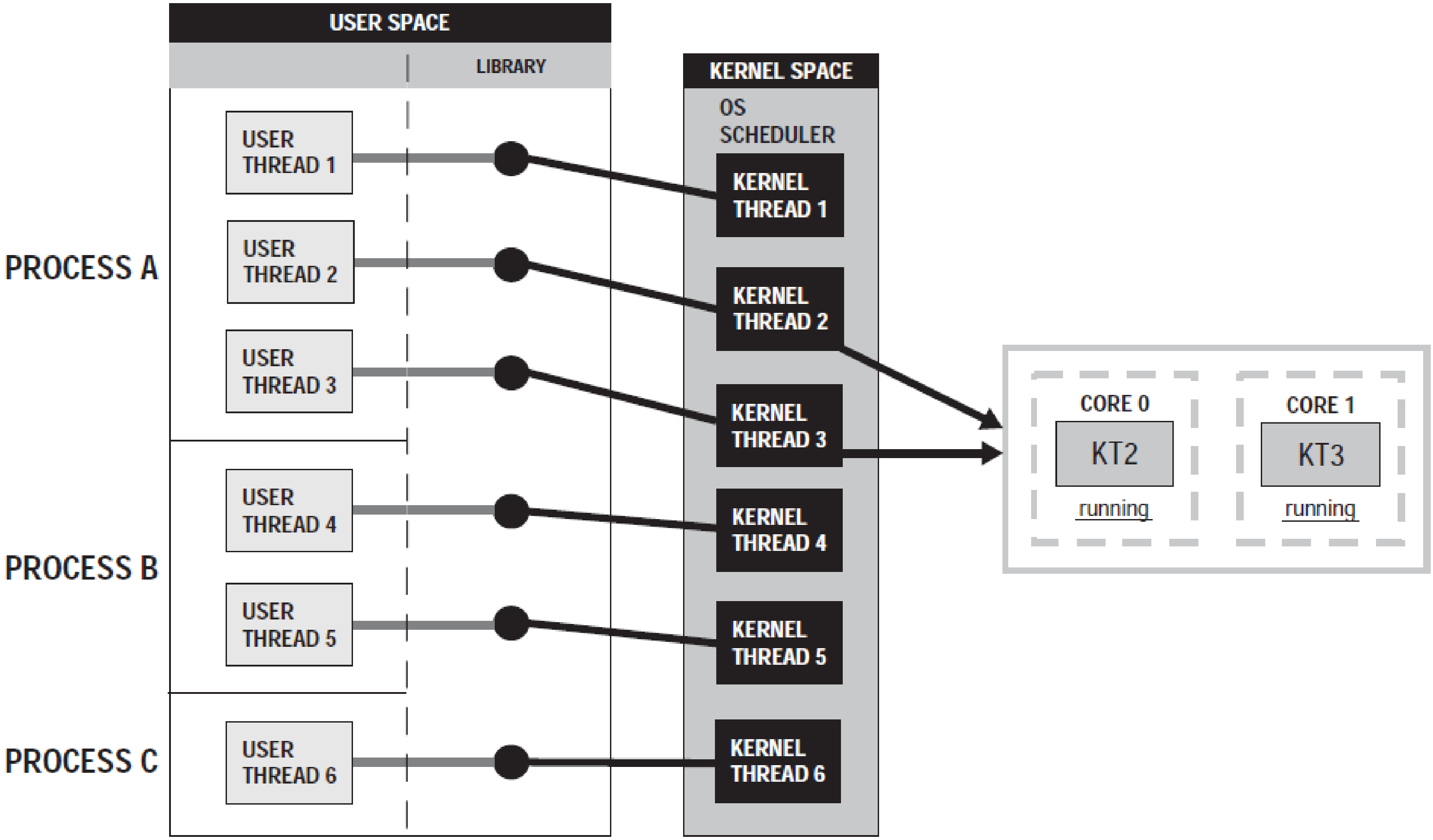
Karoly.Bosa@jku.at



- User-level threads are considered a “many-to-one” thread mapping.

# Kernel-level Thread Model

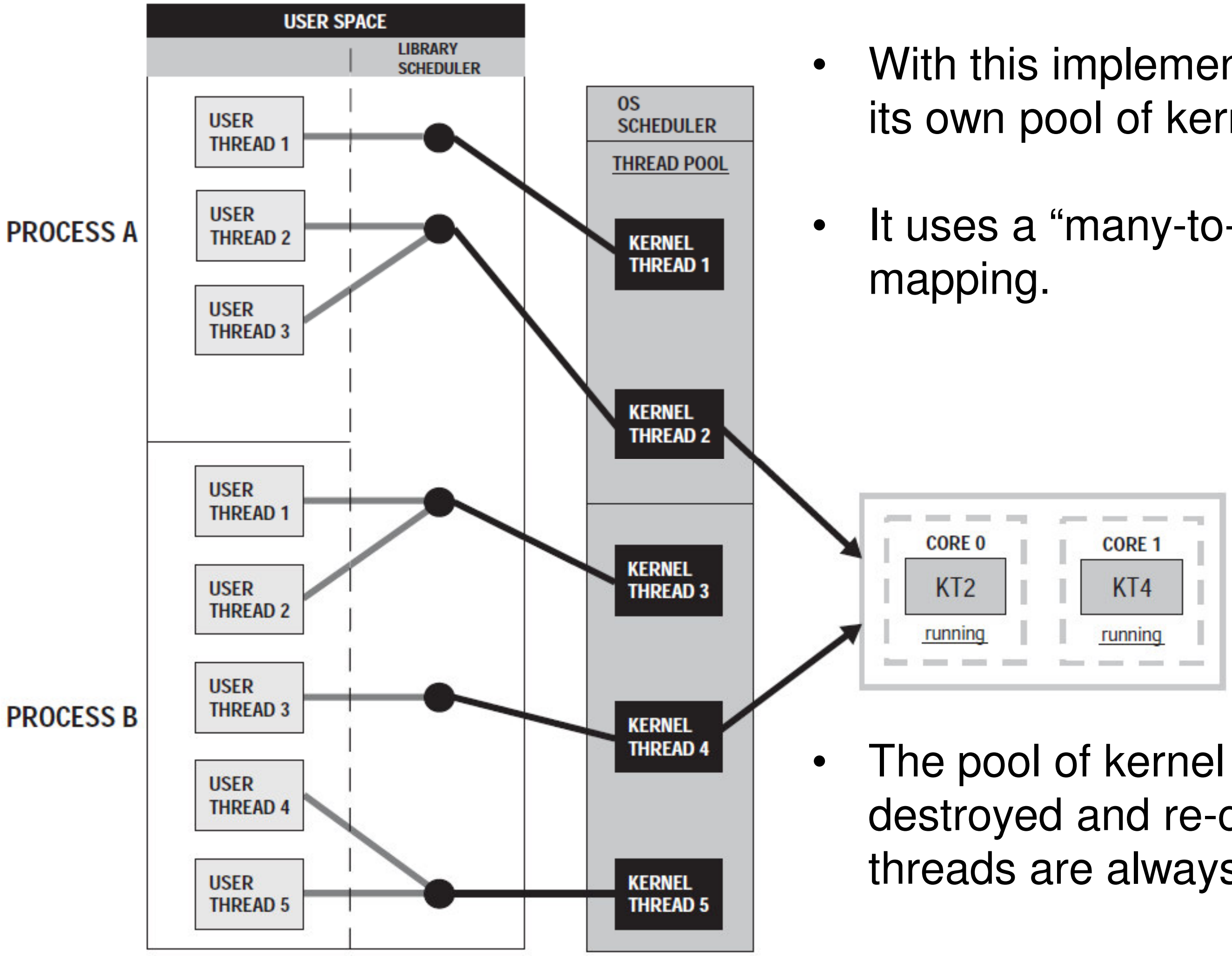
Karoly.Bosa@jku.at



- Kernel-level threads are considered a “one-to-one” thread mapping.

# Hybrid Thread Model

Karoly.Bosa@jku.at



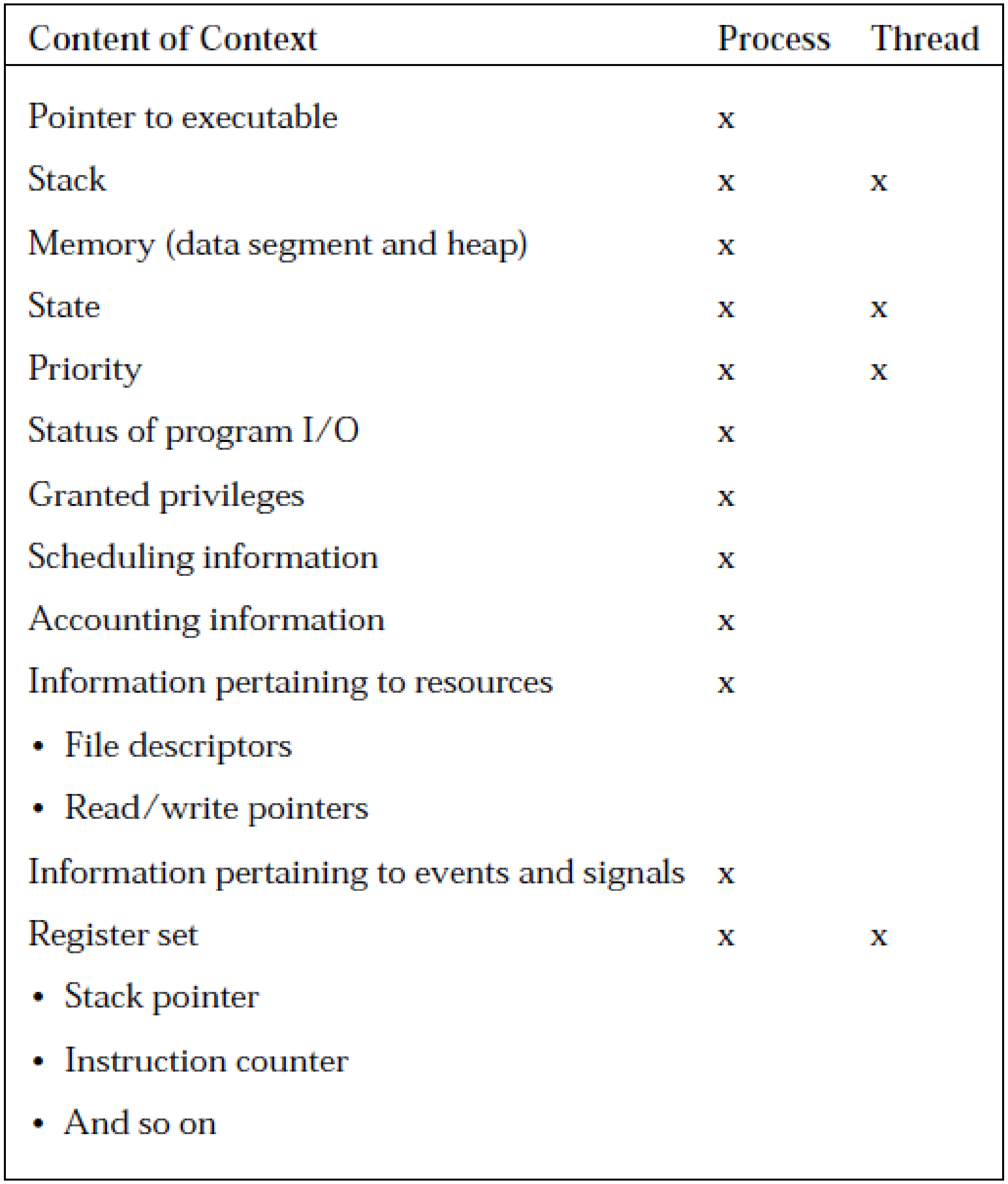
- With this implementation, a process has its own pool of kernel threads.

- It uses a "many-to-many" thread mapping.

- The pool of kernel threads is not destroyed and re-created. These threads are always in the system.

# Thread Context

- Threads also have a context.
- A context switch between threads belonging to the same process is also possible:
  - A process shares much with its threads,
  - but some information is local or unique to the thread.
- The information unique or local to a thread:
  - thread id,
  - processor registers,
  - the state and priority and
  - the thread-specific data (TSD).



| Content of Context | Process | Thread |
|---|---|---|
| Pointer to executable | x | |
| Stack | x | x |
| Memory (data segment and heap) | x | |
| State | x | x |
| Priority | x | x |
| Status of program I/O | x | |
| Granted privileges | x | |
| Scheduling information | x | |
| Accounting information | x | |
| Information pertaining to resources | x | |
| • File descriptors | | |
| • Read/write pointers | | |
| Information pertaining to events and signals | x | |
| Register set | x | x |
| • Stack pointer | | |
| • Instruction counter | | |
| • And so on | | |

# Hardware Threads and Software Threads

- Threads can be implemented in hardware as well as software.
- Chip manufacturers implement cores that have multiple hardware threads that serve as *logical cores* .
- Cores with multiple hardware threads are called *simultaneous multithreaded (SMT)* cores.
- The logical cores are treated as unique processor cores by the operating system.
- Sun's UltraSparc T1 and IBM's Cell Broadband Engine CBE utilize SMT implementing from two to four threads per core.
- Hyperthreading is Intel's implementation of SMT in which its primary purpose is to improve support for multithreaded code.
- Hyperthreading or SMT technology provides (real) parallel execution of threads on a single processor core.

# Thread Resources

- Threads share most of their resources with other threads of the same process.
- A thread can allocate additional resources such as files or mutexes, but they are accessible to all the threads of the process.
- There are limits on the resources that can be consumed by a single process.
- When threads are utilizing their resources, they must be careful not to leave them in an unstable state when they are canceled.
- Before it terminates, a thread should perform some cleanup, preventing these unwanted situations from occurring.

# Process vs. Thread: Context Switching

- A process with multiple threads can provide concurrent execution of the subtasks with less overhead for context switching.

- With low processor availability or a single core:

    - Concurrently executing processes involve heavy overhead because of the context switching.

    - By using threads, a process context switch would occur only when a thread from a different process is assigned the processor.

- Of course, if there are enough processors to go around, then context switching is not an issue.

# Process vs. Thread: Throughput

- The throughput of an application can increase with multiple threads:
- With one thread, an I/O request would halt the entire process.
- With multiple threads, as one thread waits for an I/O request, the application continues to execute.
- As one thread is blocked, another can execute. The entire application does not wait for each I/O request to be filled

# Process vs. Thread: Communication

- Threads:
  - They do not require special mechanisms for communication with other threads of the process (*peer threads*).
  - They communicate by using the memory shared within the address space of the process.
  - This saves system resources that would have to be used in the setup and maintenance of special communication mechanisms.
- Processes:
  - They can also communicate by shared memory, but processes have separate address spaces.
  - The required shared memory must exist outside the address space of both processes (e.g.: *message queue*).
  - Setup of a message queue generally requires a lot of setup to work properly.

# Process vs. Thread: Corrupting Process Data

- Threads:
  - They can easily corrupt the data of a process.
  - Without synchronization, threads write access to the same piece of data can cause data race.
- Processes:
  - Each process has its own data, and other processes don’t have access unless special communication is set up.
  - The separate address spaces of processes protect the data from possible inadvertent corruption by other processes.

# Process vs. Thread: Errors

- Errors caused by a thread are more costly than errors caused by processes.
- Foe instance, If a thread causes a fatal *access violation*, this may result in the termination of the entire process.
- Threads can create data errors that affect the entire memory space of all the peer threads.
- Processes are isolated. A process can have an access violation that causes the process to terminate, but all of the other processes continue executing.
- Data errors can be restricted to a single process.

# Process vs. Thread: Similarities

- Threads and child processes share (some) resources of their parent process without requiring additional initialization or preparation.

- As kernel entities, threads and processes compete for processor usage.

- The parent process has some control over the child process or thread. It can:
    - Cancel
    - Suspend
    - Resume
    - Change the priority

# Process vs. Thread: Relationships

- Processes:
  - They can exercise control over other processes with which they have a parent-child relationship.
  - Changes to the parent process do not affect child processes.
- Threads:
  - Peer threads are on an equal level regardless of who created them.
  - Any thread that has **access to the thread id of another peer thread** can cancel, suspend, resume, or change the priority of that thread.
  - Any thread within a process can kill the process
    - by canceling the primary thread,
    - by terminating all the threads of the process.
  - Any changes to the main thread may affect all the threads of the process.

# Thread Attributes I.

- Information about the thread used to determine the context of the thread.

- What makes peer threads unique from one another is the **id**, the **state** (set of registers) the **priority**, and the **stack**.

- The POSIX thread library defines a thread *attribute object* that encapsulates a subset of the properties:

  - Contention scope
  - Stack size
  - Stack address
  - Detached state
  - Priority
  - Scheduling policy and parameters

- These attributes are accessible and modifiable by the creator of the thread.

- A thread attribute object can be associated with one or multiple threads.

- Once a thread has been created using a thread attribute object, most attributes cannot be changed while the thread is in use.

# Thread Attributes II. - Contention Scope

Karoly.Bosa@jku.at

- **Contention Scope** attribute describes which threads competes each other for resources. There are two kinds of contention scopes:
    - **Process scope:** compete with threads within the same process.
    - **System scope:** compete for resources with threads of other processes allocated across the system.
- A thread that has system scope is prioritized and scheduled with respect to all of the process wide threads.
- Contention scope can potentially impact on the performance of your application:
    - The process scheduling model potentially provides lower overhead for making scheduling decisions.
    - On the other hand system wide threads gets CPU time slice more often.

# Thread Attributes III. - Stack Size and Location

- The thread’s stack size and location are set when the thread is created.

- If not explicitly given, a default stack size and location are assigned by the system.

- The thread’s stack size must be large enough:
    - for any function calls;

    - for any code external to the process, such as library code, called by the thread;

    - for local variable storage.

- A process with multiple threads should have a stack segment large enough for all of its thread’s stacks.

- **If you specify location and size:** the important things is how much space the thread requires and to ensure that the location does not overlap other peer thread’s stacks.

# Thread Attributes IV. – Detached State

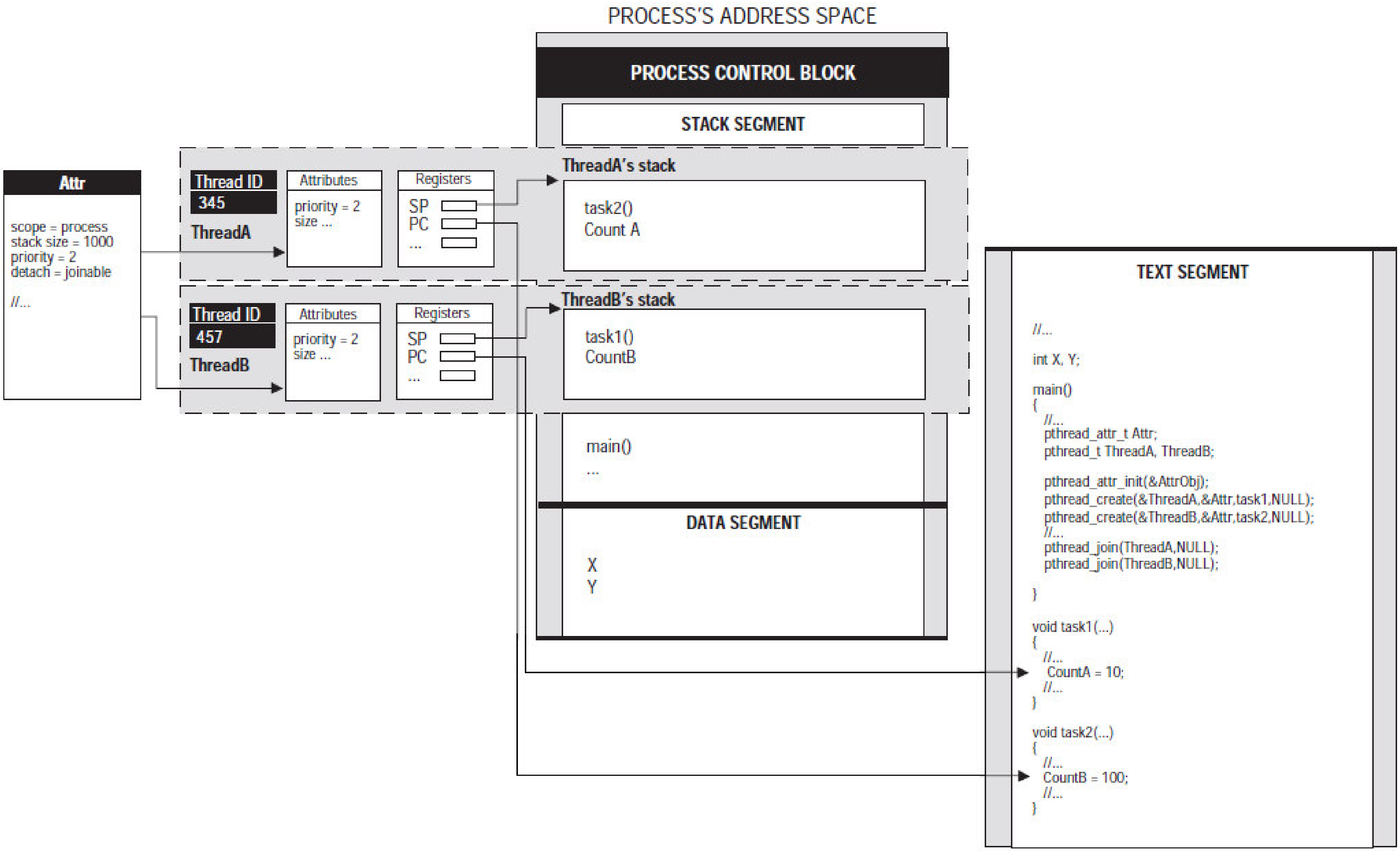
- Detached threads are threads that have become detached from their creator.
- They are not synchronized with other peer threads or the primary thread when it terminates or exits.
- The process or thread that created them gives up any control over them.
- If the thread is detached, once the thread is terminated, no resources are used to save the status or thread id.
- Use detached threads:
    - If it is not necessary for the creator of the thread to wait until it terminates or
    - if a thread does not require any type of synchronization with other peer threads once terminated.

# Thread Attributes V. – Priority and Scheduling I.

- Threads always have a priority,
  - The thread with the highest priority is executed before threads with lower priority.
  - Executing threads are preempted if a thread of higher priority (and the same contention scope) is available.
- The threads inherit scheduling attributes from the process.
- FIFO, round robin (RR), and other scheduling policies are available.
- In general, it is not necessary to change the scheduling attributes of the thread during process execution.
- Changing the scheduling attributes can have a negative impact on the overall performance of the application.

# The Architecture of a Thread

Karoly.Bosa@jku.at



scope = process
stack size = 1000
priority = 2
detach = joinable
//...

priority = 2
size ...

priority = 2
size ...

```
//...

int X, Y;

main()
{
  //...
  pthread_attr_t Attr;
  pthread_t ThreadA, ThreadB;

  pthread_attr_init(&AttrObj);
  pthread_create(&ThreadA,&Attr,task1,NULL);
  pthread_create(&ThreadB,&Attr,task2,NULL);
  //...
  pthread_join(ThreadA,NULL);
  pthread_join(ThreadB,NULL);

}

void task1(...)
{
  //...
   CountA = 10;
  //...
}

void task2(...)
{
  //...
  CountB = 100;
  //...
}
```

# Thread States

- A thread state is the mode or condition in which a thread is at any given time.

- Threads have the same states and transitions as processes.

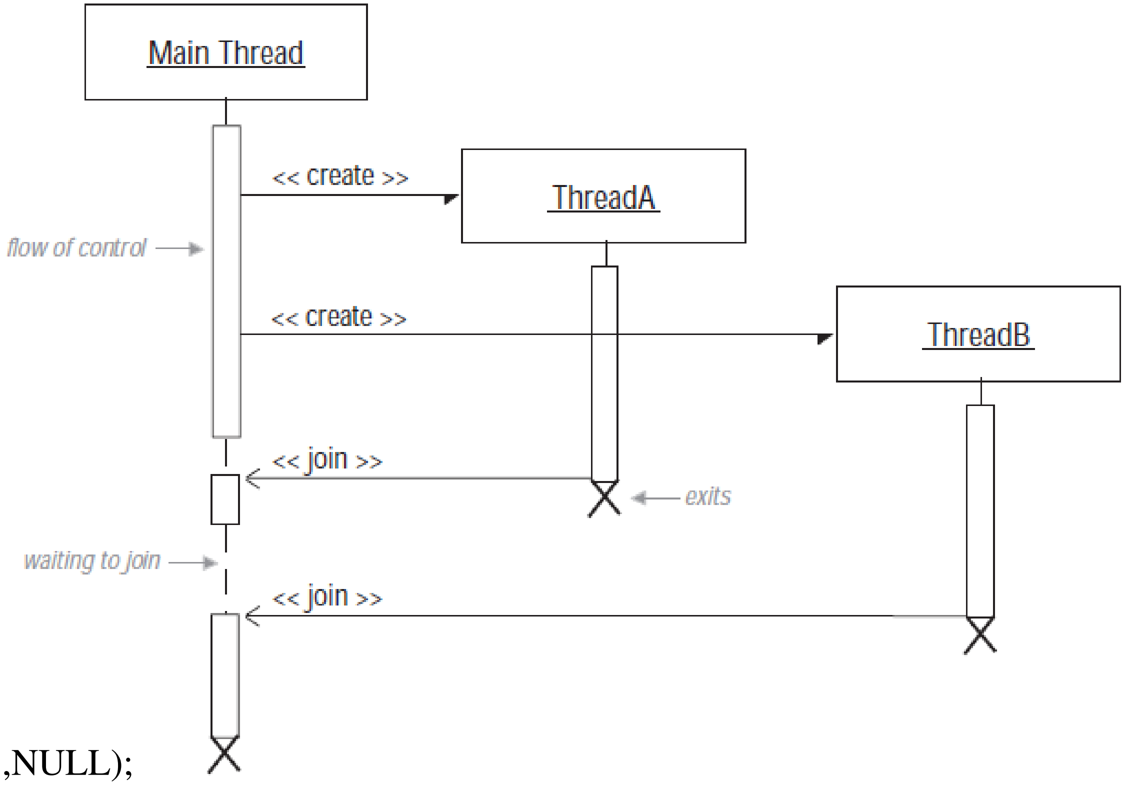
- Commonly implemented states e.g.:
    - Runnable
    - Running (active)
    - Stopped
    - Sleeping (blocked)

- Typical transitions e.g.:
    - Preempt
    - Signaled
    - Dispatch

- If one thread is active (runnable or running), then the process is considered active.

# A Simple Threaded Program

```
using namespace std;
#include < iostream >
#include < pthread.h >

//define task to be executed by ThreadA
void *task1(void *X) {
        cout < < “Thread A complete” < < endl;
        return (NULL);
}

//define task to be executed by ThreadB
void *task2(void *X) {
        cout < < “Thread B complete” < < endl;
        return (NULL);
}

int main(int argc, char *argv[]) {
        // declare threads
        pthread_t ThreadA,ThreadB;
        // create threads
        pthread_create( & ThreadA,NULL,task1,NULL);
        pthread_create( & ThreadB,NULL,task2,NULL);
        // additional processing …
        pthread_join(ThreadA,NULL); // wait for threads
        pthread_join(ThreadB,NULL);
        return (0);
}
```



# Compiling and Linking Threaded Programs

- All multithreaded programs using the POSIX thread library must include this header:
  < pthread.h >

- For compiling, we must link the pthread library to our application using the *–l* compiler switch:

  ```
  -lpthread
  ```

- The pthread library, libpthread.so, should be located in the directory where the system stores its standard library, usually /usr/lib.

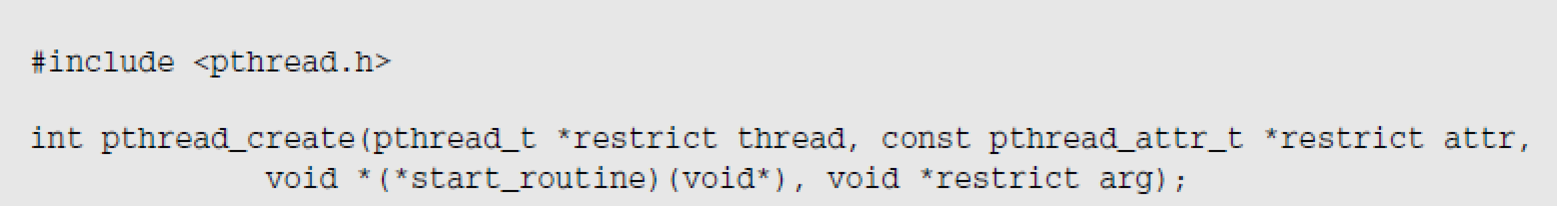
- So the compilation line would look like the following:

  ```
  g++ -o a.out test_thread.cpp -lpthread
  ```

- If the library is not located in a standard location, use the - L option to make the compiler look in a particular directory before searching the standard locations:

  ```
  g++ -o a.out -L /src/local/lib test_thread.cpp -lpthread
  ```

# Creating Threads



```
#include <pthread.h>

int pthread_create(pthread_t *restrict thread, const pthread_attr_t *restrict attr,
              void *(*start_routine)(void*), void *restrict arg);
```

- Threads can be created any time during the execution of a process because they are dynamic.

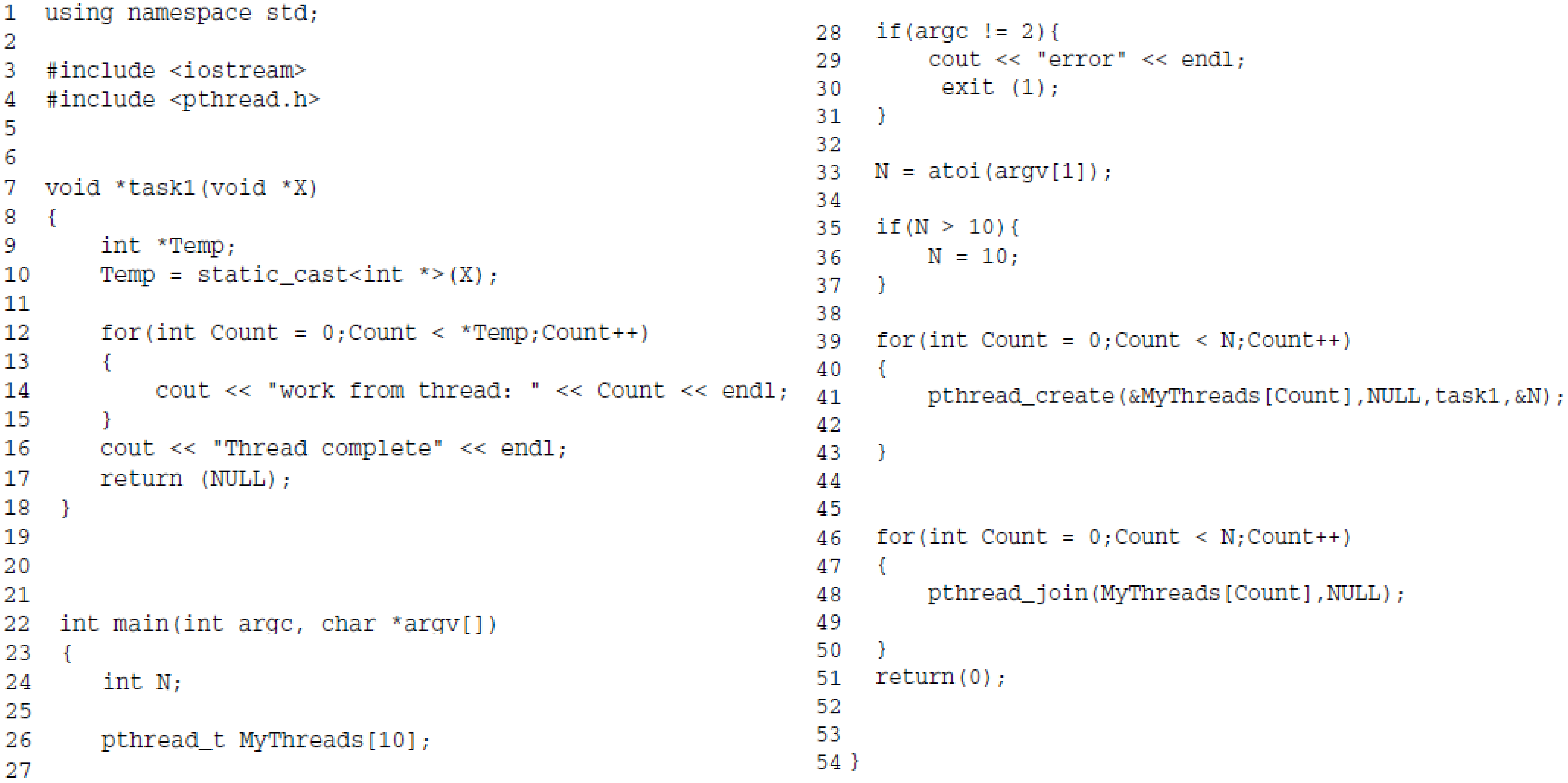
- The *thread* parameter points to a thread handle or thread id of the thread to be created.

- The new thread has the attributes specified by the attribute object *attr*.

- The thread executes the instructions in *start_routine* with the arguments specified by *arg*.

- If the function successfully creates the thread, it returns the thread id and stores the value in thread parameter.

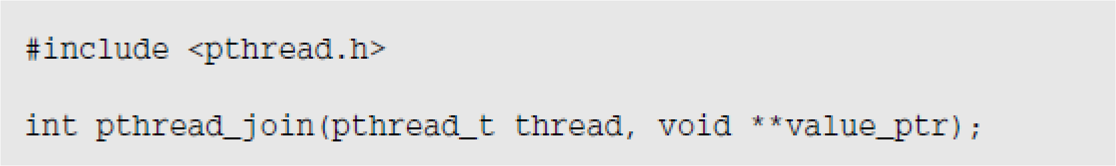
# Passing Arguments to a Thread



```
using namespace std;

#include <iostream>
#include <pthread.h>


void *task1(void *X)
{
    int *Temp;
    Temp = static_cast<int *>(X);

    for(int Count = 0;Count < *Temp;Count++)
    {
        cout << "work from thread: " << Count << endl;
    }
    cout << "Thread complete" << endl;
    return (NULL);
 }



 int main(int argc, char *argv[])
 {
    int N;

    pthread_t MyThreads[10];

    if(argc != 2){
        cout << "error" << endl;
         exit (1);
    }

    N = atoi(argv[1]);

    if(N > 10){
        N = 10;
    }

    for(int Count = 0;Count < N;Count++)
    {
        pthread_create(&MyThreads[Count],NULL,task1,&N);

    }


    for(int Count = 0;Count < N;Count++)
    {
        pthread_join(MyThreads[Count],NULL);

    }
    return(0);


}
```

- If it is necessary to pass multiple arguments to the thread function, you can create a struct with all the required arguments and pass a pointer to that structure to the thread function.

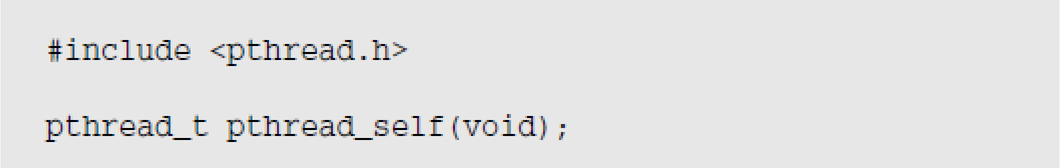
# Joining Threads



```
#include <pthread.h>

int pthread_join(pthread_t thread, void **value_ptr);
```

- pthread_join() is used to join or rejoin flows of control in a process.
- pthread_join() causes the calling thread to suspend its execution until the target thread has terminated:
    - It can be called either by the creator of a thread
    - or by peer threads if the thread handle is global.
- The *thread* parameter is the id of the target thread.
- If the target thread returns successfully, its exit status is stored in *value_ptr*.
- There should be a pthread_join() function called for all joinable threads.
- Behavior is undefined if different peer threads simultaneously call the pthread_join() function on the same thread.

# Getting the Thread Id



```
#include <pthread.h>

pthread_t pthread_self(void);
```

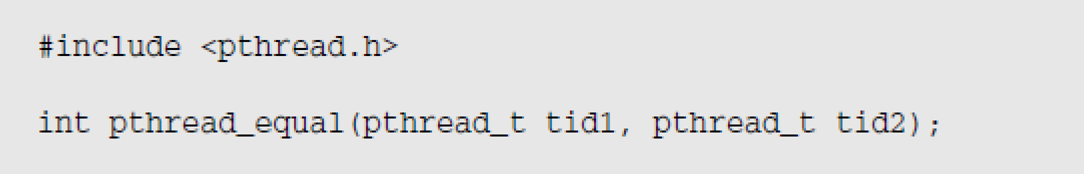
- It returns the thread id of the calling thread, e.g.:

```
pthread_t ThreadId;
ThreadId = pthread_self();
```

- Once the thread has its own id, it can be passed to other threads in the process.

- The thread id is also returned to the calling thread of pthread_create()

# Comparing Thread Ids



```
#include <pthread.h>

int pthread_equal(pthread_t tid1, pthread_t tid2);
```

- Thread ids can be compared but not by using the normal comparison operators.
- You can determine whether two thread ids are equivalent by calling pthread_equal().
- It returns a **nonzero** value if the two thread ids reference the same thread.
- If they reference different threads, it returns **zero**.

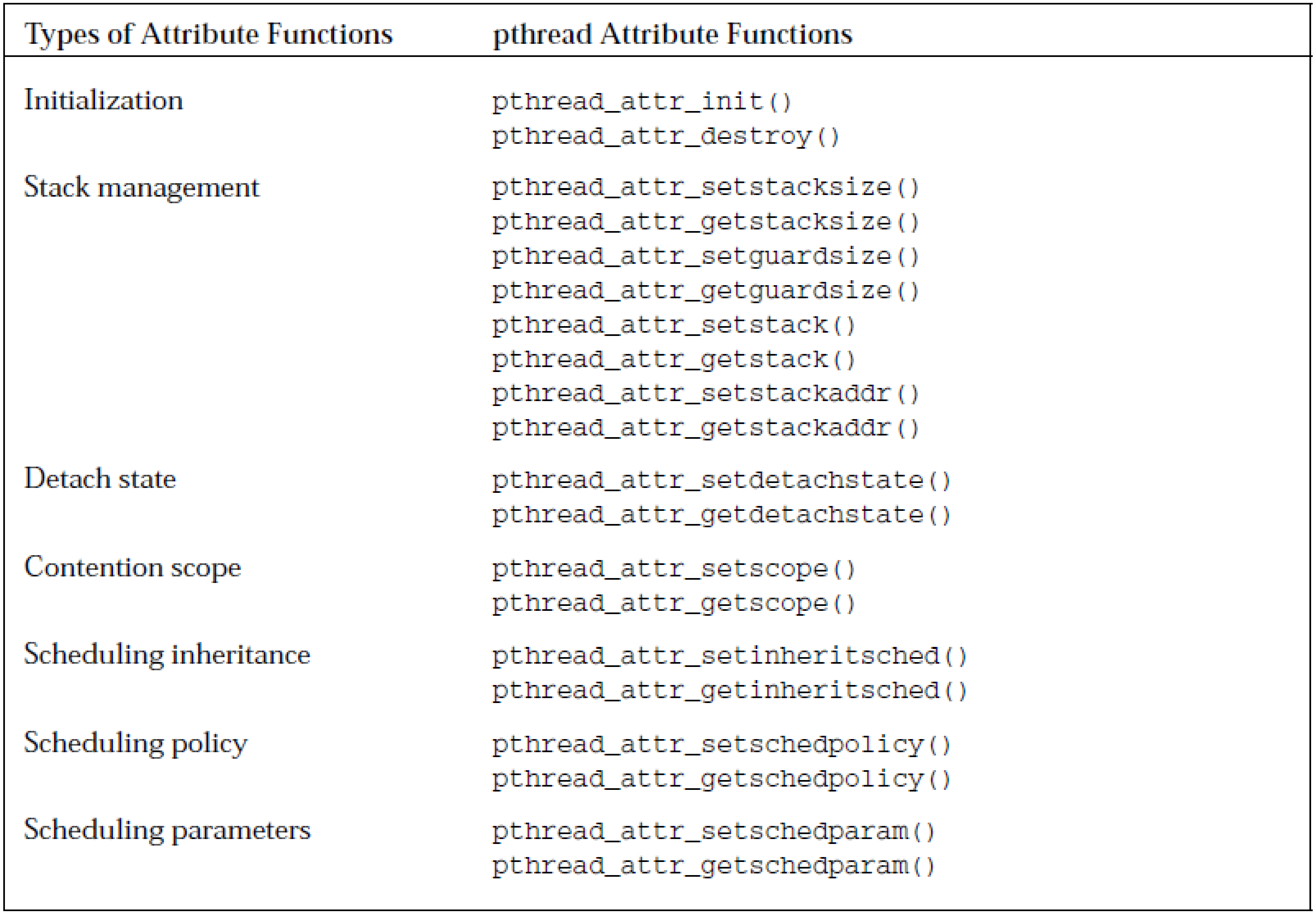
# Using the Pthread Attribute Object

- Threads have a set of attributes that can be specified at the time that the thread is created.

- The set of attributes is encapsulated in an structure whose type is *pthread_attr_t*.

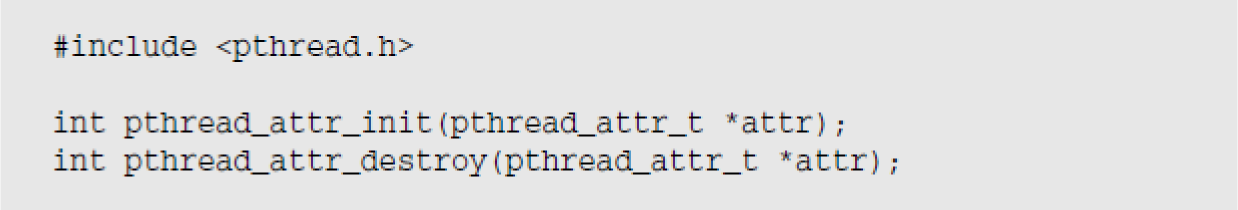
- This structure can be used to set the following thread attributes:
    - Size of the thread's stack
    - Location of the thread's stack
    - Scheduling inheritance, policy, and parameters
    - Whether the thread is detached or joinable
    - Scope of the thread

# Methods Used to Query and to Set the Attribute



| Types of Attribute Functions | pthread Attribute Functions |
|---|---|
| Initialization | `pthread_attr_init()`<br>`pthread_attr_destroy()` |
| Stack management | `pthread_attr_setstacksize()`<br>`pthread_attr_getstacksize()`<br>`pthread_attr_setguardsize()`<br>`pthread_attr_getguardsize()`<br>`pthread_attr_setstack()`<br>`pthread_attr_getstack()`<br>`pthread_attr_setstackaddr()`<br>`pthread_attr_getstackaddr()` |
| Detach state | `pthread_attr_setdetachstate()`<br>`pthread_attr_getdetachstate()` |
| Contention scope | `pthread_attr_setscope()`<br>`pthread_attr_getscope()` |
| Scheduling inheritance | `pthread_attr_setinheritsched()`<br>`pthread_attr_getinheritsched()` |
| Scheduling policy | `pthread_attr_setschedpolicy()`<br>`pthread_attr_getschedpolicy()` |
| Scheduling parameters | `pthread_attr_setschedparam()`<br>`pthread_attr_getschedparam()` |

# Initialize and Destroy Thread Attrbibutes



```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

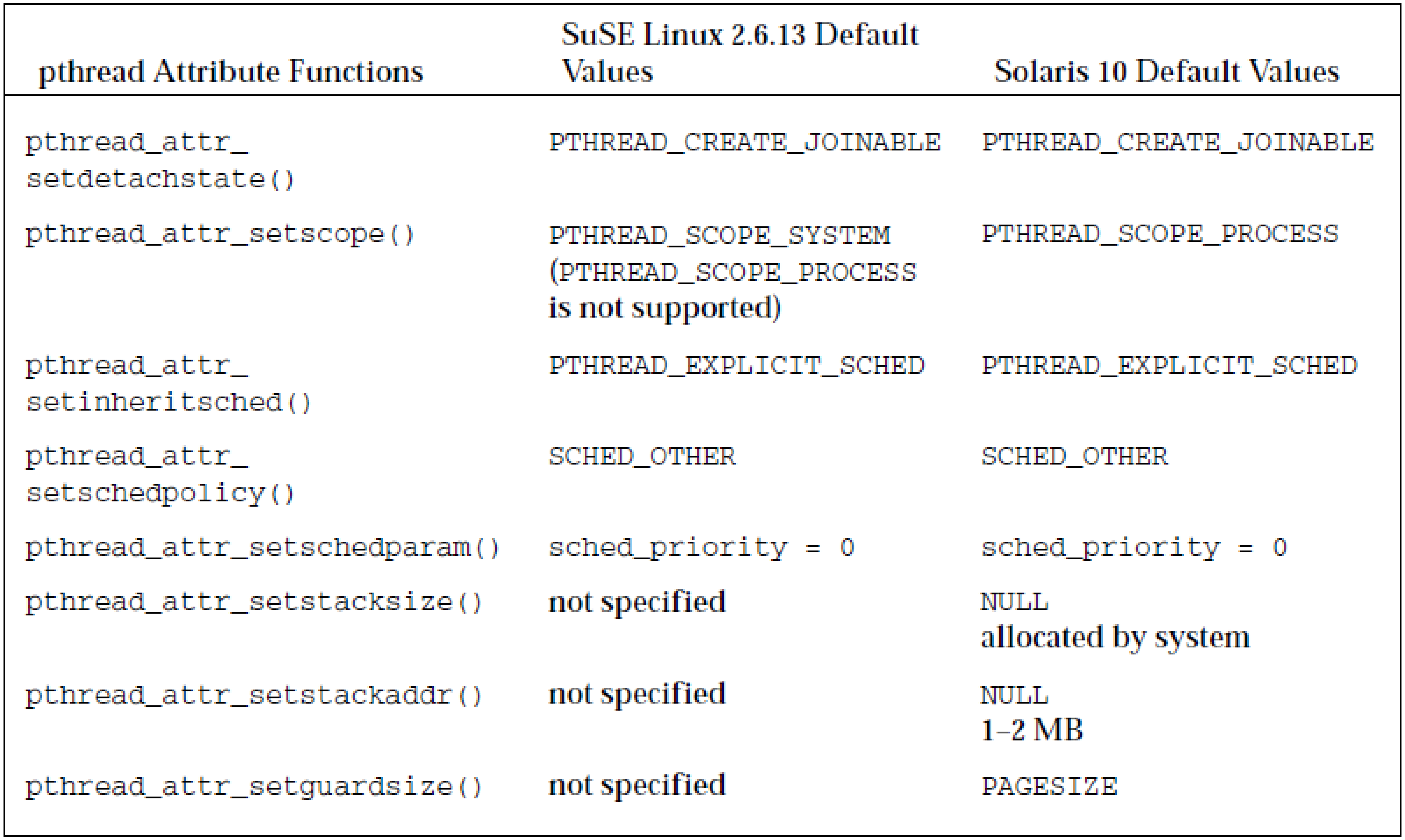
- The pthread_attr_init() initializes a thread attribute object with the default values for all the attributes.

- Once *attr* has been initialized, its attribute values can be changed by using the pthread_attr_set functions listed before.

- The pthread_attr_destroy() function can be used to destroy a pthread_attr_t object specified by *attr*.

- A call to this function deletes any hidden storage associated with the thread attribute object.

# Default Values for the Attribute Object

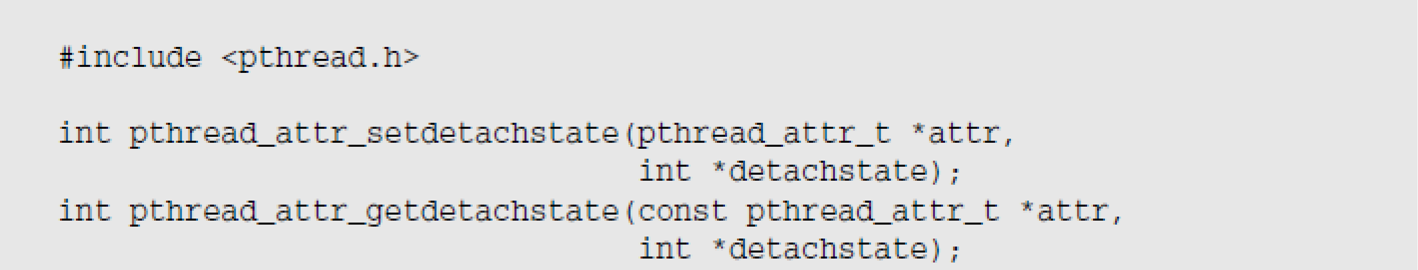


| pthread Attribute Functions | SuSE Linux 2.6.13 Default Values | Solaris 10 Default Values |
|---|---|---|
| `pthread_attr_setdetachstate()` | `PTHREAD_CREATE_JOINABLE` | `PTHREAD_CREATE_JOINABLE` |
| `pthread_attr_setscope()` | `PTHREAD_SCOPE_SYSTEM` (`PTHREAD_SCOPE_PROCESS` is not supported) | `PTHREAD_SCOPE_PROCESS` |
| `pthread_attr_setinheritsched()` | `PTHREAD_EXPLICIT_SCHED` | `PTHREAD_EXPLICIT_SCHED` |
| `pthread_attr_setschedpolicy()` | `SCHED_OTHER` | `SCHED_OTHER` |
| `pthread_attr_setschedparam()` | `sched_priority = 0` | `sched_priority = 0` |
| `pthread_attr_setstacksize()` | not specified | `NULL`<br>allocated by system |
| `pthread_attr_setstackaddr()` | not specified | `NULL`<br>1–2 MB |
| `pthread_attr_setguardsize()` | not specified | `PAGESIZE` |

- If a value is not supported its function returns an error number, for instance in Linux environments PTHREAD_SCOPE_PROCESS is not supported, e.g.:

```
int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope)
```

# Creating Detached Threads Using the Pthread Attribute Object



```
#include <pthread.h>

int pthread_attr_setdetachstate(pthread_attr_t *attr,
                                int *detachstate);
int pthread_attr_getdetachstate(const pthread_attr_t *attr,
                                int *detachstate);
```

- If an exiting thread is not joined with another thread, the exiting thread is said to be *detached.*

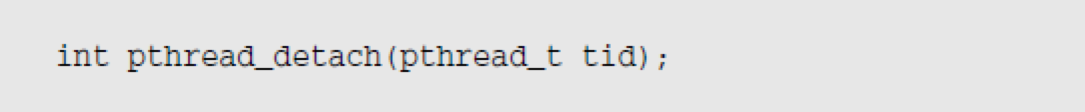
- A pthread_join() cannot be used on a detached thread. If it is used, it returns an error.

- The pthread_attr_setdetachstate() function can be used to set the detachstate attribute of the attribute object.

- The *detachstate* parameter describes the thread as detached or joinable. The detachstate can have one of these values:
  - PTHREAD_CREATE_DETACHED
  - PTHREAD_CREATE_JOINABLE

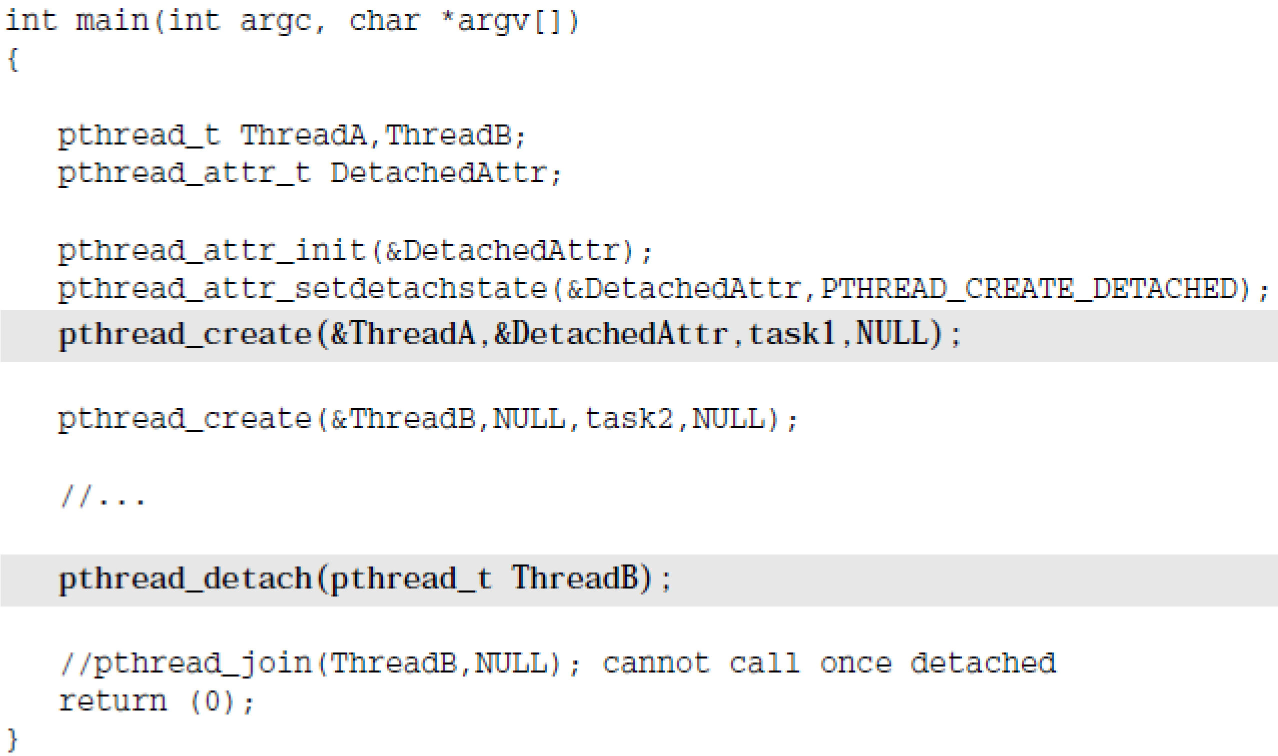
- The pthread_attr_getdetachstate() function returns the detachstate of the attribute object.

# pthread_detach and an Example



```
int pthread_detach(pthread_t tid);
```

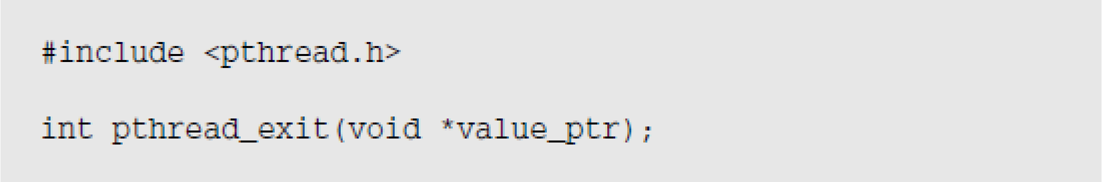
- Threads that are already running can become detached by pthread_detach().



```
int main(int argc, char *argv[])
{

   pthread_t ThreadA,ThreadB;
   pthread_attr_t DetachedAttr;

   pthread_attr_init(&DetachedAttr);
   pthread_attr_setdetachstate(&DetachedAttr,PTHREAD_CREATE_DETACHED);
   pthread_create(&ThreadA,&DetachedAttr,task1,NULL);

   pthread_create(&ThreadB,NULL,task2,NULL);

   //...

   pthread_detach(pthread_t ThreadB);

   //pthread_join(ThreadB,NULL); cannot call once detached
   return (0);
}
```

# Terminating Threads

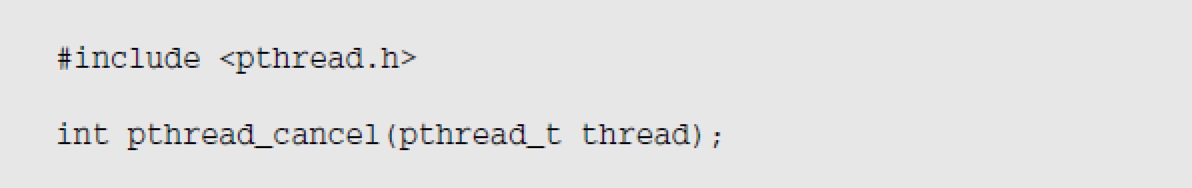
- A thread’s execution can be discontinued by several means:
  - By returning from the execution of its assigned task with or without an exit status or return value
  - By explicitly terminating itself and supplying an exit status
  - By being canceled by another thread in the same address space.
- A thread can (explicitly) self-terminate by calling pthread_exit().



```
#include <pthread.h>

int pthread_exit(void *value_ptr);
```

- When the terminating thread calls pthread_exit(), it is passed the exit status in *value_ptr*.
- The exit status is returned to pthread_join().

# Terminating Peer Threads



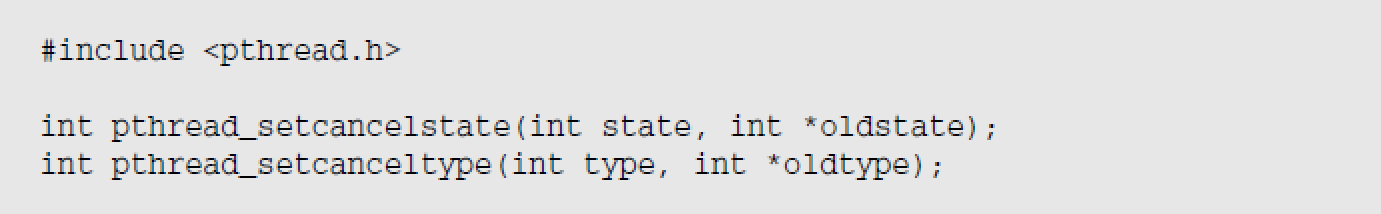
```
#include <pthread.h>

int pthread_cancel(pthread_t thread);
```

- pthread_cancel() create a request to cancel/terminate peer threads. The request can be
    - granted immediately,
    - granted at a later time or
    - even ignored.

- The thread parameter is the thread to be canceled.

# The Cancellability State and the Cancelability Type I.

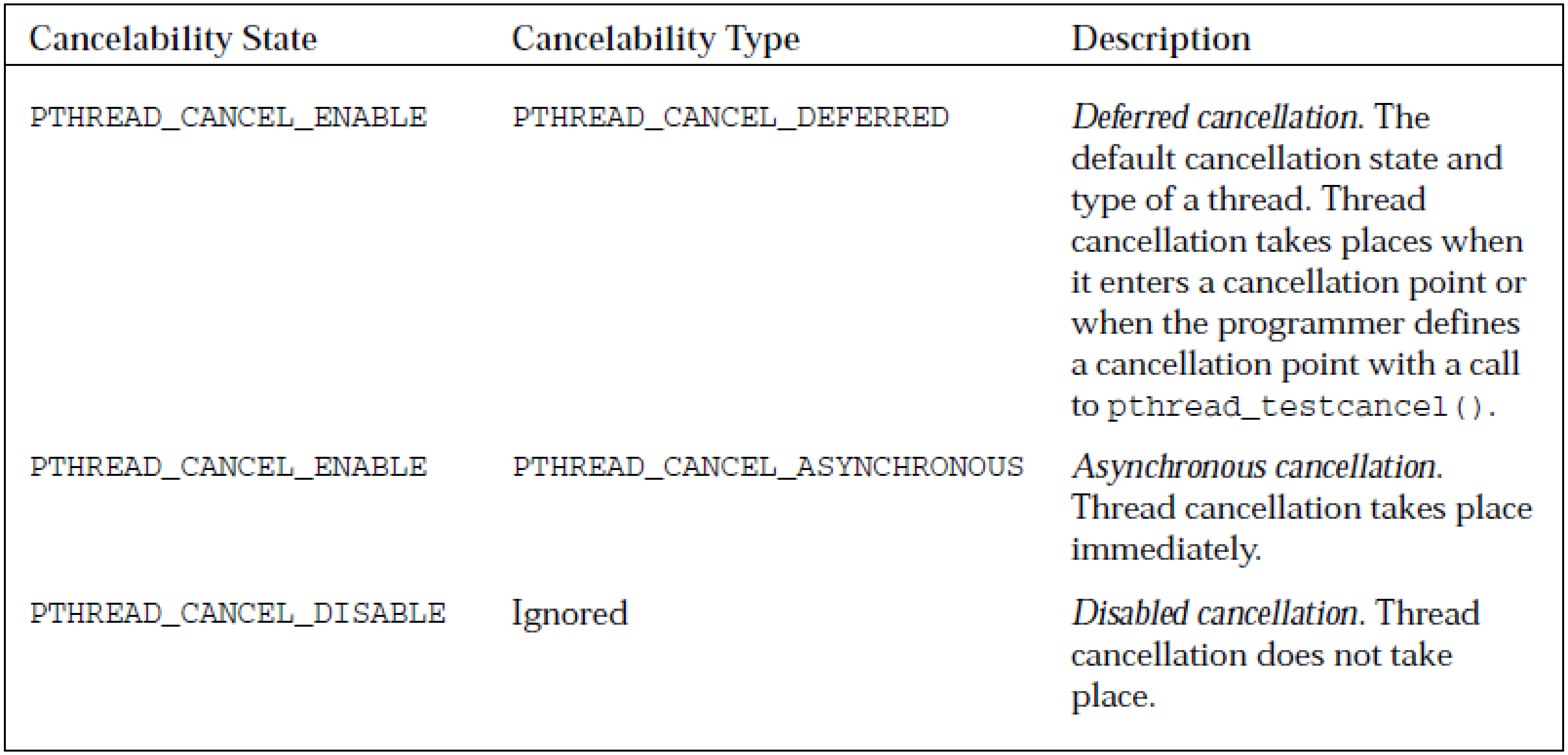
- The cancel type and cancel state of the target thread determines when cancellation actually takes place:
  - The ***cancelability state*** describes the cancel condition of a thread as being *cancelable* or *uncancelable*.
  - A thread's ***cancelability type*** determines the thread's ability to continue after a cancel request.
- The cancelability state and type are dynamically set by the thread itself.



```
#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
```

- pthread_setcancelstate() and pthread_setcanceltype() are used to set the cancelability state and type of the calling thread.

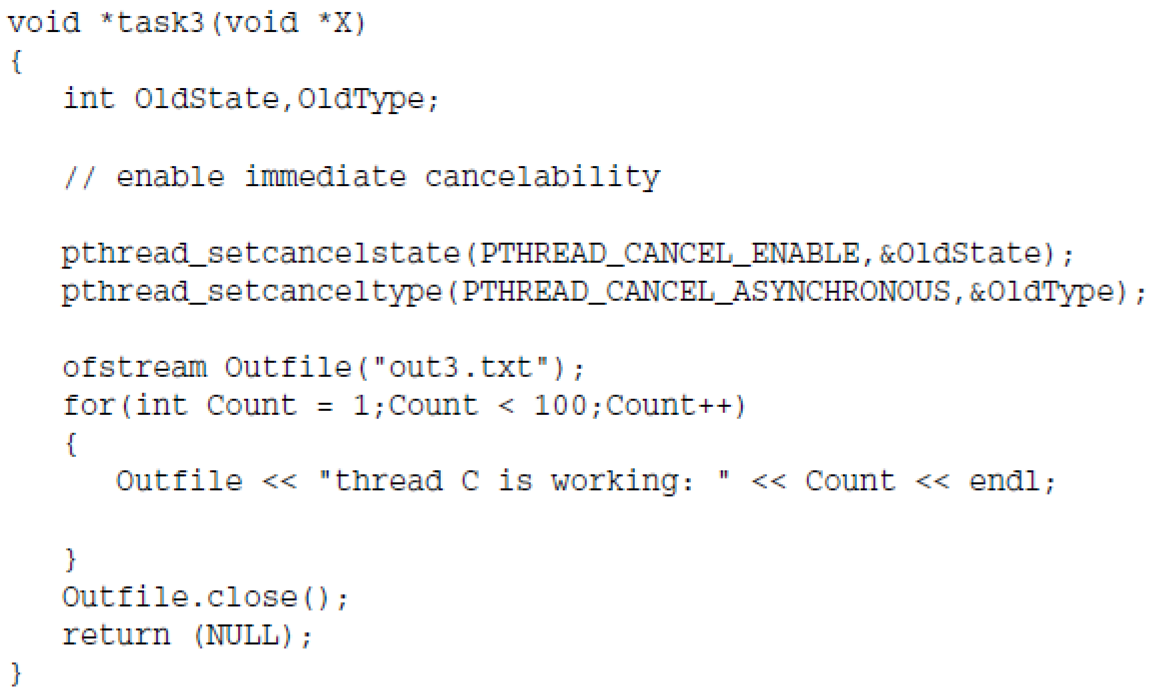
# The Cancellability State and the Cancellability Type II.



| Cancelability State | Cancelability Type | Description |
|---|---|---|
| `PTHREAD_CANCEL_ENABLE` | `PTHREAD_CANCEL_DEFERRED` | *Deferred cancellation.* The default cancellation state and type of a thread. Thread cancellation takes places when it enters a cancellation point or when the programmer defines a cancellation point with a call to `pthread_testcancel()`. |
| `PTHREAD_CANCEL_ENABLE` | `PTHREAD_CANCEL_ASYNCHRONOUS` | *Asynchronous cancellation.* Thread cancellation takes place immediately. |
| `PTHREAD_CANCEL_DISABLE` | Ignored | *Disabled cancellation.* Thread cancellation does not take place. |

- pthread_testcancel() does nothing except process a pending cancellation in a synchronously cancellable thread.
- Certain other functions are implicitly cancellation points as well. These are listed on the pthread_cancel() man page (e.g.: pthread_join()).

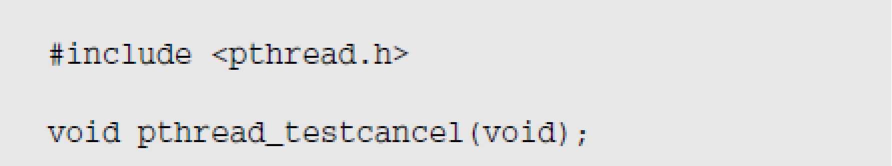
# Cancellation Example



```
void *task3(void *X)
{
   int OldState,OldType;

   // enable immediate cancelability

   pthread_setcancelstate(PTHREAD_CANCEL_ENABLE,&OldState);
   pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS,&OldType);

   ofstream Outfile("out3.txt");
   for(int Count = 1;Count < 100;Count++)
   {
      Outfile << "thread C is working: " << Count << endl;

   }
   Outfile.close();
   return (NULL);
}
```

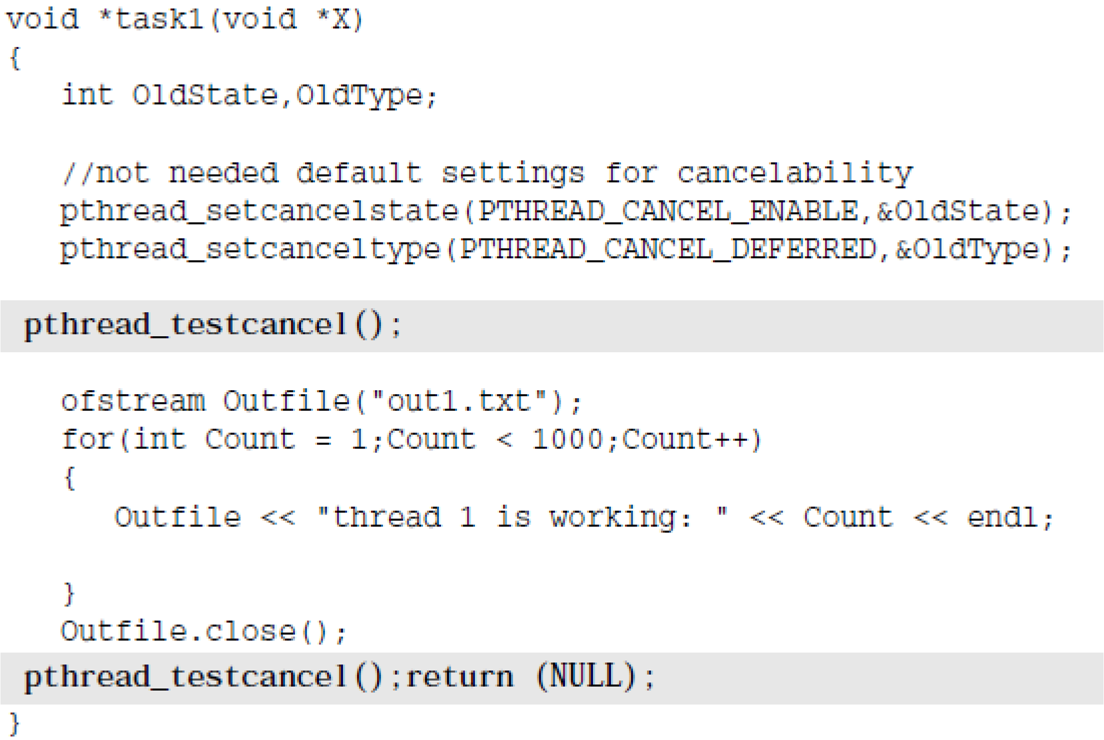
- In Example, cancellation is set to take place immediately.
- So, the thread can open the file and be canceled while it is writing to the file (dangerous and bad practice).

# Cancellation Points Example I.



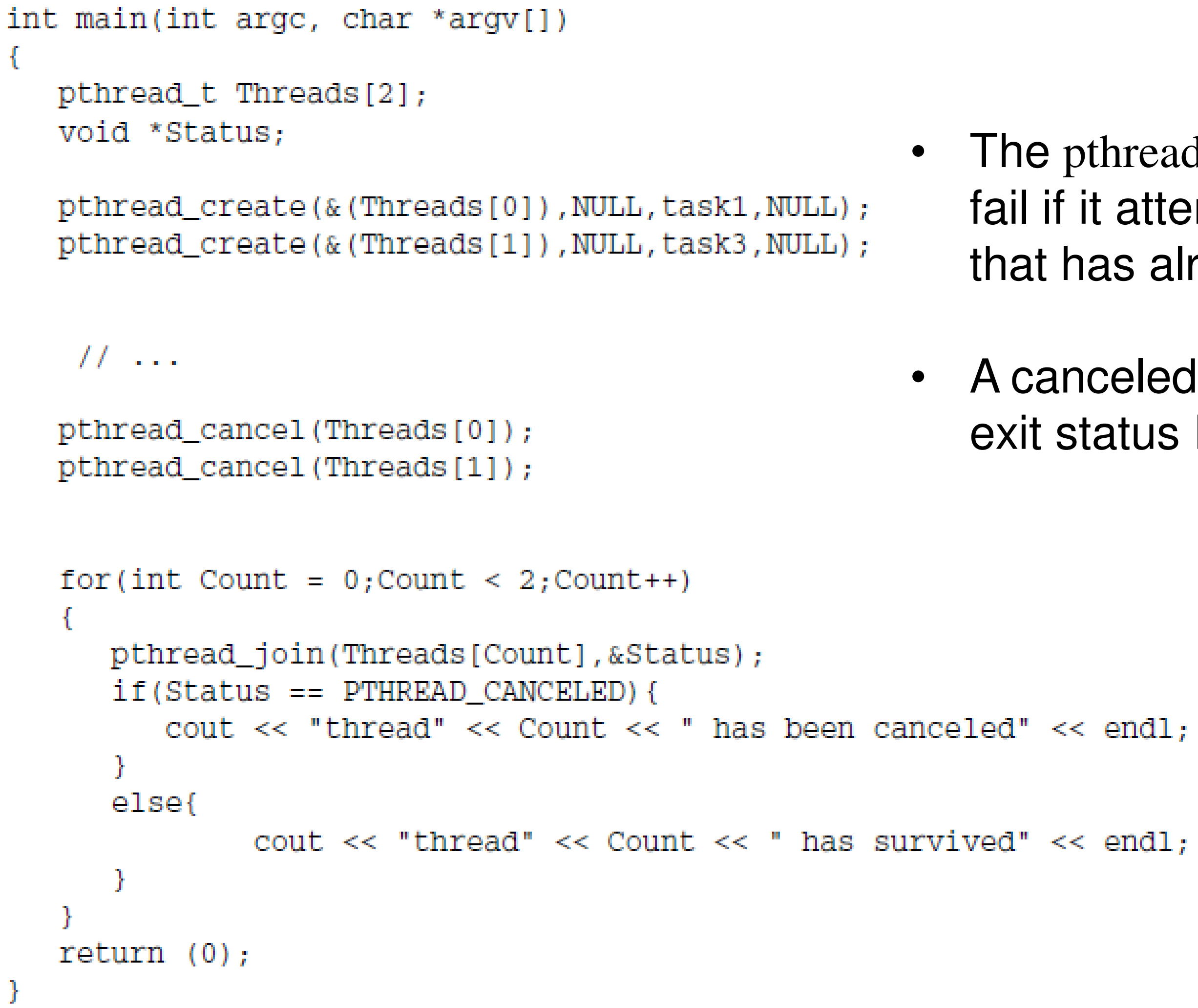
```
#include <pthread.h>

void pthread_testcancel(void);
```

- A cancellation point is a checkpoint where a thread checks if there are any cancellation requests pending and, if so, concedes to termination.



```
void *task1(void *X)
{
   int OldState,OldType;

   //not needed default settings for cancelability
   pthread_setcancelstate(PTHREAD_CANCEL_ENABLE,&OldState);
   pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED,&OldType);

 pthread_testcancel();

   ofstream Outfile("out1.txt");
   for(int Count = 1;Count < 1000;Count++)
   {
      Outfile << "thread 1 is working: " << Count << endl;

   }
   Outfile.close();
 pthread_testcancel();return (NULL);
}
```

# Cancellation Points Example II.



```
int main(int argc, char *argv[])
{
   pthread_t Threads[2];
   void *Status;

   pthread_create(&(Threads[0]),NULL,task1,NULL);
   pthread_create(&(Threads[1]),NULL,task3,NULL);


    // ...

   pthread_cancel(Threads[0]);
   pthread_cancel(Threads[1]);


   for(int Count = 0;Count < 2;Count++)
   {
      pthread_join(Threads[Count],&Status);
      if(Status == PTHREAD_CANCELED){
         cout << "thread" << Count << " has been canceled" << endl;
      }
      else{
              cout << "thread" << Count << " has survived" << endl;
      }
   }
   return (0);
}
```

- The pthread_join() function does not fail if it attempts to join with a thread that has already been terminated.
- A canceled thread may return an exit status PTHREAD_CANCELED.

# Cancellation-Safe Library Functions

- The pthread library defines functions that can serve as cancellation points and are considered **asynchronous cancellation-safe** functions.

- These functions block the calling thread, and while the calling thread is blocked, it is safe to cancel the thread.

- These are the pthread library functions that act as cancellation points:
    - pthread_testcanel()

    - pthread_cond_wait()

    - pthread_timedwait

    - pthread_join()

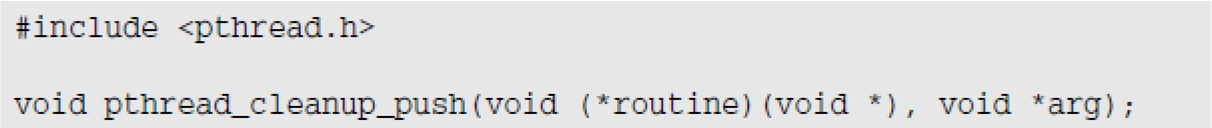
# System Calls as Cancellation Points

- Some of the POSIX system calls that are required to be cancellation points (e.g.: connect(), accept(), sleep(), system(), read(), write, etc.).
- These POSIX functions are safe to be used as deferred cancellation points, but they may not be safe for asynchronous cancellation.
- A library call that is not asynchronously safe that is canceled during execution can cause library data to be left in an incompatible state.

# Cleaning Up before Termination I.

- We mentioned earlier that a thread may need to perform some final processing before it is terminated.

- A ***cleanup stack*** is associated with every thread which contains pointers to routines that are to be executed during the cancellation process.



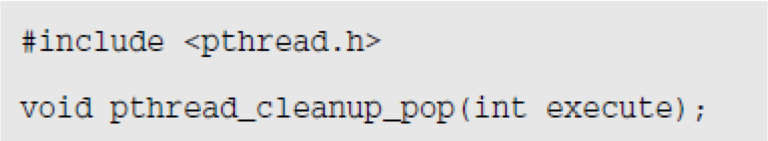
```
#include <pthread.h>

void pthread_cleanup_push(void (*routine)(void *), void *arg);
```

- The function pushes a pointer to the routine to the cleanup stack.

- The function routine is called with the arg parameter when the thread exits under these circumstances:
  - When calling pthread_exit(),
  - When the thread concedes to a termination request and
  - When the thread explicitly calls pthread_cleanup_pop() with a nonzero value for execute.

# Cleaning Up before Termination II.

Karoly.Bosa@jku.at



```
#include <pthread.h>

void pthread_cleanup_pop(int execute);
```

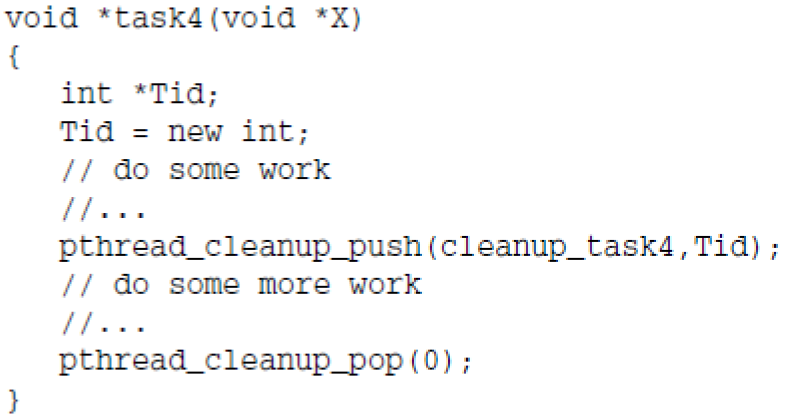
- The pthread_cleanup_pop() removes routine’s pointer from the top of the calling thread’s cleanup stack.

- The execute parameter can have a value of 1 or 0:

- If 1 , the thread executes routine even if it is not being terminated. The thread continues execution from the point after the call to this function.

- If the value is 0 , the pointer is removed from the top of the stack without executing.

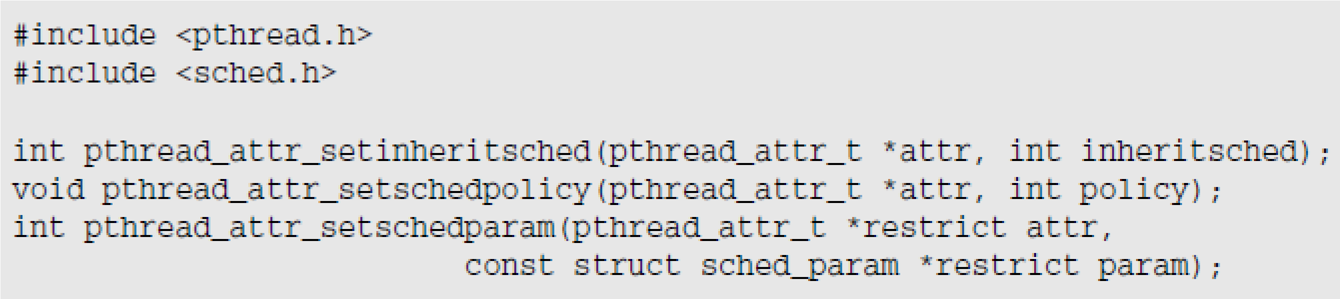
- For each push, there needs to be a pop if the clean up routine become obsolete (because the relevant code part finished its activity).



```
void *task4(void *X)
{
   int *Tid;
   Tid = new int;
   // do some work
   //...
   pthread_cleanup_push(cleanup_task4,Tid);
   // do some more work
   //...
   pthread_cleanup_pop(0);
}
```

# Setting Thread Scheduling and Priorities I.

- The scheduling policy of a thread or group of threads can be set by an attribute object using these functions:



```
#include <pthread.h>
#include <sched.h>

int pthread_attr_setinheritsched(pthread_attr_t *attr, int inheritsched);
void pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_setschedparam(pthread_attr_t *restrict attr,
                         const struct sched_param *restrict param);
```

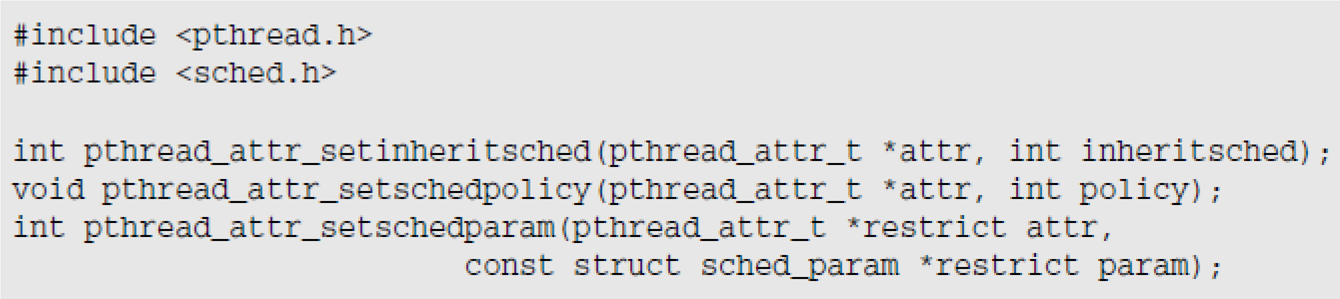
- pthread_attr_setinheritsched() is used to determine how the thread's scheduling attributes are set, by inheriting the scheduling attributes either from the creator thread or from an attribute object.

- *inheritsched* can have one of these values:
  - PTHREAD_INHERIT_SCHED : Thread scheduling attributes are inherited from the creator thread, and any scheduling attributes of the *attr* are ignored.

  - PTHREAD_EXPLICIT_SCHED :Thread scheduling attributes are set to the scheduling attributes of the attribute object *attr*.

# Setting Thread Scheduling and Priorities II.

- The scheduling policy of a thread or group of threads can be set by an attribute object using these functions:



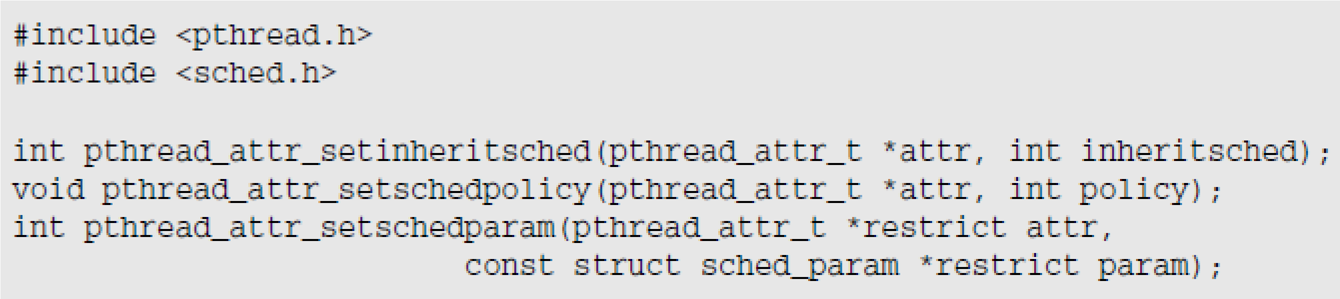
```
#include <pthread.h>
#include <sched.h>

int pthread_attr_setinheritsched(pthread_attr_t *attr, int inheritsched);
void pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_setschedparam(pthread_attr_t *restrict attr,
                         const struct sched_param *restrict param);
```

- pthread_attr_setschedpolicy() sets the scheduling policy of the thread attribute object *attr*.

- *policy* value can be one of the following defined in the <sched.h> header:
  - SCHED_FIFO : First-In, First-Out scheduling,
  - SCHED_RR Round robin scheduling and
  - SCHED_OTHER : Another scheduling policy (implementation - defined). By default, this is the scheduling policy of any newly created thread.

# Setting Thread Scheduling and Priorities III.

- The scheduling policy of a thread or group of threads can be set by an attribute object using these functions:



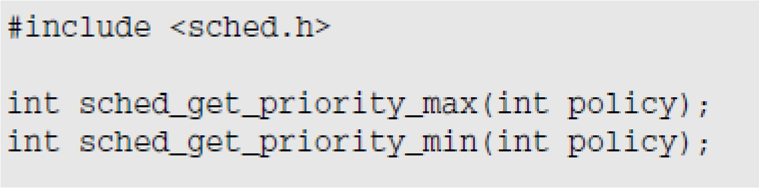
```
#include <pthread.h>
#include <sched.h>

int pthread_attr_setinheritsched(pthread_attr_t *attr, int inheritsched);
void pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_setschedparam(pthread_attr_t *restrict attr,
                         const struct sched_param *restrict param);
```

- pthread_attr_setschedparam() to set the scheduling parameters of the attribute object *attr* used by the scheduling policy.

- *param* is a structure that contains the parameters. The sched_param structure has at least this data member defined:

```
struct sched_param {
        int sched_priority;
        //...
};
```

# Query the Priority Interval



```
#include <sched.h>

int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
```

- Both functions are passed the scheduling policy *policy* for which the priority values are requested, and
- Both return either the maximum or minimum priority values for the scheduling policy.

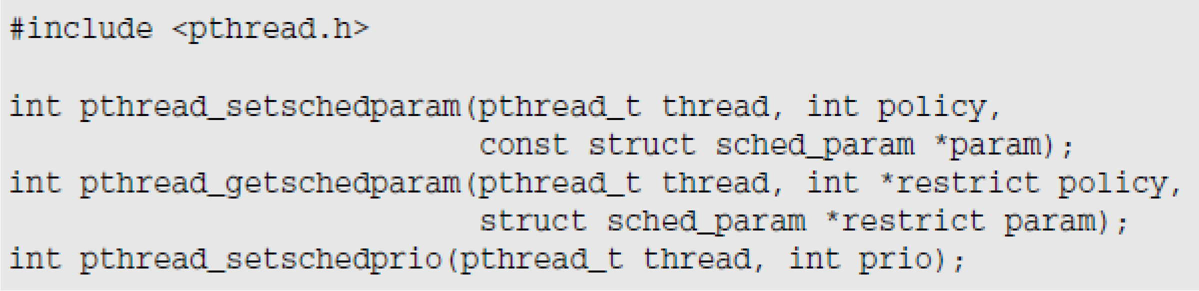
# Example : Setting the scheduling policy and priority of a thread.

```
#include <pthread.h>
#include <sched.h>

//...

pthread_t ThreadA;
pthread_attr_t SchedAttr;
sched_param SchedParam;
int MidPriority,MaxPriority,MinPriority;
```

With these methods, the scheduling policy and priority are set in the thread attribute object before the thread is created or running.

```
int main(int argc, char *argv[])
{
   //...

   // Step 1: initialize attribute object
   pthread_attr_init(&SchedAttr);

   // Step 2: retrieve min and max priority values for scheduling policy
   MinPriority = sched_get_priority_max(SCHED_RR);
   MaxPriority = sched_get_priority_min(SCHED_RR);

   // Step 3: calculate priority value
   MidPriority = (MaxPriority + MinPriority)/2;

   // Step 4: assign priority value to sched_param structure
   SchedParam.sched_priority = MidPriority;

   // Step 5: set attribute object with scheduling parameter
   pthread_attr_setschedparam(&SchedAttr,&SchedParam);

   // Step 6: set scheduling attributes to be determined by attribute object
   pthread_attr_setinheritsched(&SchedAttr,PTHREAD_EXPLICIT_SCHED);

   // Step 7: set scheduling policy
   pthread_attr_setschedpolicy(&SchedAttr,SCHED_RR);

   // Step 8: create thread with scheduling attribute object
   pthread_create(&ThreadA,&SchedAttr,task1,NULL);

   //...
}
```

# Dynamically Changing the Scheduling Policy and Priority



```
#include <pthread.h>

int pthread_setschedparam(pthread_t thread, int policy,
                          const struct sched_param *param);
int pthread_getschedparam(pthread_t thread, int *restrict policy,
                          struct sched_param *restrict param);
int pthread_setschedprio(pthread_t thread, int prio);
```

- pthread_setschedparam() sets both the scheduling policy and priority of a thread directly without the use of an attribute object.

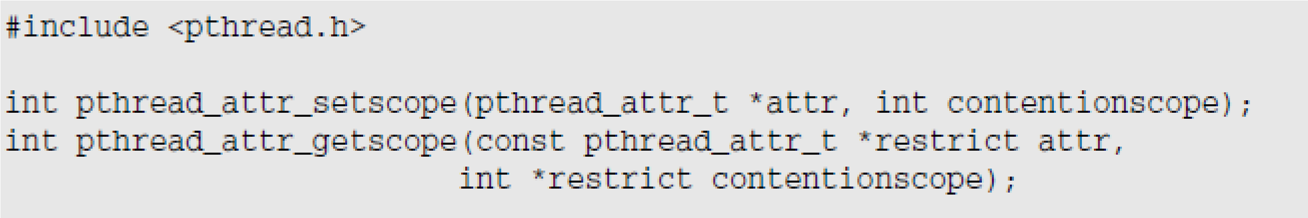
- The pthread_getschedparam() returns the scheduling policy and scheduling parameters.

- The pthread_setschedprio() is used to set the scheduling priority of an executing thread.

# Setting Contention Scope of a Thread

- The contention scope of the thread determines which set of threads a thread competes with for processor usage (systemwide).

- The contention scope of a thread is set by the thread attribute object.



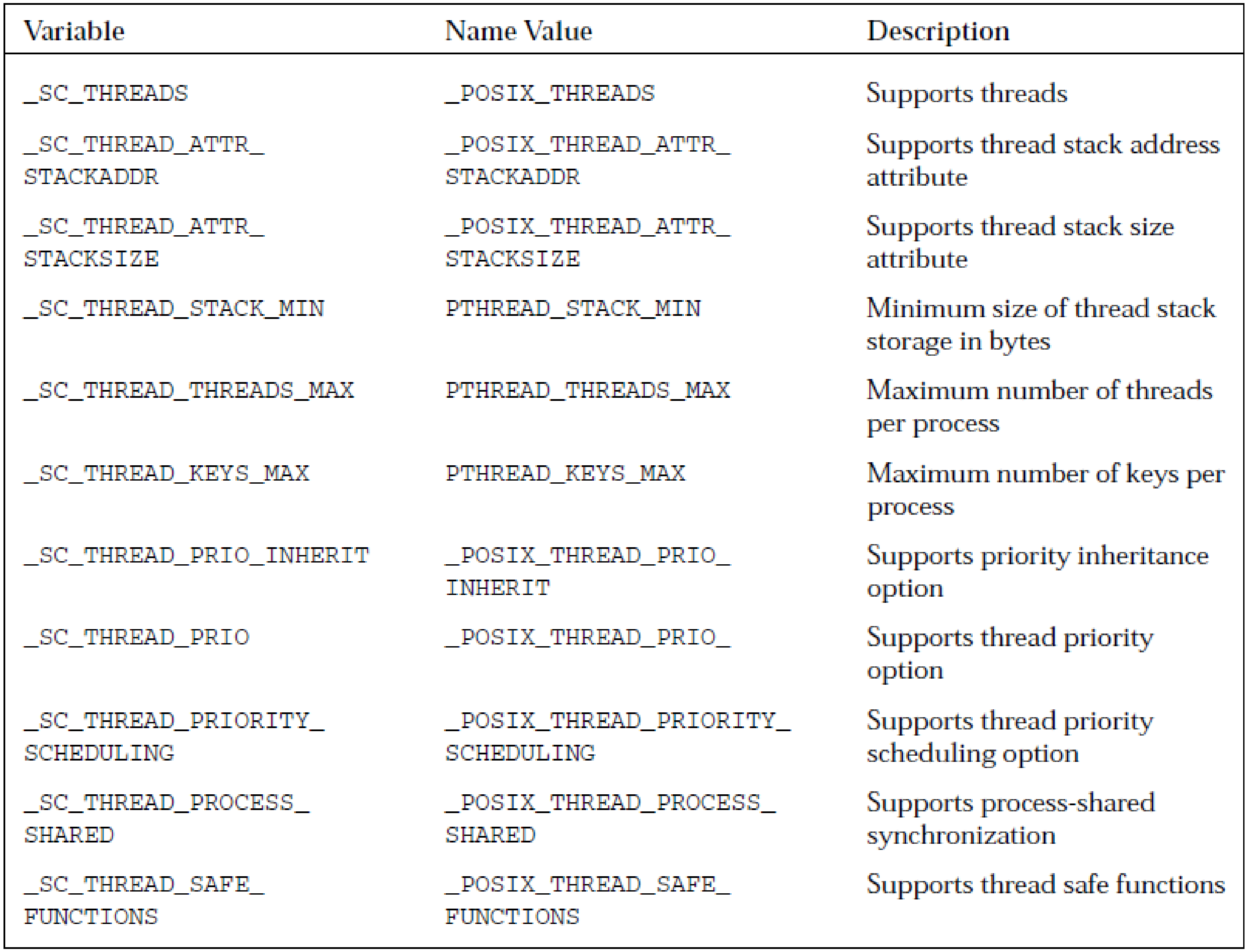
```
#include <pthread.h>

int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);
int pthread_attr_getscope(const pthread_attr_t *restrict attr,
                          int *restrict contentionscope);
```

- The pthread_attr_setscope() sets the contention scope property of the thread attribute object specified by *attr*.

- *contentionscope* can have these values:
    - PTHREAD_SCOPE_SYSTEM : System scheduling contention scope
    - PTHREAD_SCOPE_PROCESS : Process scheduling contention scope

- pthread_attr_getscope() returns the contention scope attribute from the thread attribute object specified by the *attr* .

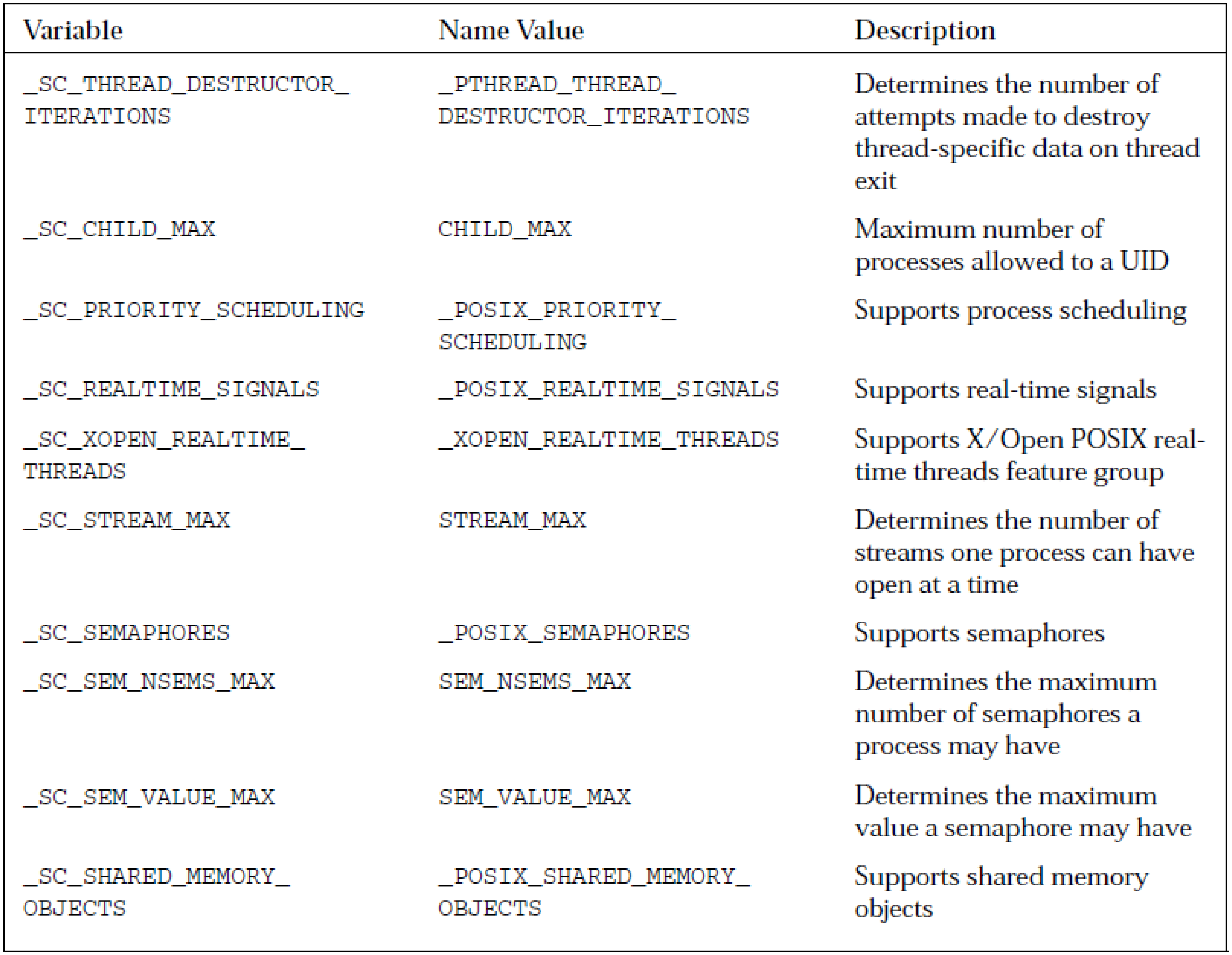
# Using sysconf() I.



| Variable | Name Value | Description |
|---|---|---|
| _SC_THREADS | _POSIX_THREADS | Supports threads |
| _SC_THREAD_ATTR_STACKADDR | _POSIX_THREAD_ATTR_STACKADDR | Supports thread stack address attribute |
| _SC_THREAD_ATTR_STACKSIZE | _POSIX_THREAD_ATTR_STACKSIZE | Supports thread stack size attribute |
| _SC_THREAD_STACK_MIN | PTHREAD_STACK_MIN | Minimum size of thread stack storage in bytes |
| _SC_THREAD_THREADS_MAX | PTHREAD_THREADS_MAX | Maximum number of threads per process |
| _SC_THREAD_KEYS_MAX | PTHREAD_KEYS_MAX | Maximum number of keys per process |
| _SC_THREAD_PRIO_INHERIT | _POSIX_THREAD_PRIO_INHERIT | Supports priority inheritance option |
| _SC_THREAD_PRIO | _POSIX_THREAD_PRIO_ | Supports thread priority option |
| _SC_THREAD_PRIORITY_SCHEDULING | _POSIX_THREAD_PRIORITY_SCHEDULING | Supports thread priority scheduling option |
| _SC_THREAD_PROCESS_SHARED | _POSIX_THREAD_PROCESS_SHARED | Supports process-shared synchronization |
| _SC_THREAD_SAFE_FUNCTIONS | _POSIX_THREAD_SAFE_FUNCTIONS | Supports thread safe functions |

**For instance:** if (PTHREAD_STACK_MIN == (sysconf(_SC_THREAD_STACK_MIN))){ //... }

# Using sysconf() II.



| Variable | Name Value | Description |
|---|---|---|
| _SC_THREAD_DESTRUCTOR_ITERATIONS | _PTHREAD_THREAD_DESTRUCTOR_ITERATIONS | Determines the number of attempts made to destroy thread-specific data on thread exit |
| _SC_CHILD_MAX | CHILD_MAX | Maximum number of processes allowed to a UID |
| _SC_PRIORITY_SCHEDULING | _POSIX_PRIORITY_SCHEDULING | Supports process scheduling |
| _SC_REALTIME_SIGNALS | _POSIX_REALTIME_SIGNALS | Supports real-time signals |
| _SC_XOPEN_REALTIME_THREADS | _XOPEN_REALTIME_THREADS | Supports X/Open POSIX real-time threads feature group |
| _SC_STREAM_MAX | STREAM_MAX | Determines the number of streams one process can have open at a time |
| _SC_SEMAPHORES | _POSIX_SEMAPHORES | Supports semaphores |
| _SC_SEM_NSEMS_MAX | SEM_NSEMS_MAX | Determines the maximum number of semaphores a process may have |
| _SC_SEM_VALUE_MAX | SEM_VALUE_MAX | Determines the maximum value a semaphore may have |
| _SC_SHARED_MEMORY_OBJECTS | _POSIX_SHARED_MEMORY_OBJECTS | Supports shared memory objects |

# Thread Safety and Libraries

Karoly.Bosa@jku.at

- A computer program or routine is described as **reentrant** if it can be safely called again before its previous invocation has been completed.

- A library is thread safe or reentrant when its functions may be called by more than one thread at a time (and work correctly) without requiring any other action.

- If the functions are not thread safe, then this means the functions:
  - Contain static variables
  - Access global data
  - Are not reentrant

- The POSIX standard defines several functions as reentrant. They are easily identified by a **_r** attached to the function name of the non-reentrant counterpart (e.g.: getgrgid_r(), getgrnam_r(), getpwuid_r(), sterror_r(), etc.).