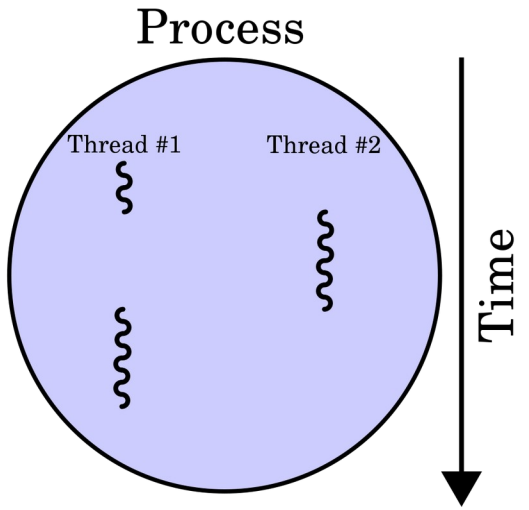
INSTITUTO POLITÉCNICO DE BRAGANÇA
Escola Superior de Tecnologia e Gestão

POSIX Threads Programming



High Performance Computing

Master in Informatics

José Rufino [rufino@ipb.pt]
Escola Superior de Tecnologia e Gestão
Instituto Politécnico de Bragança, Portugal

November 2025, v0.8.2

Outline

- #Abstract
- #Pthreads Overview
- #The Pthreads API
- #Thread Management
- #Mutex Variables
- #Read-Write Locks
- #Condition Variables
- #Barriers
- #Appendix

Disclaimer: this presentation is an edited and expanded version of the tutorial https://hpc-tutorials.llnl.gov/posix/ (Blaise Barney, Lawrence Livermore National Laboratory)

Abstract

Abstract

- In **shared memory multiprocessor** architectures, **threads** can be used to implement **parallelism**
- Historically, hardware vendors have implemented their own proprietary versions of threads, making **portability** a concern
- **UNIX** systems: a standard **C** language threads programming interface has been specified - **IEEE POSIX 1003.1c** standard
- Implementations that adhere to this standard are referred to as **POSIX threads**, or **Pthreads**

- Pre-Requisites (a – mandatory; b – highly desirable)
  - (a) proficiency in the **C** programming language
  - (b) proficiency in the UNIX/Linux **command line interface**
  - (b) proficiency in basic UNIX/Linux **system programming**

Pthreads Overview

What is a Thread ?

- **Thread**: “an **independent stream of instructions** that can be **scheduled** to run by the operating system”

- **View of a software developer**: a thread is like a "**procedure**" that runs independently from its main program
  - imagine a main program (a.out) with some procedures
  - imagine all of these procedures being able to be scheduled to run simultaneously and/or independently by the OS
  - this would describe a "**multi-threaded**" program ...

- To better understand the thread concept and how it works, one first needs to understand the same for **UNIX processes** ...

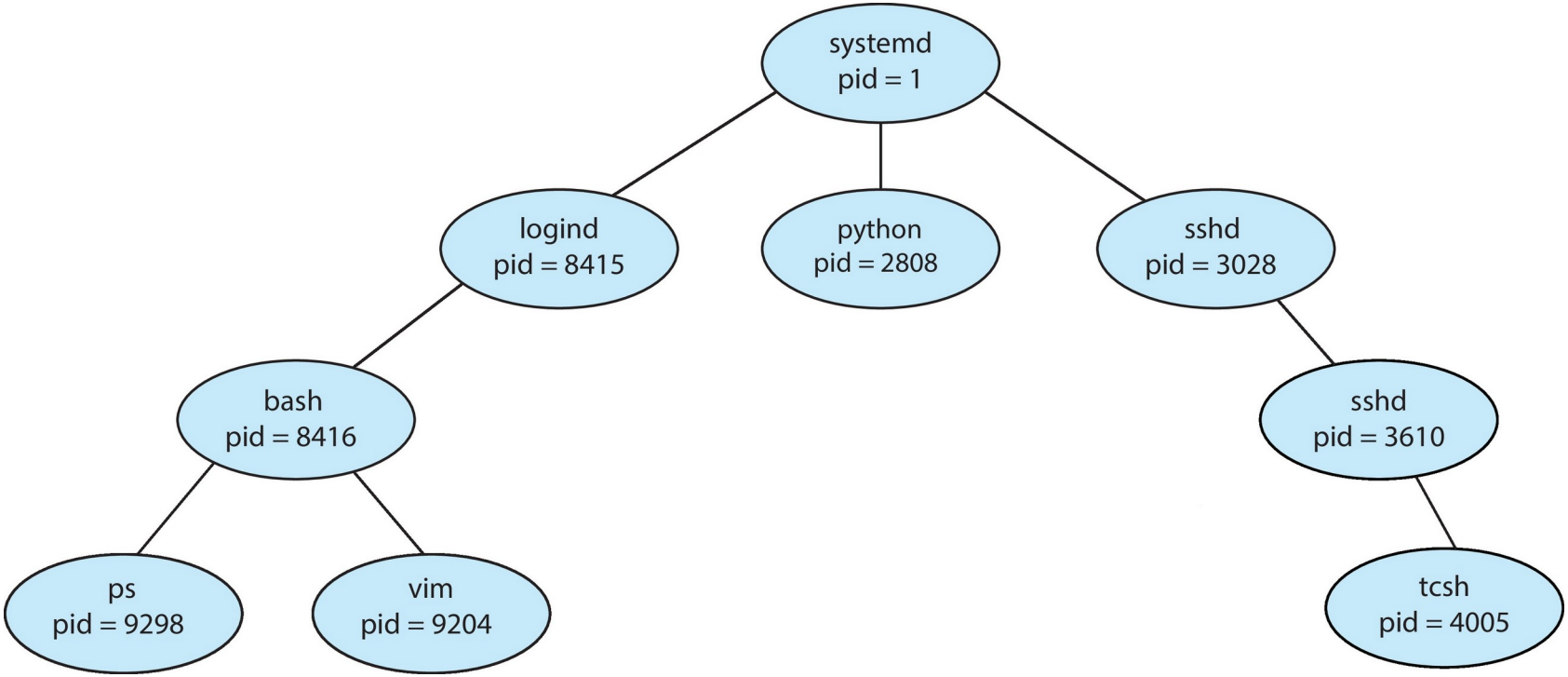
What is a Thread ?

- A **process** is created every time a new program is launched
  - This involves cloning (***forking***) a pre-existing process (parent)
  - The new (child) process reuses/replaces the inherited program

- A **process** includes information on the resources used by its program, as well as values of process specific attributes (*):
  - (*) Process ID, process group ID, user ID, and group ID
  - (*) Environment; Working directory
  - (*) Program state (waiting, running, blocked, ...)
  - Program instructions; Registers; Stack; Heap
  - File descriptors; Signal actions; Shared libraries
  - Inter-Process Communication (IPC) resources (message queues, pipes, semaphores, shared memory, ...)

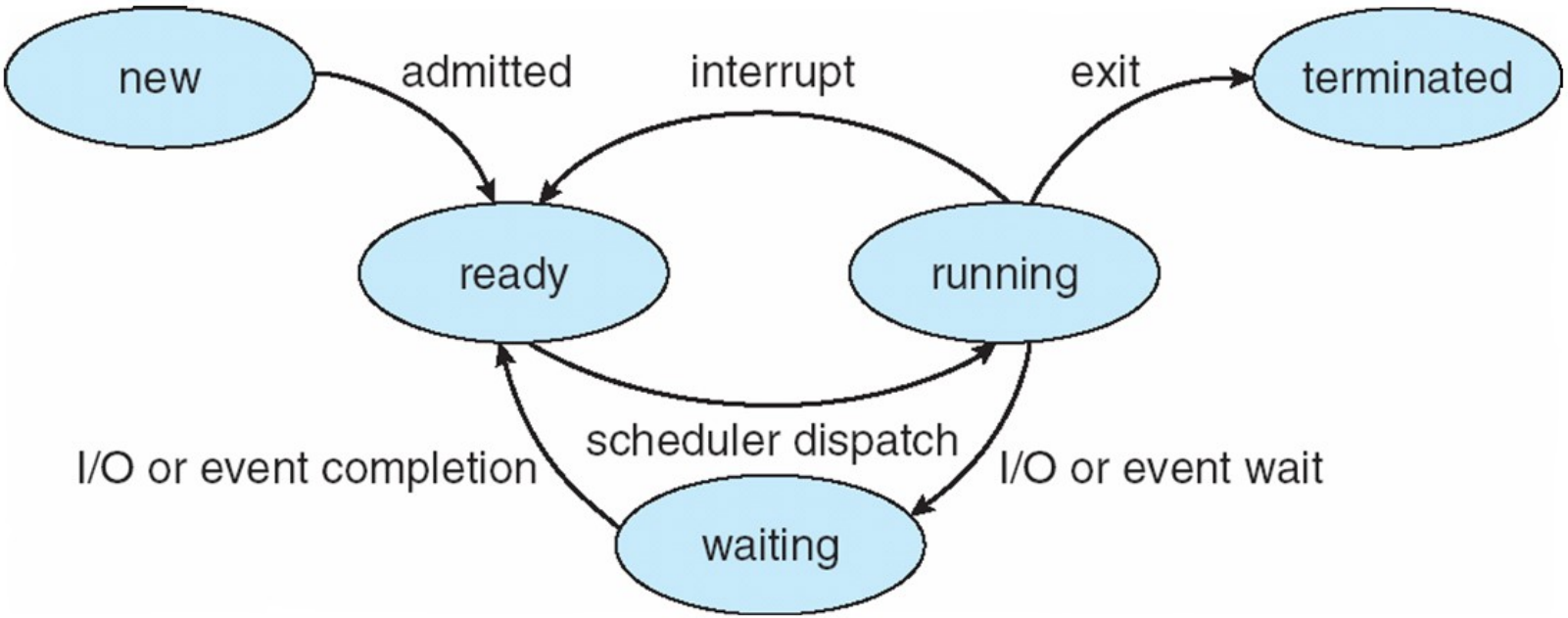
What is a Thread ?

- Process **Hierarchy**



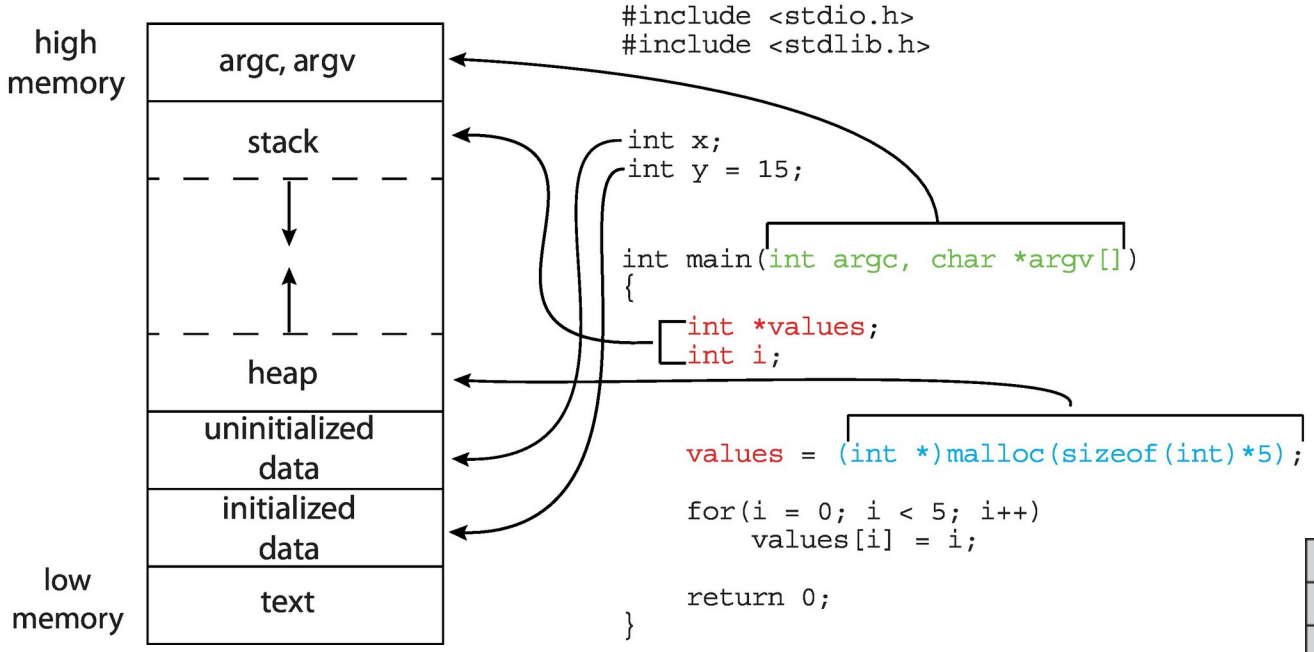
What is a Thread ?

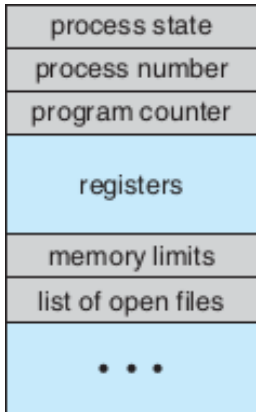
- Process **Life-Cycle**



What is a Thread ?

- Process **User Address Space**





- everything else is kept at the **Process Control Block** (this sits in a memory area accessible only by the OS)

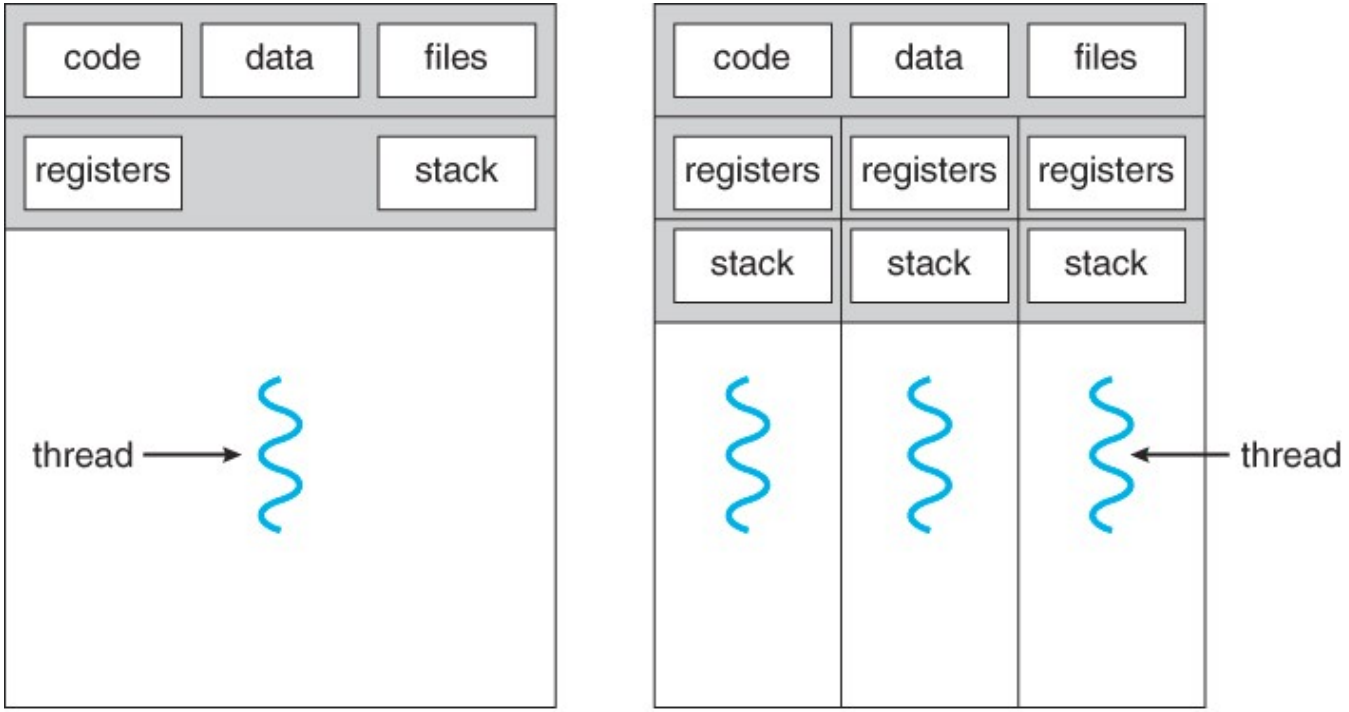
What is a Thread ?

- Threads are able to be scheduled by the operating system and run as independent entities because they **duplicate** only the **bare minimum resources enabling them to execute**

- The **independent flow of control** of a thread is accomplished because a thread maintains its own:
  - Stack pointer
  - Registers
  - Scheduling properties (such as policy or priority)
  - Set of pending and blocked signals
  - Thread specific data

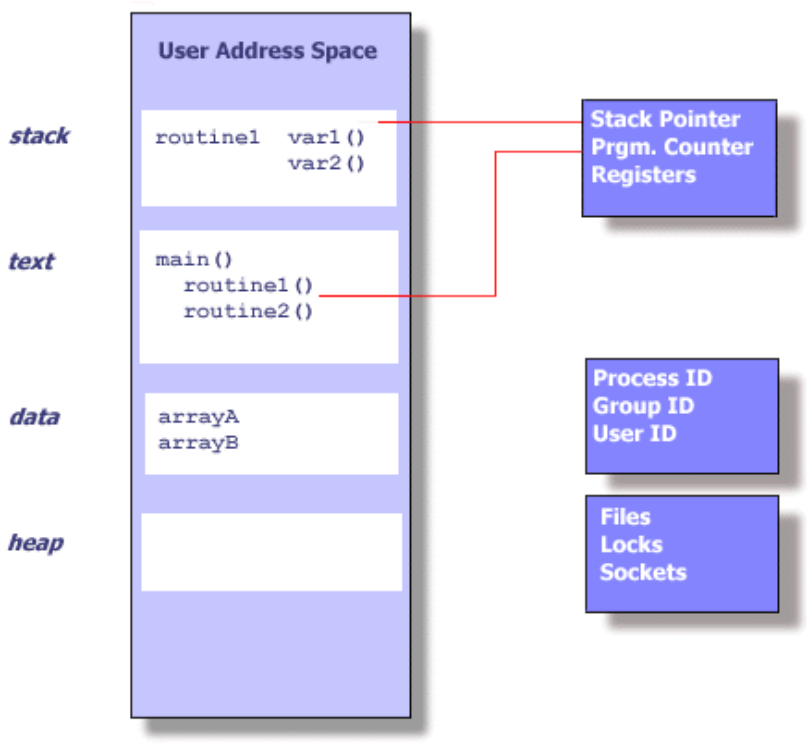
What is a Thread ?

- Threads exist within a process
  - a process has at least the `main` thread
  - any thread may create other threads

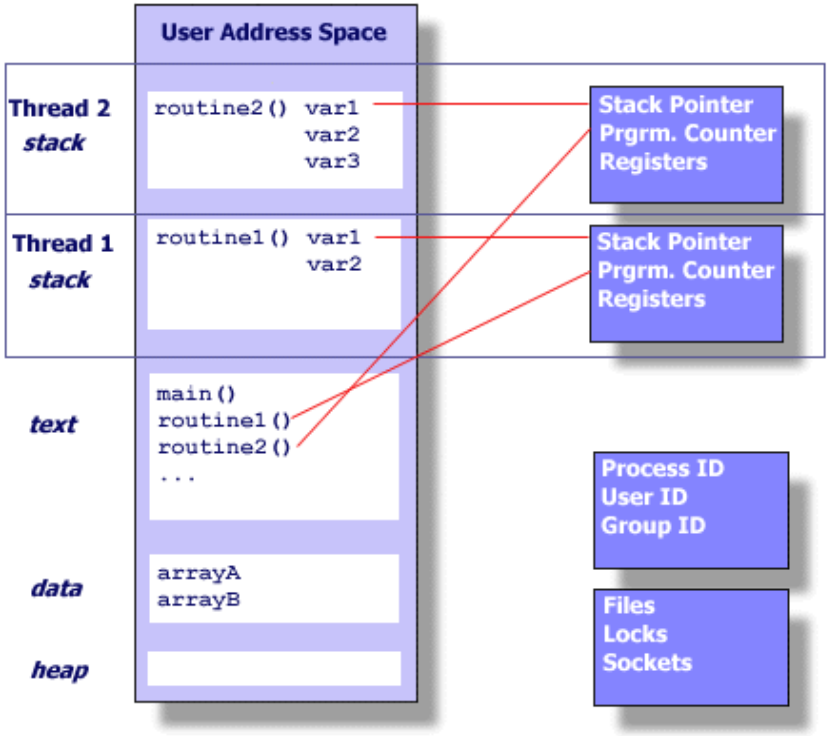


What is a Thread ?

- Threads exist within a process - example



single-threaded process



dual-threaded process

What is a Thread ?

- In summary, in the UNIX environment a thread:
  - Exists within a process and uses the process resources
  - Has its own independent flow of control as long as its parent process exists and the OS supports it
  - Duplicates only the essential resources it needs to be independently schedulable
  - May share the process resources with other threads of that process, that act independently or dependently
  - Is able to create other threads (but no hierachy is enforced)
  - Dies if the parent process dies - or something similar
  - Is "lightweight" because most of the overhead has already been accomplished through the creation of its process

What is a Thread ?

- Because threads within the same process share resources:
  - Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads
  - Two pointers having the same value point to the same data
  - Reading and writing to the same memory locations is possible, thus requiring **explicit synchronization by the programmer**

What are Pthreads ?

- Historically, hardware vendors have implemented their own **proprietary versions of threads**. These implementations differed substantially from each other making it difficult for programmers to develop **portable threaded applications**

- In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required
  - For UNIX systems: IEEE POSIX 1003.1c standard (1995)
  - Compliant implementations named as POSIX threads / Pthreads

- Pthreads are defined as:
  - A set of C language programming types and procedure calls, implemented with a pthread.h header file and a thread library

Why Pthreads ?

- **Light Weight**: Threads vs Processes
  - A thread can be created with much less OS overhead
  - Managing threads requires fewer system resources
- **example**: time to create 20000 processes (with fork) vs time to create 20000 threads (with pthread_create) – see appendix_1

CPU used	processes (wait all)	threads (joinable)
Intel i7-8665U mobile CPU; 4 cores (1.9GHz – 4.8GHz); physical machine	1,3 s	0,43 s
Intel Xeon W-2195 CPU; 18 cores (2.3GHz – 4.3GHz); virtual machine	2,42 s	0,55 s
AMD EPYC 7351 CPU; 16 cores (2.4GHz – 2.9GHz); virtual machine	2,54 s	0,77 s

Note: times measured forcing a single CPU-core to be used

Why Pthreads ?

- **Efficient Communications/Data Exchange**:
  - A parallel application based on message passing (MPI) uses shared-memory for on-node task communication
    - this involves at least one memory copy (process to process)
  - Threads share the same address space within a single process
    - no intermediate copy required; passing pointers usually enough
  - Pthread communications become more a cache-to-CPU or memory-to-CPU bandwidth issue. These speeds are much higher than MPI shared memory communications

Platform	MPI Shared Memory Bandwidth (GB/sec)	Pthreads Worst Case Memory-to-CPU Bandwidth (GB/sec)
Intel 2.6 GHz Xeon E5-2670	4.5	51.2
Intel 2.8 GHz Xeon 5660	5.6	32
AMD 2.4 GHz Opteron	1.2	5.3

Why Pthreads ?

- **Other Common Reasons:**
  - Threaded applications offer **potential performance gains** and other practical advantages over non-threaded applications
  - **Overlapping CPU work with I/O**: a program has sections where performs long I/O operations; while one thread is waiting for an I/O system call to complete, others can do CPU intensive work
  - **Priority/real-time scheduling:** tasks which are more important can be scheduled to supersede or interrupt lower priority tasks
  - **Asynchronous event handling**: tasks which service events of indeterminate frequency and duration can be interleaved
    - example: a web server can both transfer data from previous requests and manage the arrival of new requests

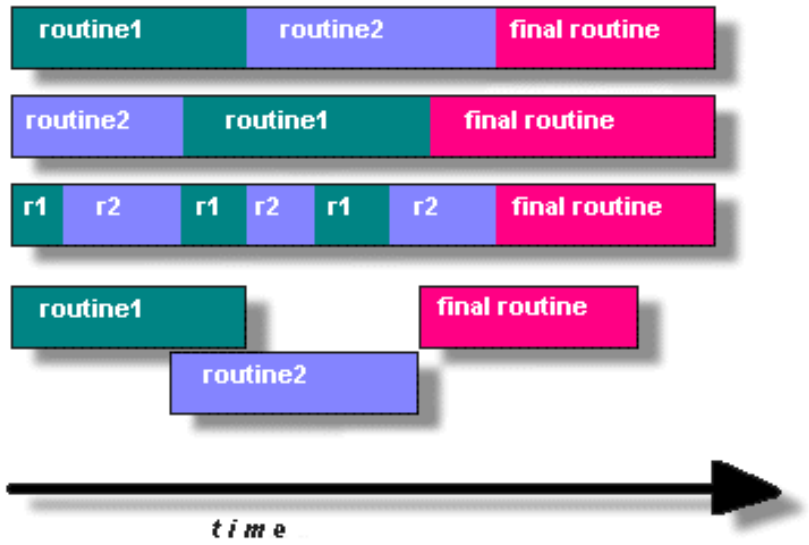
Designing Threaded Programs

- **Parallel Programming**:
  - On modern, multi-core machines, Pthreads are ideally suited for parallel programming, and whatever applies to parallel programming in general, applies to parallel Pthreads programs
  - Many considerations for designing parallel programs, such as:
    - What type of parallel programming model to use?
    - Problem partitioning
    - Load balancing
    - Communications
    - Data dependencies
    - Synchronization and race conditions
    - Memory issues
    - I/O issues
    - Program complexity
    - Programmer effort/costs/time

For a quick overview on parallel computing see:
https://computing.llnl.gov/tutorials/parallel_comp

Designing Threaded Programs

- **Parallel Programming**:
  - In order for a program to take advantage of Pthreads, it must be able to be organized into discrete, independent, concurrent tasks
  - Example: if routine1 and routine2 can interchange, interleave and/or overlap in real time, they are candidates for threading



Designing Threaded Programs

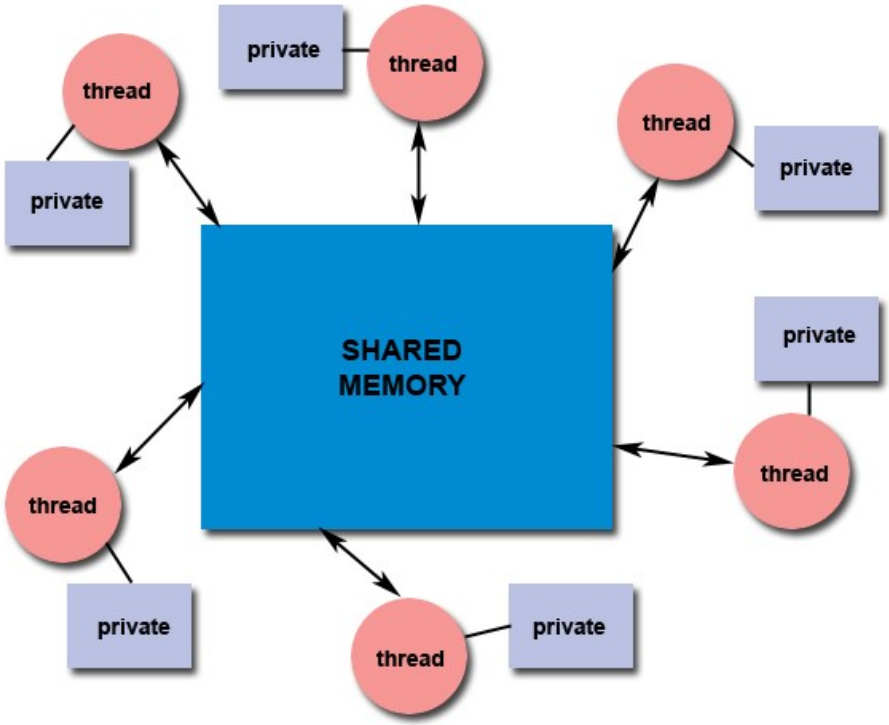
- **Parallel Programming**:
  - Programs with these characteristics may suit well Pthreads:
    - Work that can be executed, or data that can be operated on, by multiple tasks simultaneously
    - Block for potentially long I/O waits
    - Use many CPU cycles in some places but not others
    - Must respond to asynchronous events
    - Some work is more important than other (priority interrupts)

Designing Threaded Programs

- **Parallel Programming**:
  - Several common models for threaded programs exist:
    - **Manager/worker**:
      - one *manager* thread assigns work to other threads (the *workers*)
      - *manager* handles all input and parcels out work to the *workers*
      - common alternatives: static worker pool, dynamic worker pool
    - **Peer**: similar to the manager/worker model, but after the `main` thread creates other threads, it participates in the work
    - **Pipeline**:
      - a task is broken into a series of suboperations; each one is handled in series, but concurrently, by a different thread
      - an automobile assembly line best describes this model

Designing Threaded Programs

- **Shared Memory Model:**
  - All threads have access to the same global, shared memory
  - Threads also have their own private data
  - Programmers are responsible for synchronizing access (protecting) globally shared data

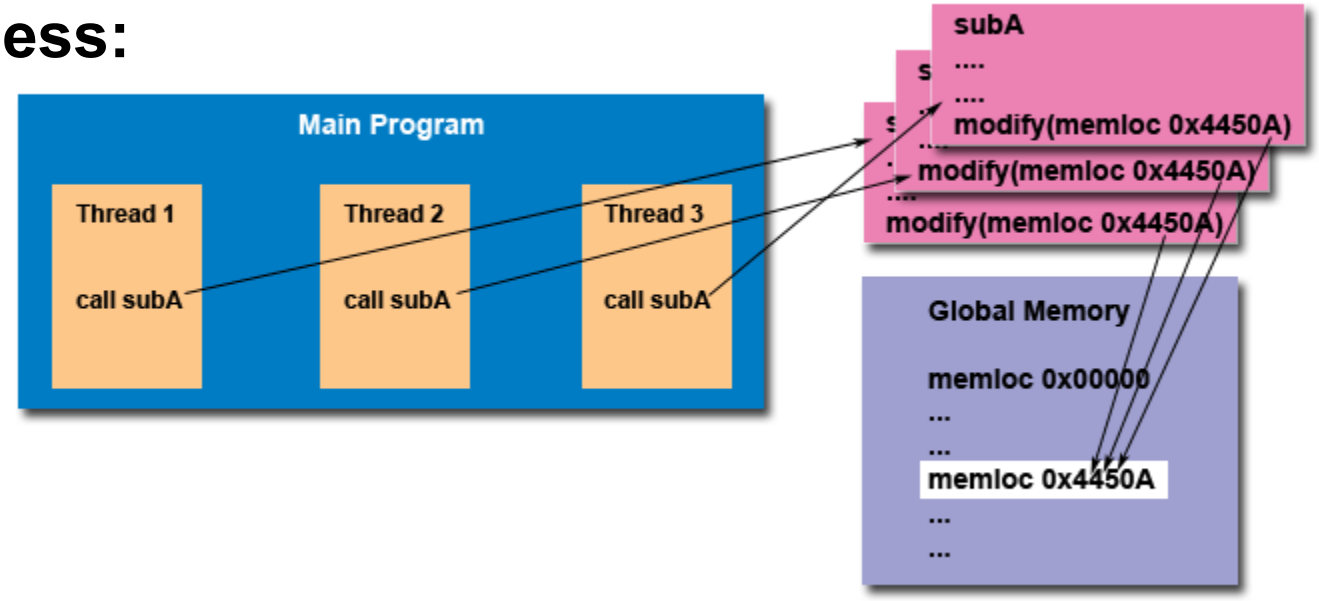


Designing Threaded Programs

- **Thread-safeness:**
  - An application's ability to execute multiple threads simultaneously without "clobbering" shared data or creating "race" conditions
  - example:
    - an application creates several threads, each of which makes a call to the same library routine
    - the library routine accesses/modifies a global structure or location in memory
    - as each thread calls this routine it is possible that they may try to modify this global structure/memory location at the same time
    - if the routine does not employ some sort of synchronization constructs to prevent data corruption, then it is not thread-safe
  - If one is not 100% certain that a library routine is thread-safe, calls to it should be “serialized” (ensure that no more than one thread executes the routine at the same time); otherwise, nasty problems could arise

Designing Threaded Programs

- **Thread-safeness:**



- **Thread Limits:**
  - Implementations of the Pthreads API may vary in non-standard ways
  - A program running fine in a platform, may fail/be erroneous in another
  - Example: the maximum number of threads permitted, and the default thread stack size, should be considered when designing a program

The Pthreads API

The Pthreads API

- Around 100 routines, organized in several groups:

Routine Prefix	Functional Group
pthread_	Threads themselves and miscellaneous subroutines
pthread_attr_	Thread attributes objects
pthread_mutex_	Mutexes
pthread_mutexattr_	Mutex attributes objects.
pthread_cond_	Condition variables
pthread_condattr_	Condition attributes objects
pthread_key_	Thread-specific data keys
pthread_rwlock_	Read/write locks
pthread_barrier_	Synchronization barriers

The Pthreads API

- **Thread management:**
  - Routines that work directly on threads - creating, detaching, joining, etc.
  - They also include functions to set/query thread attributes (joinable, scheduling etc.)

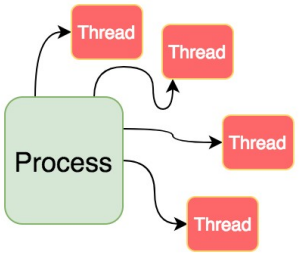
- **Mutexes:**
  - Routines that deal with synchronization
  - "Mutex" = "mutual exclusion"
  - Mutex functions provide for creating, destroying, locking and unlocking mutexes
  - These are supplemented by mutex attribute functions that set or modify attributes associated with mutexes

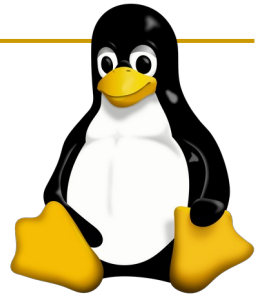
The Pthreads API

- **Condition variables:**
  - Routines that address communications between threads that share a mutex
  - Based upon programmer specified conditions
  - Includes functions to create, destroy, wait and signal based upon specified variable values
  - Includes also functions to set/query condition variable attributes
- **Synchronization:**
  - Routines that manage **read/write locks** and **barriers**
- **Opaque Objects:**
  - The basic calls work to create or modify opaque objects
  - The opaque objects can be modified by calls to attribute functions, which deal with opaque attributes

Pthreads in Linux

for more details: `man 7 pthreads`





- offered by the **Native POSIX Thread Library (NPTL)**, a so-called 1×1 threads library, part of the GNU C Library
  - simple approach: threads created by users (using the function `pthread_create`) correspond (1 to 1) with schedulable entities in the kernel (tasks), created via the `clone` system call
- **man pages:** two different sets (NPTL/GNU and POSIX)
  - **install**: `apt-get install glibc-doc manpages-posix manpages-posix-dev`
  - **usage**: `man [3] pthread_create ; man 3posix pthread_create`
- **GCC compilation:** compile with `gcc … myprog.c -pthread`
  - **warning**: `-lpthread` instead of `-pthread` does not define reentrant functions
- **access to NPTL non-portable (NP) features:**
  - add `#define __USE_GNU` before `#include <pthread.h>`
  - see also `/usr/include/pthread.h`

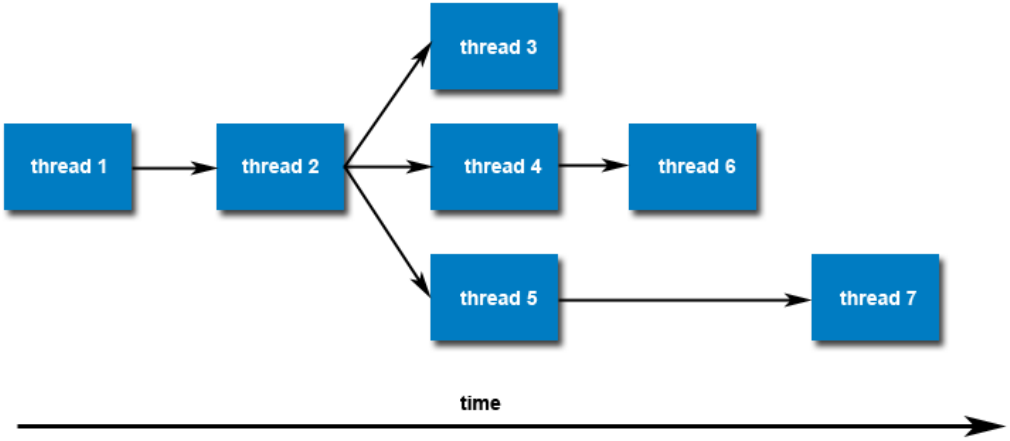
Thread Management

Creating and Terminating Threads

- **Creating Threads:**
  - `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg)`
    - **`thread`**: opaque, unique identifier for the new thread returned by `pthread_create` (in Linux it is of type `unsigned long int`)
    - **`attr`**: opaque attribute object used to set thread attributes
      - `NULL` is used for the default attribute values
    - **`start_routine`**: routine where the thread will start executing
    - **`arg`**: a single argument that may be passed to `start_routine`
      - it must be passed as a pointer cast of type `void`
      - `NULL` is used if no argument is to be passed
    - on success returns `0`; on error returns an error number

Creating and Terminating Threads

- **Creating Threads:**
  - The `main()` function comprises a single, default thread; all other threads must be explicitly created by the programmer
  - **`pthread_create()`** can be called any num. of times, anywhere
  - After a call to **`pthread_create()`**, it is indeterminate which thread – the caller thread or the new thread – will next execute
  - Once created, threads are **peers**, and may create other threads. There is no implied hierarchy or dependency between threads



Creating and Terminating Threads

- **Terminating Threads:**
  - There are several ways in which a thread may be terminated:
    - The thread calls **`pthread_exit()`** whether its work is done or not
    - The thread ends its **`start_routine`** via `return` (its work is done)
      - (*) **`pthread_exit()`** is called implicitly with the parameter of `return`
    - The thread is cancelled by another thread via **`pthread_cancel()`**
    - If `main()` finishes first, without calling **`pthread_exit()`** explicitly
    - The entire process is terminated due to a call to `exit()`
    - The program of the process is replaced by a call to `exec()`
  - When a thread terminates, process-shared resources (mutexes, condition variables, semaphores, file descriptors) aren't released
  - After the last thread in a process terminates, the process terminates as by calling `exit()` with an exit status of zero

Creating and Terminating Threads

- **Terminating Threads: Self-Termination**
  - `void pthread_exit(void *retval)`
    - **`retval`**: pointer to the return status of the thread, available to any other thread in the same process that "joins" to the ending thread
    - **`retval`** may be `NULL`; also, it should not point the calling thread stack, since its contents are undefined after the thread terminates
  - (*) Performing a **`return`** from the start function of any thread other than the **`main`** thread results in an implicit call to **`pthread_exit()`**, using the start function's return value as the thread's exit status
    - no need to invoque **`pthread_exit()`** if executing to completion
  - calling **`pthread_exit()`** from **`main()`**:
    - To allow other threads to continue execution, the **`main()`** thread should terminate via **`pthread_exit()`** and not **`exit()`**; that way, **`main()`** will block to support the threads it created until they are all done (**`main()`** will wait for them even if they are in a *detached* state)

Creating and Terminating Threads

- **Example 0:**

Process with 2 threads. Thread **`main`** creates a new thread using **`pthread_create`**. The new thread starts at the function **`aThread`**. Both threads end with **`pthread_exit`**. The new thread does not receive any parameter from **`main`**.
Q1: what are the possible execution orders of **A**, **B** and **C** ?
Q2: what may happen if **`main`** doesn't call **`pthread_exit`**?

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

void *aThread(void *arg)
{
   printf("Process %d: new thread: Hello World !\n", getpid()); // A
   pthread_exit(NULL);
}

int main()
{
   pthread_t thread_opaque_id;
   int ret;

   printf("Process %d: main thread: creating a new thread\n", getpid()); // B
   ret = pthread_create(&thread_opaque_id, NULL, aThread, NULL);
   if (ret){
       printf("ERROR; return code from pthread_create() is %d\n", ret);
       exit(ret);
   }

   printf("Process %d: main thread: program completed: exiting\n", getpid()); // C
   pthread_exit(NULL);     /* Last thing that main() should do */
}
```

Error-checking code. Omitted onwards for the sake of simplicity.

Creating and Terminating Threads

- **Example 1:**

Creates 5 threads with **`pthread_create`**. Each thread prints the PID of its hosting process and a pseudo ID received as a parameter, and then terminates with **`pthread_exit`**.

The parameter "**`(void*)t`**" in **`pthread_create`** is passed (copied) by value to the new thread stack and so there is no danger in "**`t`**" being modified by the main thread in the for loop (**`t++`** only happens after **`t`** is copied to the new thread stack).

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

void *aThread(void *arg)
{
   long tid=(long)arg;

   printf("Process %d: thread %ld: Hello World !\n", getpid(), tid);
   pthread_exit(NULL);
}

int main()
{
   pthread_t threads[NUM_THREADS];
   long t;

   for(t=0;t<NUM_THREADS;t++){
     printf("Process %d: thread main: creating thread %ld\n", getpid(), t);
     pthread_create(&threads[t], NULL, aThread, (void *)t);
   }

   printf("Process %d: thread main: program completed: exiting\n",getpid());
   pthread_exit(NULL);
}
```

Creating and Terminating Threads

- **Example 1A:**

Similar to Example 1, but shows that threads share access to **global variables** (in this situation for read-only access).

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

char *messages[NUM_THREADS]={"English: Hello World!","French: Bonjour, le monde!","Spanish: Hola al mundo",
                             "German: Guten Tag, Welt!","Russian: Zdravstvytye, mir!"};

void *aThread(void *arg)
{
   long tid=(long)arg;

   printf("Process %d: thread %ld: %s\n", getpid(), tid, messages[tid]);
   pthread_exit(NULL);
}

int main()
{
   pthread_t threads[NUM_THREADS];
   long t;

   for(t=0;t<NUM_THREADS;t++){
     printf("Process %d: thread main: creating thread %ld\n", getpid(), t);
     pthread_create(&threads[t], NULL, aThread, (void *)t);
   }

   printf("Process %d: thread main: program completed: exiting\n",getpid());
   pthread_exit(NULL);
}
```

Passing Arguments to Threads

- **`pthread_create`** passes one argument to the start routine
- To pass multiple parameters, one may use a vector (if they are of the same type) or enclose them in a structure (if they are of different types) and pass a pointer to that vector / structure cast to `void*`
- But thread parameter(s) must be **thread-safe**:
  - passing by value is safe; passing by reference may be not **(*)**
  - **(*)** parameters should not be changeable by any other thread
  - **(*)** parameters should not be originally created in the stack of the calling thread if this thread may end before the new one
  - **solutions**: place parameters in dynamic memory (process heap kept if calling thread exits); place parameters in global variables

Passing Arguments to Threads

- **Example 1B:**

Similar to Example 1, but susceptible to a Race Condition (*): the parameter (**`&t`**) to the new threads points to data (**`t`**) being changed in the **`main`** thread while being read in the new ones.

(**) Moreover, by the time the new threads de-reference the **`arg`** pointer, the **`main`** thread may already have reached **`pthread_exit`**, meaning all **`main`** local variables are gone (including **`t`**) and so it would be invalid to try to access them.

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

void *aThread(void *arg)
{
   long tid=*(long*)arg;

   printf("Process %d: thread %ld: Hello World !\n", getpid(), tid);
   pthread_exit(NULL);
}

int main()
{
   pthread_t threads[NUM_THREADS];
   long t;

   for(t=0;t<NUM_THREADS;t++){
     printf("Process %d: thread main: creating thread %ld\n", getpid(), t);
     pthread_create(&threads[t], NULL, aThread, (void *)&t);
   }

   printf("Process %d: thread main: program completed: exiting\n",getpid());
   pthread_exit(NULL);
}
```

(*) Race Condition !

(**) Invalid Access !

Passing Arguments to Threads

- **Example 1B** (output in red and blue reveals Race-Condition):

```
#thread 0 (*) thinks that it is thread 1 (and this side-effect propagates to the other threads)
Process 1027136: thread main: creating thread 0
Process 1027136: thread main: creating thread 1
Process 1027136: thread 1: Hello World !
Process 1027136: thread main: creating thread 2
Process 1027136: thread 2: Hello World !
Process 1027136: thread main: creating thread 3
Process 1027136: thread 3: Hello World !
Process 1027136: thread main: creating thread 4
Process 1027136: thread main: program completed: exiting
Process 1027136: thread 4: Hello World !
Process 1027136: thread 5: Hello World !

#threads 0 (*) and 1 (*) think they are thread 2 and 2 (and this side-effect propagates to the other threads)
Process 1027174: thread main: creating thread 0
Process 1027174: thread main: creating thread 1
Process 1027174: thread main: creating thread 2
Process 1027174: thread 2: Hello World !
Process 1027174: thread 2: Hello World !
Process 1027174: thread main: creating thread 3
Process 1027174: thread 3: Hello World !
Process 1027174: thread main: creating thread 4
Process 1027136: thread main: program completed: exiting
Process 1027174: thread 4: Hello World !
Process 1027174: thread 5: Hello World !

(*) assuming they are scheduled by the creation order, which is not even ensured ...
```

Passing Arguments to Threads

- **Example 1C:**

Similar to Example 1B but solves both problems, once the thread parameters are **dynamically allocated**. This still suffers from a **memory leak**, once the dynamic array **`tids`** is not freed.

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

void *aThread(void *arg)
{
   long tid=*(long*)arg;

   printf("Process %d: thread %ld: Hello World !\n", getpid(), tid);
   pthread_exit(NULL);
}

int main()
{
   pthread_t threads[NUM_THREADS]; long *tids;
   long t;

   tids=(long*)malloc(NUM_THREADS*sizeof(long));
   for(t=0;t<NUM_THREADS;t++){
     tids[t]=t;
     printf("Process %d: thread main: creating thread %ld\n", getpid(), t);
     pthread_create(&threads[t], NULL, aThread, (void *)&tids[t]);
   }

   printf("Process %d: thread main: program completed: exiting\n",getpid());
   pthread_exit(NULL);
}
```

Passing Arguments to Threads

- **Example 2:**

Passes several parameters of different types to each thread, packed in a specific structure for each thread. Parameters are placed in dynamic memory and there is still a **memory leak** (no thread frees **`params`**).

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NUM_THREADS 5
typedef struct { long tid; char character; } longchar_t;

void *aThread(void *arg)
{
   longchar_t *args=(longchar_t*)arg;

   printf("thread %ld: character %c: Hello World !\n", args->tid, args->character);
   pthread_exit(NULL);
}

int main()
{
   pthread_t threads[NUM_THREADS]; longchar_t *params;
   long t;

   params=(longchar_t *)malloc(NUM_THREADS*sizeof(longchar_t));
   for(t=0;t<NUM_THREADS;t++){
     params[t].tid=t; params[t].character='a'+(int)t;
     printf("thread main: creating thread %ld\n", t);
     pthread_create(&threads[t], NULL, aThread, (void *)&params[t]);
   }

   printf("thread main: program completed: exiting\n");
   pthread_exit(NULL);    /* Last thing that main() should do */
}
```

Passing Arguments to Threads

- **Example 2A:**

Similar to Example 2 but passes some parameters via a global array. Each thread still needs to receive and index (**t**) into the global array. There are no memory leaks once dynamic memory is never allocated.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NUM_THREADS 5
typedef struct { float num; char character; } floatchar_t;
floatchar_t params[NUM_THREADS];

void *aThread(void *arg)
{
   long tid=(long)arg;

   printf("thread %ld: num %f: character %c: Hello World !\n", tid, params[tid].num, \
                                                               params[tid].character);
   pthread_exit(NULL);
}

int main()
{
   pthread_t threads[NUM_THREADS];
   long t;

   for(t=0;t<NUM_THREADS;t++){
     params[t].num=(float)t; params[t].character='a'+(int)t;
     printf("thread main: creating thread %ld\n", t);
     pthread_create(&threads[t], NULL, aThread, (void *)t);
   }

   printf("thread main: program completed: exiting\n");
   pthread_exit(NULL);    /* Last thing that main() should do */
}
```

Identity Enquiry and Comparison

- **Identity Enquiry:**
  - `pthread_t` **`pthread_self`**`(void)`
    - returns the unique opaque ID of the calling thread (this is the same value returned to `*thread` by `pthread_create()` in its creation)
    - thread IDs are guaranteed to be unique only within a process
    - a thread ID may be reused after a terminated thread has been *joined*, or a *detached* thread has terminated
    - in Linux, a thread ID is represented by a `unsigned long int` (see `/usr/include/x86_64-linux-gnu/bits/pthreadtypes.h`)

- **Identity Comparison:**
  - `int` **`pthread_equal`**`(pthread_t` **`t1`**`, pthread_t` **`t2`**`)`
    - compares two opaque thread IDs (== should not be used)
    - returns a non-zero value if the IDs are equal, and 0 otherwise

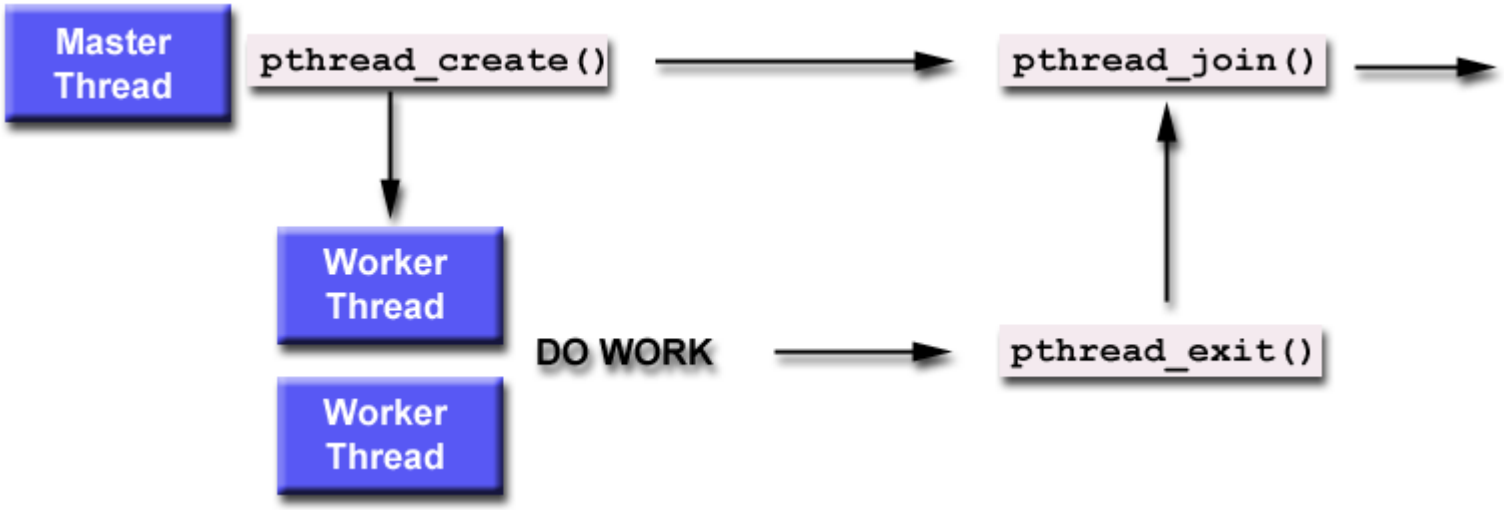
Identity Enquiry and Comparison

- **Example 3:**

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

pthread_t thread;

void *aThread(void *arg) {
 printf("another thread: my thread opaque ID: %lu\n", pthread_self());
 if (pthread_equal(pthread_self(), thread))
     printf("SUCCESS: threads are the same, which is correct.\n");
 else
     printf("ERROR: threads are not the same, which is incorrect.\n");
 pthread_exit(NULL);
}

int main()
{
 pthread_create(&thread, NULL, aThread, NULL);
 printf("main thread: my thread opaque ID: %lu\n", pthread_self());
 if (pthread_equal(pthread_self(), thread))
     printf("ERROR: threads are the same, which is incorrect.\n");
 else
     printf("SUCCESS: threads are not the same, which is correct.\n");
 pthread_exit(NULL);
}
```

Joining and Detaching Threads

- **Joining:**
  - One way to accomplish **synchronization between threads**
    - other methods: *mutexes* and *condition variables*
  - Any thread can join with any other thread in the process
    - a creator thread may join with a thread created by it, and vice-versa (in the figure bellow, only the first alternative is represented)



Joining and Detaching Threads

- **Joining:**
  - `int` **`pthread_join`**`(pthread_t` **`thread`**`, void` **`**retval`**`)`
    - blocks the calling thread until the specified **`thread`** terminates
      - if the specified **`thread`** already terminated, returns immediatelly
      - the specified **`thread`** must be *joinable* (i.e., not *detached*)
    - if **`retval`** is not `NULL`, copies to it the exit status of the **`thread`**
      - the exit status of the target **`thread`** may be a value or a real pointer; that status is copied to the location pointed by **`retval`**
      - if **`thread`** was cancelled, the status is `PTHREAD_CANCELED`
    - on success returns `0`; on error returns an error number
  - A terminating thread can only be joined by **one** other thread, and **once**
    - multiple joins on the same terminating thread are not allowed, either by different threads, or by the same thread in different moments
  - If the calling thread is *cancelled*, the target thread will remain *joinable*

Joining and Detaching Threads

- **Joinable or Not ?**
  - When a thread is created, one of its **attributes** defines whether it is ***joinable*** or ***detached***. Only threads that are created as joinable can be joined. If a thread is created as detached, it can never be joined
  - The standard specifies that threads should be created as joinable
  - To explicitly create a thread as ***joinable*** or ***detached***, the argument **`attr`** in the **`pthread_create()`** routine is used, following 4-steps:
    - Declare a pthread attribute variable of the `pthread_attr_t` type
    - Initialize the attribute variable with `pthread_attr_init()`
    - Set the detached status with `pthread_attr_setdetachstate()`
    - Free resources used by the attribute with `pthread_attr_destroy()`

Joining and Detaching Threads

- **Detaching:**
  - `int pthread_detach(pthread_t thread);`
    - Marks **`thread`** as *detached* (even if created as *joinable*)
    - When a *detached* thread terminates, its resources are automatically released without the need for another thread to join with it
      - **caveat**: by calling `pthread_exit` the `main` waits for all other threads
    - A *detached* thread cannot be put back on the *joinable* state
    - A thread can detach itself: `pthread_detach(pthread_self())`

- **Recommendations:**
  - If a thread requires joining, consider explicitly creating it as *joinable*
    - Not all implementations may create threads as *joinable* by default
  - If its known in advance that a thread will never need to join with another thread, consider creating it in a *detached* state (frees some resources)

Joining and Detaching Threads

- **Example 4:**

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

void *aThread(void *arg)
{
 long tid = (long)arg;

 printf("thread %ld: Hello World !\n", tid);
 pthread_exit(NULL);
}
```

```
int main()
{
 pthread_t threads[NUM_THREADS];
 long t; int ret;

 for(t=0;t<NUM_THREADS;t++){
     printf("thread main: creating thread %ld\n", t);
     pthread_create(&threads[t], NULL, aThread, (void *)t);
 }

 /* Wait for the other threads */
 for(t=0; t<NUM_THREADS; t++) {
     ret = pthread_join(threads[t], NULL);
     if (ret) {
        printf("ERROR; return code from pthread_join() is %d\n", ret);
        exit(ret);
     }
     printf("thread main: joined with thread %ld\n", t);
 }

 printf("thread main: program completed: exiting\n"); // (*)
 pthread_exit(NULL);
}
```

Error-checking code. Omitted onwards for the sake of simplicity.

Shows how to wait for thread completion using **`pthread_join`** .
Q: is a specific joining order needed ?

The created threads do not return meaningful state (exit status is NULL).

The sentence **(*)** is always printed last.
Q: what if the last loop is commented ?

Joining and Detaching Threads

- **Example 4A:**

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

void *aThread(void *arg)
{
 long tid = (long)arg;

 printf("thread %ld: Hello World !\n", tid);
 pthread_exit(NULL);
}
```

```
int main()
{
 pthread_t threads[NUM_THREADS];
 pthread_attr_t attr;
 long t;

 /* Initialize and set thread detached attribute */
 pthread_attr_init(&attr);
 pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

 for(t=0;t<NUM_THREADS;t++){
     printf("thread main: creating thread %ld\n", t);
     pthread_create(&threads[t], &attr, aThread, (void *)t);
 }

 /* Free attribute */
 pthread_attr_destroy(&attr);

 for(t=0; t<NUM_THREADS; t++) {
     pthread_join(threads[t], NULL);
     printf("thread main: joined with thread %ld\n", t);
 }

 printf("thread main: program completed: exiting\n");
 pthread_exit(NULL);
}
```

Like Example 4 but shows how to explicitly create *joinable* threads (explicitly setting threads as *joinable* is not usually necessary; typically they are created *joinable* by default).

Joining and Detaching Threads

- **Example 4B:**

Warning: using this technique only works if the size of the value to return is less or equal to the size of `void*`; and even so the compiler may not allow casting to `void*`; to see this problem happening use code **B** instead of **A**.

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

void *aThread(void *arg)
{
 long tid = (long)arg;
 long square = tid*tid;       // A
 //double square = tid*tid;   // B

 printf("thread %ld: Hello World !\n", tid);
 pthread_exit((void*)square);
}
```

```
int main()
{
 pthread_t threads[NUM_THREADS];
 long t; void *status;

 for(t=0;t<NUM_THREADS;t++){
     printf("thread main: creating thread %ld\n", t);
     pthread_create(&threads[t], NULL, aThread, (void *)t);
 }

 for(t=0; t<NUM_THREADS; t++) {
     pthread_join(threads[t], &status);
     printf("thread main: joined with thread %ld: \
             exit status is: %ld\n", \
             t, (long)status); // A
     //printf("thread main: joined with thread %ld: \
               //exit status is: %f\n", \
               //t, (double)status); // B
 }

 printf("thread main: program completed: exiting\n");
 pthread_exit(NULL);
}
```

Similar to Example 4, but each worker returns a long value as exit status. That value is copied from the stack of a worker thread to the stack of the main thread (to its **`status`** variable).

Joining and Detaching Threads

- **Example 4C:**

Note: this approach allows a thread to return any data type, once what is really being returned is a pointer to that type; to confirm that this solution works, use code **B** instead of **A**.

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

void *aThread(void *arg)
{
 long tid = (long)arg;
 long *square;     // A
 //double *square; // B

 square=(long*)malloc(sizeof(long));   // A
 //square=(double*)malloc(sizeof(double)); // B
 *square = tid*tid;
 printf("thread %ld: Hello World !\n", tid);
 pthread_exit((void*)square);
}
```

```
int main()
{
 pthread_t threads[NUM_THREADS];
 long t; void *status;

 for(t=0;t<NUM_THREADS;t++){
     printf("thread main: creating thread %ld\n", t);
     pthread_create(&threads[t], NULL, aThread, (void *)t);
 }

 for(t=0; t<NUM_THREADS; t++) {
     pthread_join(threads[t], &status);
     printf("thread main: joined with thread %ld: \
             exit status is: %ld\n", \
             t, *(long*)status);   // A
     //printf("thread main: joined with thread %ld: \
               //exit status is: %f\n", \
               //t, *(double*)status); // B
     free(status);
 }

 printf("thread main: program completed: exiting\n");
 pthread_exit(NULL);
}
```

Like Example 4B, but each worker reserves dynamic memory for a value and returns the pointer to it. The main thread uses the pointer to access the value and frees the dynamic memory (and so there are no memory leaks).

Joining and Detaching Threads

- **Example 4D:**

Note: this approach is also compatible with any data type; however, global variables should be used with caution: besides prone to race-conditions, intensive access to these variables will harm performance (local variables or dynamic memory are preferable, performance-wise).

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

long squares[NUM_THREADS];     // A
//double squares[NUM_THREADS]; // B

void *aThread(void *arg)
{
 long tid = (long)arg;

 squares[tid] = tid*tid
 printf("thread %ld: Hello World !\n", tid);
 pthread_exit(NULL);
}
```

```
int main()
{
 pthread_t threads[NUM_THREADS];
 long t;

 for(t=0;t<NUM_THREADS;t++){
     printf("thread main: creating thread %ld\n", t);
     pthread_create(&threads[t], NULL, aThread, (void *)t);
 }

 for(t=0; t<NUM_THREADS; t++) {
     pthread_join(threads[t], NULL);
     printf("thread main: joined with thread %ld: \
             thread square is: \
             %ld\n", t, squares[t]); // A
     //printf("thread main: joined with thread %ld: \
             //thread square is: \
             //%f\n", t, squares[t]);  // B
 }

 printf("thread main: program completed: exiting\n");
 pthread_exit(NULL);
}
```

Variant of Example 4C where values from workers are passed via a global shared array; note that each thread writes in a separate cell of the array (and so there isn't any Race Condition)

Joining and Detaching Threads

- **Example 5:**

Shows that it is possible to join with the main thread, unless (*) that thread gets detached before someone tries to join with it. (note: error codes are documented in the pthread_join manual)

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <errno.h>
#include <stdint.h> // avoid -Wpointer-to-int-cast

pthread_t thread_main;

void *aThread(void *arg)
{
   long tid=(long)arg;
   int ret; void *status;

   for (int i=0; i<5*tid; i++) {
       printf("thread %ld: i=%d\n", tid, i);
       sleep(1);
   }

   printf("thread %ld: going to join with thread main\n", tid);
   ret = pthread_join(thread_main, &status);

   if (ret==0) printf("thread %ld: SUCCESS; thread main returned %d\n", tid, (int)(uintptr_t)status);
   else if (ret==EINVAL) printf("thread %ld: ERROR; thread main is not a joinable thread or \
                                another thread is already waiting to join with main\n", tid);
   else if (ret==ESRCH) printf("thread %ld: ERROR; thread main does not exist\n", tid);
   else printf("thread %ld: ERROR; thread main could not be joined due to reason %d\n", tid, ret);

   printf("thread %ld: going to die\n", tid);
   pthread_exit(NULL);
}
```

```
int main()
{
   pthread_t thread1, thread2;

   thread_main=pthread_self();

   //pthread_detach(thread_main); // (*)

   pthread_create(&thread1, NULL, aThread, (void *)1);
   pthread_create(&thread2, NULL, aThread, (void *)2);

   printf("thread main: going to die\n");
   pthread_exit((void*)3);
}
```

Terminating Threads - Cancellation

- **Cancelling Other Threads**
  - `int` **`pthread_cancel`**`(pthread_t *`**`thread`**`)`
    - sends a cancellation request to the thread **`thread`**
    - if and when **`thread`** reacts to the request depends on its cancelability *state* (enabled/disabled) and *type* (deferred/asynchronous)
      - *type* is "deferred" by default: cancellation is delayed until the thread calls a function that is a cancellation point (see **`man 7 pthreads`** for a func. list)
    - during cancellation, previously defined clean-up handlers and data destructors are executed, and the thread terminates with status **(*)**
      - see the man page for details
    - on success returns `0`; on error returns an error number
      - this value merely informs whether the request was successfully queued
    - after a cancelled thread terminates, a join with that thread using **`pthread_join`** obtains `PTHREAD_CANCELED` **(*)** as exit status
      - joining with a thread is the only way to know that cancellation has completed

Terminating Threads - Cancellation

- **Example 6A:**

Shows how to cancel a thread before it would self terminate (counting on the thread created stops at 2, never reaches 5).

Compare with running the program with lines (*) commented.

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

void *aThread(void *arg)
{
   for (int i=0; i<5; i++) { printf("thread created: %d\n", i); sleep(1); }
   printf("thread created: going to self terminate\n");
   pthread_exit(NULL);
}

int main()
{
   pthread_t thread; int ret; void *status;

   ret = pthread_create(&thread, NULL, aThread, (void *)NULL);
   if (ret){ printf("ERROR; return code from pthread_create() is %d\n", ret); exit(ret); }

   for (int i=0; i<2; i++) { printf("thread main: %d\n", i); sleep(1); }

   ret = pthread_cancel(thread); // (*)
   if (ret){ printf("ERROR; return code from pthread_cancel() is %d\n", ret); exit(ret); } // (*)

   ret = pthread_join(thread, &status);
   if (ret){ printf("ERROR; return code from pthread_join() is %d\n", ret); exit(ret); }
   if (status == PTHREAD_CANCELED) printf("thread main: thread created was cancelled\n");
   else printf("thread main: thread created was not cancelled\n");

   pthread_exit(NULL);
}
```

Terminating Threads - Cancellation

- **Check If Own Cancelling Is Due**
  - `int pthread_testcancel(void)`
    - request delivery of any pending cancellation request
    - creates a cancellation point within the calling thread, so that a thread that is otherwise executing code that contains no cancellation points (or they've yet to be reached) will respond to a cancellation request
    - if cancellability is disabled (*), or no cancellation request is pending, then a call to `pthread_testcancel` has no effect
      - (*) using `pthread_setcancelstate(3)`
    - this function always succeeds and does not return a value
      - if the calling thread is cancelled as a consequence of a call to this function, then the function does not return

Terminating Threads - Cancellation

- **Example 6B:**

Shows how to use pthread_testcancel to introduce explicit opportunities for a pending cancellation. Execute several times to see how far the global counter **i** goes each time.

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

unsigned long long i=0;
```

```
void *aThread(void *arg)
{
   while (++i) pthread_testcancel();
   printf("aThread: ERROR; should never reach this printf\n");
   pthread_exit(NULL);
}

int main()
{
   pthread_t thread; int ret; void *status;

   ret = pthread_create(&thread, NULL, aThread, (void *)NULL);
   if (ret){ printf("ERROR; return code from pthread_create() is %d\n", ret); exit(ret); }

   ret = pthread_cancel(thread);
   if (ret){ printf("ERROR; return code from pthread_cancel() is %d\n", ret); exit(ret); }

   ret = pthread_join(thread, &status);
   if (ret){ printf("ERROR; return code from pthread_join() is %d\n", ret); exit(ret); }
   if (status == PTHREAD_CANCELED) printf("thread main: thread created was cancelled\n");
   else printf("thread main: thread created was not cancelled\n");

   printf("thread main: i=%llu\n",i);

   pthread_exit(NULL);
}
```

Mutex Variables

Overview

- **Mutex** is an abbreviation for "**mutual exclusion**".
- Mutex variables allow for **thread synchronization** and for **protecting shared data** when multiple writes occur
- A mutex variable acts like a "lock"
  - Only one thread can lock (and thus own) a mutex at any time
  - If several threads try to lock a mutex only one will be successful
  - No other thread can own a mutex until the owning thread unlocks that mutex. Threads must "take turns" accessing protected data
  - When more than one thread is waiting for a locked mutex, the thread that will be granted the lock first, after it is released, is decided by the native system scheduler; applications shouldn't make any particular assumptions other than that decision will appear to be simply random

Overview

- Mutexes prevent "race" conditions, ensuring exclusive access to **critical sections** of code where **shared data may be changed**

- Example of a race condition involving a bank transaction:

Thread 1	Thread 2	Balance
Read balance: $1000		$1000
	Read balance: $1000	$1000
	Deposit $200	$1000
Deposit $200		$1000
Update balance $1000+$200		$1200
	Update balance $1000+$200	$1200

  - solution: a mutex should be used to lock the "Balance" while any thread is using this shared data resource

Overview

- Typical sequence in the use of a mutex:
  - **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 (**critical section**)
  - The owner **unlocks** the mutex
  - Another thread **acquires** the mutex and repeats the process
  - Finally the mutex is destroyed
- Except otherwise stated, all the following functions return 0 on success, and another value (error code) on insuccess
- When several threads compete for a mutex, the loosers block at that call (**exception**: an unblocking call is available – "trylock")
- When protecting shared data, it is the programmer's responsibility to make sure every thread that needs to use a mutex does so

Creating and Destroying Mutexes

- Mutex variables are declared with type **`pthread_mutex_t`**

- **Initializing Mutexes:**
  - Mutexes must be initialized before they can be used
  - After initialization, the mutex is initially *unlocked*
    - unlike traditional semaphores, that may be initialized as *locked*
  - Two ways to initialize a mutex variable:
    - **Statically**, when declared, assuming **default** attributes:
      `pthread_mutex_t mymutex=PTHREAD_MUTEX_INITIALIZER;`
    - **Dynamically**, with the `pthread_mutex_init()` routine
      - This method permits setting mutex object attributes

Creating and Destroying Mutexes

- **Dynamic Initializing Mutexes:**
  - `int` **`pthread_mutex_init`**`(pthread_mutex_t *`**`mutex`**`, const pthread_mutextattr_t *`**`attr`**`)`
    - initializes **`mutex`** in an unlocked state, with the attributes in **`attr`**
    - **default** attribute values are assumed if **`attr`** is `NULL`
      - or if **`attr`** is initialized with **`pthread_mutexattr_init`**
    - Linux: always returns 0; POSIX: returns 0 / ≠0 on success/insuccess
- **Destroying Mutexes:**
  - `int` **`pthread_mutex_destroy`**`(pthread_mutex_t *`**`mutex`**`)`
    - destroys the mutex object **`mutex`** (frees up its resources)
      - the object may be later re-initialized with `pthread_mutex_init`
    - the mutex must be *unlocked* on entrance or will return `EBUSY`
    - Linux: it only checks if the mutex is unlocked

Managing Mutex Attribute Objects

- **POSIX Mutex *Types*:**
  - **`PTHREAD_MUTEX_NORMAL`**: For single locking; no deadlock detection: a thread trying to relock the mutex without first unlocking it will deadlock
    (*1) Attempting to unlock a mutex locked by a different thread results in undefined behavior. (*2) Attempting to unlock an unlocked mutex results in undefined behavior
  - **`PTHREAD_MUTEX_ERRORCHECK`**: Provides error checking: a thread trying to relock the mutex without first unlocking it returns with an error
    (*3) Attempting to unlock a mutex locked by a different thread returns with an error.
    (*4) Attempting to unlock an unlocked mutex returns with an error.
  - **`PTHREAD_MUTEX_RECURSIVE`**: Provides recursive locking: a thread trying to relock the mutex without first unlocking it will succeed; multiple locks of this mutex requires the same number of unlocks to release the mutex before another thread can acquire the mutex (*3) (*4)
  - **`PTHREAD_MUTEX_DEFAULT`**: An implementation maps this type to one of the types above, most probably to the `PTHREAD_MUTEX_NORMAL` type, but one should check the documentation of the implementation

Locking and Unlocking Mutexes

- **Acquire a lock (locking):**
  - `int` **`pthread_mutex_lock`**`(pthread_mutex_t *`**`mutex`**`)`
    - the calling thread *locks* (acquires) the given pre-initialized **`mutex`**
    - if the **`mutex`** is currently *unlocked*, it becomes *locked* and *owned* by the calling thread, and `pthread_mutex_lock` returns 0 immediately
    - if the **`mutex`** is already *locked* by another thread, the calling thread will *block* until the **`mutex`** becomes available to the calling thread
    - if the **`mutex`** is already *locked* by the calling thread, the outcome of relocking will depend on the **type** of the **`mutex`**:
      - **`NORMAL`** type: the calling thread blocks in a *deadlock*
      - **`ERRORCHECK`**: the calling thread will not block and locking fails (`pthread_mutex_lock` returns immediatelly with `EDEADLK`)
      - **`RECURSIVE`**: the calling thread won't block and locking succeeds (`pthread_mutex_lock` returns immediatelly with 0)

Locking and Unlocking Mutexes

- **Release a lock (unlocking):**
  - `int` **`pthread_mutex_unlock`**`(pthread_mutex_t *`**`mutex`**`)`
    - the calling thread *unlocks* (releases) the given **`mutex`** if it *owns* it
    - if the **`mutex`** is currently *locked* and *owned* by the calling thread, it becomes *unlocked* and `pthread_mutex_unlock` returns 0 immediately
      - **exception:** each time the owning thread unlocks a **`RECURSIVE`** mutex, the lock count decrements by one; only when it reaches zero will the mutex become available for other threads to acquire
    - if the **`mutex`** is already *unlocked* or *owned* by another thread, the outcome of unlocking will depend on the **type** of the **`mutex`**:
      - **`NORMAL`**, **`ERRORCHECK`**, **`RECURSIVE`**: returns with an error if calling thread not owner; undefined behavior if already unlocked
      - **`FAST-NP`** (NPTL): allows unlocking by a thread not owning the mutex
        - **caveat**: this is non-portable behavior and must not be relied upon !!

Locking and Unlocking Mutexes

- **Summing up `NORMAL` type usage**
  - **locking**
    - if already locked by me, I will block forever (deadlock)
    - if already locked by another, I will block until that another unlocks
    - if currently unlocked, I will grab the lock
  - **unlocking**
    - if currently locked by me, I will release the lock
    - if currently locked by another, result is undefined
    - if already unlocked, result is undefined
- There is nothing "magical" about mutexes… they are akin to a "gentlemen's agreement" between participating threads. It is up to the code writer to insure that the necessary threads all make the mutex lock and unlock calls correctly.
  - example of a logical error: (Thread 3 does not honour the agreement to protect the critical section A=A*B)

```
Thread 1      Thread 2      Thread 3
Lock          Lock
A = 2         A = A+1       A = A*B
Unlock        Unlock
```

Locking and Unlocking Mutexes

- **Try to acquire a lock without blocking:**
  - `int pthread_mutex_trylock(pthread_mutex_t *mutex)`
    - equivalent to `pthread_mutex_lock` except that if the referenced **`mutex`** is already locked (by any thread, including the calling thread), `pthread_mutex_trylock` will return immediately with `EBUSY`
    - **exception**: if the mutex **type** is **`RECURSIVE`** and the mutex is currently *owned* by the calling thread, the mutex lock count will increment by one and `pthread_mutex_trylock` returns 0
- **Acquire a lock within a timeout:**
  - `int pthread_mutex_timedlock(pthread_mutex_t *mutex, const struct timespec *abstime)`
    - like `pthread_mutex_lock` but if **`mutex`** is already locked by another thread, it will wait until **`abstime`** for the lock to be unlocked
    - returns: `ETIMEDOUT` on timeout, 0 if locked within timeout

Locking and Unlocking Mutexes

- **Example 7A:**

Shows how to use a mutex to protect the update of a global **counter** by several threads. Test without/with locking. Compare execution time. Note: make sure you use -O0 to compile; otherwise the code may run so fast that GLOBAL_counter is not accessed simultaneously.

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

#define NUM_THREADS 4
#define NUM_INCREMENTS 10000000
int GLOBAL_counter=0;

pthread_mutex_t GLOBAL_counter_mutex = PTHREAD_MUTEX_INITIALIZER;

void *aThread (void *arg)
{
 for (int i=0; i<NUM_INCREMENTS; i++) {
   pthread_mutex_lock (&GLOBAL_counter_mutex);
   GLOBAL_counter ++;
   pthread_mutex_unlock (&GLOBAL_counter_mutex);
 }

 pthread_exit(NULL);
}
```

```
int main()
{
 long t;
 pthread_t threads[NUM_THREADS];

 /* Create worker threads to perform the increments */
 for(t=0;t<NUM_THREADS;t++)
   pthread_create(&threads[t], NULL, aThread, NULL);

 /* Wait for the worker threads to end */
 for(t=0;t<NUM_THREADS;t++)
   pthread_join(threads[t], NULL);

 /* Show value of the counter */
 printf("GLOBAL_counter is %d: should be %d\n",
        GLOBAL_counter, NUM_THREADS*NUM_INCREMENTS);

 pthread_mutex_destroy(&GLOBAL_counter_mutex);
 pthread_exit(NULL);
}
```

Locking and Unlocking Mutexes

- **Example 7B:**

Similar to Example 7A but shows the failed lock attempts.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <errno.h> // EBUSY

#define NUM_THREADS 4
#define NUM_INCREMENTS 10000000
int GLOBAL_counter=0;

pthread_mutex_t GLOBAL_counter_mutex = \
            PTHREAD_MUTEX_INITIALIZER;

void *aThread (void *arg)
{
 unsigned long failed_lock_attempts=0;

 for (int i=0; i<NUM_INCREMENTS; i++) {
   while (pthread_mutex_trylock \
            (&GLOBAL_counter_mutex) == EBUSY)
       failed_lock_attempts ++;
   GLOBAL_counter ++;
   pthread_mutex_unlock (&GLOBAL_counter_mutex);
 }

 pthread_exit((void*)failed_lock_attempts);
}
```

```
int main()
{
 long t; void *status;
 pthread_t threads[NUM_THREADS];

 /* Create worker threads to perform the increments */
 for(t=0;t<NUM_THREADS;t++)
   pthread_create(&threads[t], NULL, aThread, NULL);

 /* Wait for the worker threads to end */
 for(t=0;t<NUM_THREADS;t++) {
   pthread_join(threads[t], &status);
   printf("thread %ld: failed_lock_attempts = %lu\n", t, \
            (unsigned long)status);
 }

 /* Show value of the counter */
 printf("GLOBAL_counter is %d: should be %d\n",
        GLOBAL_counter, NUM_THREADS*NUM_INCREMENTS);

 pthread_mutex_destroy(&GLOBAL_counter_mutex);
 pthread_exit(NULL);
}
```

Locking and Unlocking Mutexes

- **Example 8:** Example that shows how to acquire a lock within a timeout.

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h> // sleep
#include <errno.h> // ETIMEDOUT
#include <time.h> // struct timespec { time_t tv_sec; long tv_nsec;}

pthread_mutex_t GLOBAL_mutex = PTHREAD_MUTEX_INITIALIZER;
struct timespec GLOBAL_ts_future;

void *aThread (void *arg)
{
 long tid=(long)arg; int ret;

 ret=pthread_mutex_timedlock (&GLOBAL_mutex, \
                              &GLOBAL_ts_future);
 if (ret==0) {
    printf("thread %ld: before sleep\n",tid);
    sleep(5);
    printf("thread %ld: after sleep\n",tid);
    pthread_mutex_unlock (&GLOBAL_mutex);
 }
 else if (ret==ETIMEDOUT)
    printf("thread %ld: timeout happened\n",tid);

 printf("thread %ld: exiting\n",tid);
 pthread_exit(NULL);
}
```

```
int main()
{
 struct timespec ts_now;
 pthread_t thread1,thread2;

 clock_gettime(CLOCK_REALTIME, &ts_now);
 //GLOBAL_ts_future.tv_sec=ts_now.tv_sec+2; // A (timeout = 2s)
 GLOBAL_ts_future.tv_sec=ts_now.tv_sec+6; // B (timeout = 6s)
 GLOBAL_ts_future.tv_nsec=0;

 pthread_create(&thread1, NULL, aThread, (void*)1);
 pthread_create(&thread2, NULL, aThread, (void*)2);

 pthread_join(thread1, NULL);
 pthread_join(thread2, NULL);

 pthread_mutex_destroy(&GLOBAL_mutex);
 printf("thread main: ending\n");
 pthread_exit(NULL);
}
```

Locking and Unlocking Mutexes

- **Relation with thread cancellation**
  - although it may block indefinitely …
    `pthread_mutex_lock()` is NOT a cancellation point !
    - trying to cancel a thread blocked in `pthread_mutex_lock` won't work; the thread needs first to acquire the lock and then reach a cancellation point function so the thread may be cancelled there
      - see `man 7 pthreads` for a list of cancellation point functions
  - cancelling a thread that currently holds a lock does NOT invoque **`pthread_mutex_unlock()`** automatically !
    - other threads currently blocked on the same lock will block forever !
    - if a thread holds some lock(s), and there is the possibility of being cancelled, it should try to release the lock(s) before reaching a function that is a cancellation point
      - see `man 7 pthreads` for a list of cancellation point functions

Locking and Unlocking Mutexes

- **Example 9:**

This example supports many different scenarios of interaction between mutex locking, destruction and thread cancellation. See next slides for the details.

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h> // sleep
#include <errno.h> // EBUSY=16

pthread_mutex_t GLOBAL_counter_mutex = PTHREAD_MUTEX_INITIALIZER;

void *aThread (void *arg)
{
 long tid=(long)arg;

 pthread_mutex_lock (&GLOBAL_counter_mutex);
 printf("%ld: before sleep\n",tid);
 sleep(5); // a cancellation point
 printf("%ld: after sleep\n",tid);
 pthread_mutex_unlock (&GLOBAL_counter_mutex);

 pthread_exit(NULL);
}
```

```
int main()
{
 int ret;
 pthread_t thread1,thread2;

 pthread_mutex_lock (&GLOBAL_counter_mutex);
 pthread_create(&thread1, NULL, aThread, (void*)1);
 pthread_create(&thread2, NULL, aThread, (void*)2);
 pthread_mutex_unlock (&GLOBAL_counter_mutex);

 sleep(2); //A ("ensure" threads 1 and 2 reach pthread_mutex_lock)
 pthread_cancel(thread1); //B
 pthread_cancel(thread2); //C

 pthread_join(thread1, NULL); //D
 pthread_join(thread2, NULL); //E

 //sleep(2); //F ("ensure" threads 1 and 2 reach pthread_mutex_lock)
 ret=pthread_mutex_destroy(&GLOBAL_counter_mutex); //G
 printf("main: ret=%d\n", ret); //G
 pthread_exit(NULL);
}
```

Locking and Unlocking Mutexes

- **Example 9 – scenario 1:**
  - A,B,C,D,E commented; F,G uncommented
    - `pthread_mutex_destroy` returns `EBUSY`
    - possible output:

```
2: before sleep
main: ret=16
2: after sleep
1: before sleep
1: after sleep
```

Locking and Unlocking Mutexes

- **Example 9 – scenario 2:**

  - A,B,C,F commented; D,E,G uncommented
    - `pthread_join` ensures `pthread_mutex_destroy` succeeds
    - possible output:

```
1: before sleep
1: after sleep
2: before sleep
2: after sleep
main: ret=0
```

Locking and Unlocking Mutexes

- **Example 9 – scenario 3:**
  - A,B,C,D,E,F commented; G uncommented
    - `pthread_mutex_destroy` may or may not return `EBUSY`
    - possible output (returns `EBUSY`)

```
2: before sleep
main: ret=16
2: after sleep
1: before sleep
1: after sleep
```

mutex not destroyed once found busy; mutex integrity not affected; locking still effective; threads 1 and 2 execute serially as supposed

    - possible output (returns `0`):

```
main: ret=0
2: before sleep
1: before sleep
2: after sleep
1: after sleep
```

mutex destroyed; mutex integrity affected; locking is ineffective; threads 1 and 2 execute in parallel

Locking and Unlocking Mutexes

- **Example 9 – scenario 4:**
  - A,B,C,G uncommented; D,E,F commented
    - one `pthread_cancel` catches a thread in `sleep` (a cancellation point); the problem is that thread won't unlock the mutex
    - the other `pthread_cancel` catches the other thread in `pthread_mutex_lock`, which is NOT a cancellation point
      - as such, this thread will become blocked indefinitely
    - possible output (thread 2 cancelled, thread 1 blocked)

```
2: before sleep
main: ret=16
```

    - possible output (thread 1 cancelled, thread 2 blocked)

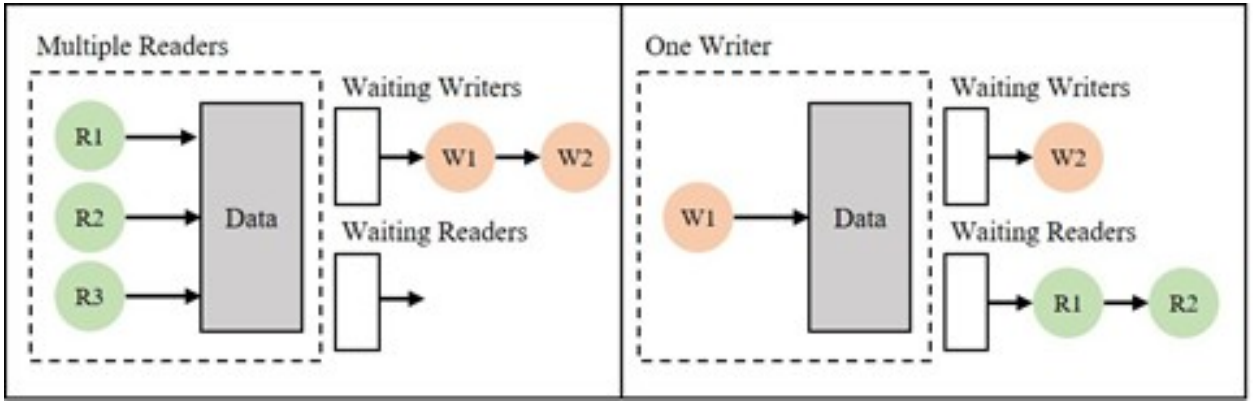
```
1: before sleep
main: ret=16
```

    - note: if D and E uncommented, main will block in join

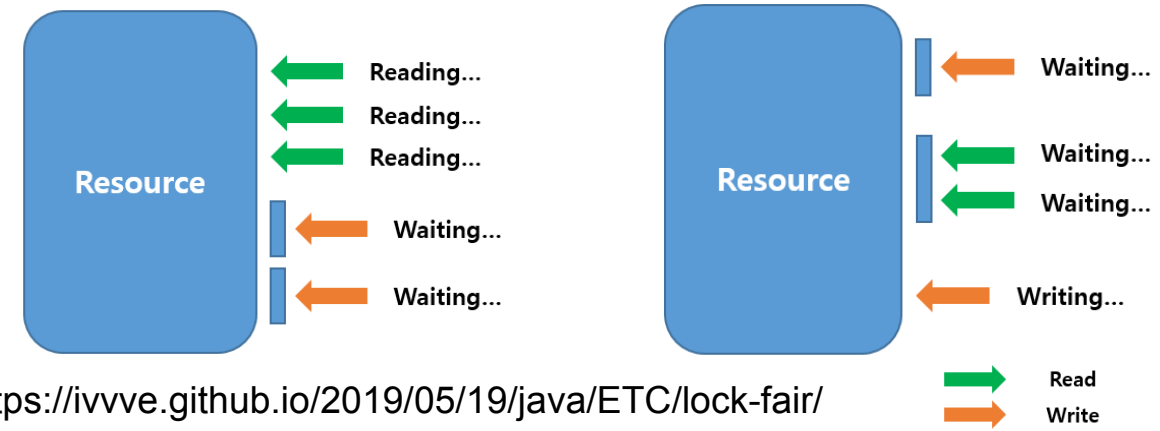
Read-Write Locks

Overview

- **Multiple-Readers-One-Writer semantics:**



https://oslab.kaist.ac.kr/ee488-fall-2022-homework-5/



https://ivvve.github.io/2019/05/19/java/ETC/lock-fair/

Overview

- **Read-Write Locks** allow **concurrent reads** of a protected shared resource and enforce **exclusive writes** to the resource
  - To **modify** the resource, a thread must 1st acquire the **Write lock**
  - To **read** the resource, a thread must 1st acquire the **Read lock**
  - Only **one thread** can hold the **Write lock** at any time
    - If one thread currently holds the Write lock, no other thread currenlty holds any lock (neither the Write, nor the Read lock)
  - **Many threads** may hold the **Read lock** at the same time
    - If one or more threads currently hold the Read lock, no other thread currently holds the Write lock

Overview

- **RW-Locks vs Mutex Variables:**
  - Mutex Variables always enforce exclusive access to a protected resource even if just to read the resource
  - RW-Locks are more flexible: exclusive acess is only enforced for a write operation (many-readers-one-writer)
  - If only exclusive access is intended, use Mutex Variables; it is much faster than locking/unlocking a write RW-Lock
  - If there is a mix of read and write accesses, use RW-Locks (specially if reading is more frequent than writing)
    - example: a database (75% reads, 25% writes)
- **RW-Locks Functions Return Values**
  - 0 on success, ≠0 on insucess (unless otherwise stated)

Creating and Destroying RW-Locks

- RW-Locks are declared with type **`pthread_rwlock_t`**

- **Initializing RW-Locks:**
  - RW-Locks must be initialized before they can be used
  - After initialization, the RW-Lock is initially *unlocked*
  - Two ways to initialize a RW-Lock:
    - **Statically**, when declared, assuming **default** attributes:
      `pthread_rwlock_t myrwlock=PTHREAD_RWLOCK_INITIALIZER`
    - **Dynamically**, with the `pthread_rwlock_init()` routine
      - This method permits setting RW-Lock object attributes

Creating and Destroying RW-Locks

- **Dynamic Initializing RW-Locks:**
  - `int` **`pthread_rwlock_init`**`(pthread_rwlock_t *`**`rwlock`**`, const pthread_rwlockattr_t *`**`attr`**`)`
    - initializes **`rwlock`** in an unlocked state, with the attributes in **`attr`**
    - **default** attribute values are assumed if **`attr`** is `NULL`
      - or if **`attr`** is initialized with **`pthread_rwlockattr_init`**
      - we always use **default** attributes; for further information on RW-Locks attributes, check the manuals of **`pthread_rwlockattr_*`**

- **Destroying RW-Locks:**
  - `int` **`pthread_rwlock_destroy`**`(pthread_rwlock_t *`**`rwlock`**`)`
    - destroys the RW-Lock object **`rwlock`** (frees up its resources)
      - the object may be later re-initialized with `pthread_rwlock_init`
    - if **`rwlock`** is locked or unitialized, result is undefined

Locking and Unlocking RW-Locks

- **Acquire / Attempt to Acquire a Read Lock:**
  - `int` **`pthread_rwlock_rdlock`**`(pthread_rwlock_t *`**`rwlock`**`)`
    - the caller tries to apply a R-lock on the pre-initialized **`rwlock`**
    - the calling thread acquires the R-lock if a writer thread does not hold **`rwlock`** (if the writer is the calling thread see (*)) and there are no writers blocked (waiting) on **`rwlock`**; otherwise, the caller blocks
    - the same thread may hold multiple R-locks on **`rwlock`**, that is, successfully called `pthread_rwlock_rdlock` *n* times; if so, it must match the unlocks (call `pthread_rwlock_unlock` *n* times)
    - (*) if caller owns the write lock, *deadlocks* (POSIX) or returns `EDEADLK` (Linux)
    - if **`rwlock`** is unitialized, result is undefined (**)
  - `int` **`pthread_rwlock_tryrdlock`**`(pthread_rwlock_t *`**`rwlock`**`)`
    - like `pthread_mutex_rdlock` but fails (returns) if it were to block
    - it always either acquires the R-lock or fails and returns `EBUSY` (**)

Locking and Unlocking RW-Locks

- **Acquire / Attempt to Acquire a Write Lock:**
  - `int` **`pthread_rwlock_wrlock`**`(pthread_rwlock_t *`**`rwlock`**`)`
    - the caller tries to apply a W-lock on the pre-initialized **`rwlock`**
    - the calling thread acquires the W-lock if none thread holds a lock (R or W) on **`rwlock`** (if the calling thread owns the W-lock see (*))
    - if the caller already owns a R-lock on **`rwlock`** it blocks (*deadlock*)
    - if **`rwlock`** is owned by another thread, the caller thread blocks
    - (*) if caller owns the write lock, *deadlocks* (POSIX) or returns `EDEADLK` (Linux)
    - if **`rwlock`** is unitialized, result is undefined (**)
  - `int` **`pthread_rwlock_trywrlock`**`(pthread_rwlock_t *`**`rwlock`**`)`
    - like `pthread_mutex_wrlock` but fails (returns) if it were to block
    - it always either acquires the W-lock or fails and returns `EBUSY` (**)
- **Acquire a RW-Lock within a timeout**
  - see **`pthread_rwlock_timedrdlock`** , **`pthread_rwlock_timedwrlock`**

Locking and Unlocking RW-Locks

- **Release a lock (unlocking):**
  - `int` **`pthread_rwlock_unlock`**`(pthread_rwlock_t *`**`rwlock`**`)`
    - the calling thread *releases* a lock (R or W) held on **`rwlock`**
      - releasing a R-Lock:
        - if there are other threads holding a R-Lock on the same **`rwlock`**, this object remains *locked* for reading
        - otherwise, the **`rwlock`** object becomes *unlocked*
      - releasing a W-Lock: **`rwlock`** becomes *unlocked*
      - if there are threads blocked on the **`rwlock`** when it becomes unlocked, the scheduling policy determines which thread(s) acquire the lock (Linux: blocked writers are prioritized)
    - **`rwlock`** in unlocked state: result undefined (POSIX); 0 (Linux)
    - **`rwlock`** not initialized: result undefined/`EINVAL` (POSIX); 0 (Linux)
    - of course, this operation *never* blocks the caller (by definition)

Locking and Unlocking RW-Locks

- **Summing up RW-Locks Usage (Linux implementation)**
  - note: bellow, the **`rwlock`** object is assumed to have been initialized
  - **W-locking**
    - if already W-locked by me, I will continue (returns `EDEADLK`)
    - if already R-locked by me, I will block forever (become *deadlock*)
    - if already locked (R or W) by other(s), I will block until all unlock
    - if currently unlocked, I will grab the lock and continue (returns 0)
  - **R-locking**
    - if already W-locked by me, I will continue (returns `EDEADLK`)
    - if already R-locked by me, I will continue (returns 0)
    - if already R-locked by other(s), I will continue (returns 0)
    - if already W-locked by other, I will block until it unlocks
    - if currently unlocked, I will grab the lock and continue (returns 0)

Locking and Unlocking RW-Locks

- **Summing up RW-Locks Usage (Linux implementation)**
  - note: bellow, the **`rwlock`** object is assumed to have been initialized
  - **unlocking**
    - if currently locked by me, I will release the lock
      - if it was a W-lock, the lock object becomes unlocked
      - if it was a R-lock, the lock object becomes unlocked
        - but only if no other thread holds the R-lock
    - if currently locked by other(s), it keeps locked and returns 0
    - if currently unlocked, it keeps unlocked and returns 0

Yes, its a little messy …
But this summary should help !

Locking and Unlocking RW-Locks

- **Example 10A:**

Shows usage of a RW-lock to protect the update of a global **counter** by several threads. Test with the Write-lock, the Read-lock, without any locking and with mutex variables. Compare execution time.

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

#define NUM_THREADS 4
#define NUM_INCREMENTS 10000000
int GLOBAL_counter=0;

//pthread_mutex_t GLOBAL_counter_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_rwlock_t GLOBAL_counter_rwlock = PTHREAD_RWLOCK_INITIALIZER;

void *aThread (void *arg)
{
 for (int i=0; i<NUM_INCREMENTS; i++) {
   //pthread_mutex_lock (&GLOBAL_counter_mutex);
   pthread_rwlock_wrlock (&GLOBAL_counter_rwlock);
   //pthread_rwlock_rdlock (&GLOBAL_counter_rwlock);
   GLOBAL_counter ++;
   pthread_rwlock_unlock (&GLOBAL_counter_rwlock);
   //pthread_mutex_unlock (&GLOBAL_counter_mutex);
 }

 pthread_exit(NULL);
}
```

```
int main()
{
 long t;
 pthread_t threads[NUM_THREADS];

 /* Create worker threads to perform the increments */
 for(t=0;t<NUM_THREADS;t++)
   pthread_create(&threads[t], NULL, aThread, NULL);

 /* Wait for the worker threads to end */
 for(t=0;t<NUM_THREADS;t++)
   pthread_join(threads[t], NULL);

 /* Show value of the counter */
 printf("GLOBAL_counter is %d: should be %d\n",
        GLOBAL_counter, NUM_THREADS*NUM_INCREMENTS);

 //pthread_mutex_destroy(&GLOBAL_counter_mutex);
 pthread_rwlock_destroy(&GLOBAL_counter_rwlock);
 pthread_exit(NULL);
}
```

Locking and Unlocking RW-Locks

- **Example 10B:**

Similar to Example 10A but shows the failed lock attempts per thread (use code // A in alternative to code // B).

Question: how many failed locks attempts are there in version // B ?

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <errno.h> // EBUSY

#define NUM_THREADS 4
#define NUM_INCREMENTS 10000000
int GLOBAL_counter;

pthread_rwlock_t GLOBAL_counter_rwlock = PTHREAD_RWLOCK_INITIALIZER;

void *aThread (void *arg)
{
 unsigned long failed_lock_attempts=0;
 int local_counter;

 for (int i=0; i<NUM_INCREMENTS; i++) {
   while (pthread_rwlock_trywrlock \
            (&GLOBAL_counter_rwlock)\ == EBUSY) // A
   //while (pthread_rwlock_tryrdlock \
   //         (&GLOBAL_counter_rwlock)\ == EBUSY) // B
       failed_lock_attempts ++;
   local_counter = GLOBAL_counter ++;
   pthread_rwlock_unlock (&GLOBAL_counter_rwlock);
 }

 pthread_exit((void*)failed_lock_attempts);
}
```

```
int main()
{
 long t; void *status;
 pthread_t threads[NUM_THREADS];

 /* Create worker threads to perform the increments */
 for(t=0;t<NUM_THREADS;t++)
   pthread_create(&threads[t], NULL, aThread, NULL);

 /* Wait for the worker threads to end */
 for(t=0;t<NUM_THREADS;t++)
   pthread_join(threads[t], &status);

 /* Show value of the counter */
 printf("GLOBAL_counter is %d: should be %d\n",
         GLOBAL_counter, NUM_THREADS*NUM_INCREMENTS);

 pthread_rwlock_destroy(&GLOBAL_counter_rwlock);
 pthread_exit(NULL);
}
```

Condition Variables

Overview

- **Condition variables:** another way for threads to synchronize
  - While **mutexes/RW-Locks** implement synchronization by controlling thread access to data, **condition variables** allow threads to synchronize based upon the actual value of data

- **Without condition variables**, threads would **continually poll** (possibly in a critical section), to check if a condition is met
  - This *busy waiting* can be very resource (CPU) consuming
  - A condition variable achieves the same goal without polling

- A **condition variable** is always used together with a **mutex** lock

Overview

- A representative sequence for using condition 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 the point 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. Note that a 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 and atomically 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

Creating and Destroying Condition Variables

- Condition variables are declared with type **`pthread_cond_t`**
- **Initializing Condition Variables:**
  - Condition variables must be initialized before used
  - Two ways to initialize a condition variable:
    - **Statically**, when declared, assuming **default** attributes: `pthread_cond_t mycond=PTHREAD_COND_INITIALIZER`
    - **Dynamically**, with the `pthread_cond_init()` routine
      - This method permits setting condition variable object attributes

Creating and Destroying Condition Variables

- **Dynamic Initializing Condition Variables:**
  - `int` **`pthread_cond_init`**`(pthread_cond_t *`**`cond`**`, const pthread_condattr_t *`**`attr`**`)`
    - initializes the specified **`cond`** with the attributes in **`attr`**
    - **default** attribute values are assumed if **`attr`** is `NULL`
      - or if **`attr`** is initialized with **`pthread_condattr_init`**

- **Destroying Condition Variables:**
  - `int` **`pthread_cond_destroy`**`(pthread_cond_t *`**`cond`**`)`
    - destroys the condition variable **`cond`** (frees up its resources)
      - the object may be later re-initialized with `pthread_cond_init`
    - no threads should be waiting on the variable **`cond`** on entrance or `pthread_cond_wait` will return `EBUSY`
    - Linux: it only checks if the condition has no waiting threads

Waiting and Signaling on Condition Variables

- **Waiting on a Condition Variable:**
  - `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)`
    - the calling thread *waits* until the condition **`cond`** is *signaled*
    - must be called with the **`mutex`** previously *locked*
    - automatically (1) *unlocks* the **`mutex`** before (2) *waits* (1+2 atomically)
      - the calling thread execution is *suspended* and does not consume any CPU time until the condition variable is *signaled*
    - after a signal is received and the thread awakes, **`mutex`** will be automatically *locked* for use by the thread; the programmer is responsible for *unlocking* **`mutex`** when the thread is finished with it
  - `int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,const struct timespec *abstime)`
    - like `pthread_cond_wait` but if **`cond`** has not been signaled until the future time **`abstime`**, gives up waiting and reacquires **`mutex`**

Waiting and Signaling on Condition Variables

- **Signaling on a Condition Variable:**
  - `int` **`pthread_cond_signal`**`(pthread_cond_t *`**`cond`**`)`
    - the calling thread *signals* other threads *waiting* on condition **`cond`** and **>=1** of those threads will wake up from `pthread_cond_wait`
      - if several threads are waiting, more than one may restart – see [https://stackoverflow.com/questions/55935188/can-pthread-cond-signal-make-more-than-one-thread-to-wake-up]
      - if several threads are waiting, which one(s) restart(s) is unspecified
    - must be called with the accompanying **`mutex`** previously *locked*
    - after returns, the **`mutex`** must be *unlocked* (to allow at least one signaled thread that was *waiting* to *wake up* and *lock* the **`mutex`**)
  - `int` **`pthread_cond_broadcast`**`(pthread_cond_t *`**`cond`**`)`
    - like `pthread_cond_signal` but wakes **all** threads waiting on **`cond`**
  - both functions have no effect if no threads are waiting on **`cond`**

warning: if the thread signaled is not blocked in `pthread_cond_wait`, it will miss the signals from `pthread_cond_signal` because these signals do not queue up !!

Waiting and Signaling on Condition Variables

- **Code snippets from the man pages:**

```
// Consider two shared variables x and y, protected by
// the mutex mut, and a condition  variable cond that
// is to be signaled whenever x becomes greater than y

int x,y;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

// Waiting until x is greater than y is performed as follows
// (note the use of while instead of if; why not if ? (*))
pthread_mutex_lock(&mut);
while (x <= y)
  pthread_cond_wait(&cond, &mut);
/* operate on x and y */
pthread_mutex_unlock(&mut);

// Modifications on x and y that may cause x to become
// greater than y should signal the condition if needed:
pthread_mutex_lock(&mut);
/* modify x and y */
if (x > y) pthread_cond_broadcast(&cond);
pthread_mutex_unlock(&mut);

(*) 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; using while ensures they will retest the condition and
will block (wait) again if the conditions still holds true
```

```
// To wait for x to becomes greater than
// y with a timeout of 5 seconds, do

struct timeval now;
struct timespec timeout;
int retcode;

pthread_mutex_lock(&mut);
gettimeofday(&now);
timeout.tv_sec = now.tv_sec + 5;
timeout.tv_nsec = now.tv_usec * 1000;
retcode = 0;
while (x <= y && retcode != ETIMEDOUT)
  retcode = pthread_cond_timedwait
                (&cond, &mut, &timeout);
if (retcode == ETIMEDOUT) {
  /* timeout occurred */
} else {
  /* operate on x and y */
}
pthread_mutex_unlock(&mut);
```

Waiting and Signaling on Condition Variables

- **Example 11A:**

The main/producer thread creates a secondary/consumer thread. The producer reads a number from the keyboard; the consumer shows that number in the screen. Using condition variables ensures the consumer only shows the number **after** it has been produced. Two situations may happen: a) the consumer locks in C1 before the producer locks in P1; condition C2 is true and the consumer waits in C3; the producer unlocks in P1 and signals the consumer in P2; b) the producer locks in P1 before the consumer locks in C1; the producer signals the consumer in P2 but the signal is lost once the consumer is still in C1; when the consumer reaches C2, this condition is false and so it will not wait in C3, which is correct.

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h> // sleep

int GLOBAL_number=-1;
pthread_mutex_t GLOBAL_mutex=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t GLOBAL_cond=PTHREAD_COND_INITIALIZER;

void* consumer(void *arg)
{
 pthread_mutex_lock(&GLOBAL_mutex); //C1
 // verify if the number was produced; if not, wait
 while (GLOBAL_number==-1) { //C2
     printf("consumer: waiting\n");
     pthread_cond_wait(&GLOBAL_cond, &GLOBAL_mutex); //C3
 }
 printf("consumer: %d\n", GLOBAL_number);
 pthread_mutex_unlock(&GLOBAL_mutex);

 pthread_exit(NULL);
}
```

```
int main() // producer
{
 pthread_t thread;

 pthread_create(&thread, NULL, consumer, NULL);
 //sleep(1); // make consumer gain the lock first

 pthread_mutex_lock(&GLOBAL_mutex); //P1
 printf("producer: "); scanf("%d", &GLOBAL_number);
 // notify consumer that the number was produced
 pthread_cond_signal(&GLOBAL_cond); //P2
 pthread_mutex_unlock(&GLOBAL_mutex);

 pthread_join(thread, NULL);
 pthread_exit(NULL);
}
```

Waiting and Signaling on Condition Variables

- **Example 11B:**

Similar to Example 11A, but with a timeout: the consumer gives up waiting for the signal after 5 seconds. Again, the consumer may lock the mutex 1st, or the producer, but the program always behaves correctly: a) if the producer locks 1st, the consumer will never wait once when it locks it will find the number already filled; b) to force the consumer to lock 1st, uncomment the `sleep` ; however, note that if the `scanf` takes more than 5s, after the consumer wakes up inside `pthread_cond_timedwait` by timeout, it will block when trying to lock the mutex, rendering the timeout ineffective (locking will only succeed after the unlock in the producer, which only happens after `scanf` ; also, the signal sent to the consumer will be lost, once he is not awaiting for it anymore); c) to make the timeout effective replace `sleep(1)` by `sleep(n)` (n>5)

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h> // sleep
#include <errno.h> // ETIMEDOUT
#include <time.h> // struct timespec { time_t tv_sec; long tv_nsec;}

int GLOBAL_number=-1;
pthread_mutex_t GLOBAL_mutex=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t GLOBAL_cond=PTHREAD_COND_INITIALIZER;

void* consumer(void *arg)
{
 int ret=-1; struct timespec ts_now, ts_future;
 clock_gettime(CLOCK_REALTIME, &ts_now);
 ts_future.tv_sec=ts_now.tv_sec+5; //(timeout=5s)
 ts_future.tv_nsec=0;

 pthread_mutex_lock(&GLOBAL_mutex);
 // verify if the number was produced; if not, wait up to 5s
 while (GLOBAL_number==-1 && ret != ETIMEDOUT) {
     printf("consumer: waiting\n");
     ret=pthread_cond_timedwait(&GLOBAL_cond, \
                                &GLOBAL_mutex, &ts_future);
 }
 if (ret == ETIMEDOUT) printf("consumer: timedout\n");
 else printf("consumer: %d\n", GLOBAL_number);
 pthread_mutex_unlock(&GLOBAL_mutex);

 pthread_exit(NULL);
}
```

```
int main() // producer
{
 pthread_t thread;

 pthread_create(&thread, NULL, consumer, NULL);
 //sleep(1); // make consumer lock first

 pthread_mutex_lock(&GLOBAL_mutex);
 printf("producer: "); scanf("%d", &GLOBAL_number);
 // notify consumer that the number was produced
 pthread_cond_signal(&GLOBAL_cond);
 pthread_mutex_unlock(&GLOBAL_mutex);

 pthread_exit(NULL);
}
```

Waiting and Signaling on Condition Variables

- **Example 11C:**

Similar to Example 11A, but with **loops** so the producer produces a number, and the consumer shows it, while the number is not zero. Note that nothing prevents the producer to grab the lock several times in a row and send several signals to the consumer, but they will be lost because the consumer may not yet be waiting for them. Uncommenting the `sleep` "ensures" the consumer grabs the lock.

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h> // sleep

int GLOBAL_number=-1;
pthread_mutex_t GLOBAL_mutex=PTHREAD_MUTEX_INITIALIZER
pthread_cond_t GLOBAL_cond=PTHREAD_COND_INITIALIZER;

void* consumer(void *arg)
{
 int end_loop=0;
 do {
     pthread_mutex_lock(&GLOBAL_mutex);
     // verify if the number was produced; if not, wait
     while (GLOBAL_number==-1) {
        printf("consumer: waiting\n");
        pthread_cond_wait(&GLOBAL_cond, &GLOBAL_mutex)
     }
     printf("consumer: %d\n", GLOBAL_number);
     if (GLOBAL_number==0) end_loop=1;
     else GLOBAL_number=-1;
     pthread_mutex_unlock(&GLOBAL_mutex);
 } while (!end_loop);

 pthread_exit(NULL);
}
```

```
int main() // producer
{
 pthread_t thread; int end_loop=0;

 pthread_create(&thread, NULL, consumer, NULL);

 do {
    //sleep(1); // make consumer gain lock first

    pthread_mutex_lock(&GLOBAL_mutex);
    printf("producer: "); scanf("%d",&GLOBAL_number);
    if (GLOBAL_number==0) end_loop=1;
    // notify consumer that the number was produced
    pthread_cond_signal(&GLOBAL_cond);
    pthread_mutex_unlock(&GLOBAL_mutex);
 } while (!end_loop);

 pthread_join(thread, NULL);
 pthread_exit(NULL);
}
```

Waiting and Signaling on Condition Variables

- **Example 11D:**

Similar to Example 11C, but solves the possible starvation of the consumer: the producer does not acquire a new number while the consumer has not consumed the current. This solution enforces the alternation of producer and consumer. `pthread_broadcast` is used, once both threads are receivers. Execute with and without `sleep`. Also, note that `broadcast` is now used instead of `signal` (*). Why ?

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h> // sleep

int GLOBAL_number=-1;
pthread_mutex_t GLOBAL_mutex=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t GLOBAL_cond=PTHREAD_COND_INITIALIZER;

void* consumer(void *arg)
{
 int end_loop=0;
 do {
     pthread_mutex_lock(&GLOBAL_mutex);
     // verify if the number was produced; if not, wait
     while (GLOBAL_number==-1) {
        printf("consumer: waiting\n");
        pthread_cond_wait(&GLOBAL_cond, &GLOBAL_mutex);
     }
     printf("consumer: %d\n", GLOBAL_number);
     if (GLOBAL_number==0) end_loop=1;
     else GLOBAL_number=-1;
     pthread_cond_broadcast(&GLOBAL_cond); // (*)
     pthread_mutex_unlock(&GLOBAL_mutex);
 } while (!end_loop);

 pthread_exit(NULL);
}
```

```
int main() // producer
{
 pthread_t thread; int end_loop=0;
 pthread_create(&thread, NULL, consumer, NULL);

 do {
    //sleep(1); // make consumer gain lock first

    pthread_mutex_lock(&GLOBAL_mutex);
    while (GLOBAL_number!=-1) {
        printf("producer: waiting\n");
        pthread_cond_wait(&GLOBAL_cond,&GLOBAL_mutex);
    }
    printf("producer: "); scanf("%d",&GLOBAL_number);
    if (GLOBAL_number==0) end_loop=1;
    // notify consumer that the number was produced
    pthread_cond_broadcast(&GLOBAL_cond); // (*)
    pthread_mutex_unlock(&GLOBAL_mutex);
 } while (!end_loop);

 pthread_join(thread, NULL);
 pthread_exit(NULL);
}
```

Waiting and Signaling on Condition Variables

- **Example 12:**

The main thread creates a worker thread. The main thread will output 'B', and the worker thread will output 'A' and 'C'. The program ensures the sequence 'A','B','C'. Note: if X happens before Z, the signal to main is just lost because main hadn't yet reached Y; however, main will not block indefinitely on Y because fails condition W.

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

int flag_worker=0;
pthread_mutex_t mutex_worker=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond_worker=PTHREAD_COND_INITIALIZER;
int flag_main=0;
pthread_mutex_t mutex_main=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond_main=PTHREAD_COND_INITIALIZER;

void* worker(void *arg)
{
 printf("task A\n");
 pthread_mutex_lock(&mutex_worker);
 flag_worker=1;
 pthread_cond_signal(&cond_worker); //X
 pthread_mutex_unlock(&mutex_worker);

 pthread_mutex_lock(&mutex_main);
 while (flag_main != 1)
     pthread_cond_wait(&cond_main, &mutex_main);
 pthread_mutex_unlock(&mutex_main);
 printf("task C\n");

 pthread_exit(NULL);
}
```

```
int main();
{
 pthread_t thread;
 pthread_create(&thread, NULL, worker, NULL);

 pthread_mutex_lock(&mutex_worker); //Z
 while (flag_worker != 1) //W
     pthread_cond_wait(&cond_worker, &mutex_worker); //Y
 pthread_mutex_unlock(&mutex_worker);

 printf("task B\n");
 pthread_mutex_lock(&mutex_main);
 flag_main=1;
 pthread_cond_signal(&cond_main);
 pthread_mutex_unlock(&mutex_main);

 pthread_exit(NULL);
}
```

Waiting and Signaling on Condition Variables

- **Example 13A:**

The producer (main) thread creates a consumer thread. The producer generates and shows N random numbers. If a number is even, it must be also shown by the consumer. This solution is problematic: the producer may generate many even numbers, and send the corresponding signals, before the consumer has the chance to block in `pthread_cond_wait`.

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

#define N 10
int number=1;
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond=PTHREAD_COND_INITIALIZER;

void* consumer(void *arg)
{
 while(1) {
   pthread_mutex_lock(&mutex);
   while (number % 2 != 0)
     pthread_cond_wait(&cond, &mutex);
   printf("consumer: number = %d\n", number);
   number=1;
   pthread_mutex_unlock(&mutex);
 }
 pthread_exit(NULL);
}
```

```
int main()
{
 pthread_t thread;

 pthread_create(&thread, NULL, consumer, NULL);

 int i=0; unsigned int seedp=getpid();
 while(++i <= N) {
     pthread_mutex_lock(&mutex);
     number = rand_r(&seedp) % 10;
     printf("producer: i = %d: number = %d\n",
             i, number);
     if (number % 2 == 0)
         pthread_cond_signal(&cond);
     pthread_mutex_unlock(&mutex);
 }
 pthread_cancel(thread);
 pthread_exit(NULL);
}
```

Waiting and Signaling on Condition Variables

- **Example 13B:**

Like Example 13A but when the producer generates an even random it will only generate the next random after the consumer has consumed the current even number. Also, `broadcast` is used instead of `signal` (*). Why ?

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

#define N 10
int number=1;
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond=PTHREAD_COND_INITIALIZER;

void* consumer(void *arg)
{
 while(1) {
     pthread_mutex_lock(&mutex);
     while (number % 2 != 0)
         pthread_cond_wait(&cond, &mutex);
     printf("consumer: number = %d\n", number);
     number=1;
     pthread_cond_broadcast(&cond); // (*)
     pthread_mutex_unlock(&mutex);
 }
 pthread_exit(NULL);
}
```

```
int main()
{
 pthread_t thread;

 pthread_create(&thread, NULL, consumer, NULL);

 int i=0; unsigned int seedp=getpid();
 while(++i <= N) {
     pthread_mutex_lock(&mutex);
     while(number % 2 == 0)
         pthread_cond_wait(&cond, &mutex);
     number = rand_r(&seedp) % 10;
     printf("producer: i = %d: number = %d\n",
             i, number);
     if (number % 2 == 0)
         pthread_cond_broadcast(&cond); // (*)
     pthread_mutex_unlock(&mutex);
 }
 pthread_cancel(thread);
 pthread_exit(NULL);
}
```

Waiting and Signaling on Condition Variables

- **Example 14:**

The main thread creates three worker threads: T1,T2,T3. Threads T2 and T3 increment a global "count" variable COUNT_INCREMENTS times, one unit at a time. Thread T1 awaits for “count” to reach a COUNT_LIMIT value; when that happens, T1 is signaled by T2 or T3, awakens, adds 100 to “count” and ends. T2 and T3 then continue incrementing “count” until both reach COUNT_INCREMENTS increments. The main thread awaits for all workers to finish and prints the final value of “count”.

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

#define NUM_THREADS  3
#define COUNT_INCREMENTS 10
#define COUNT_LIMIT 12

int count = 0;
pthread_mutex_t count_mutex;
pthread_cond_t count_cond;
```

```
void *add_1(void *t)
{
  int i;
  long my_id = (long)t;

  printf("add_1(): thread %ld, Starting\n", my_id);

  for (i=0; i < COUNT_INCREMENTS; i++) {
    pthread_mutex_lock(&count_mutex);
    count++;
    printf("add_1(): thread %ld, count = %d\n", my_id, count);

    //Check the value of count and signal waiting thread when condition
    //is reached. Note that this occurs while mutex is locked.
    if (count == COUNT_LIMIT) {
      printf("add_1(): thread %ld, count = %d, Threshold reached\n",
              my_id, count);
      pthread_cond_signal(&count_cond);
      printf("add_1: thread %ld, Just sent signal\n",my_id);
    }
    pthread_mutex_unlock(&count_mutex);

    // Simulate some work so threads can alternate on mutex lock
    sleep(1);
  }

  printf("add_1(): thread %ld, Ending\n", my_id);
  pthread_exit(NULL);
}
```

Waiting and Signaling on Condition Variables

- **Example 14:**

The main thread creates three worker threads: T1,T2,T3. Threads T2 and T3 increment a global "count" variable COUNT_INCREMENTS times, one unit at a time. Thread T1 awaits for “count” to reach a COUNT_LIMIT value; when that happens, T1 is signaled by T2 or T3, awakens, adds 100 to “count” and ends. T2 and T3 then continue incrementing “count” until both reach COUNT_INCREMENTS increments. The main thread awaits for all workers to finish and prints the final value of “count”.

```
void *add_100(void *t)
{
  long my_id = (long)t;

  printf("add_100(): thread %ld, Starting\n", my_id);

  // Lock mutex and wait for signal. pthread_cond_wait will automatically
  // and atomically unlock mutex and wait. In case COUNT_LIMIT is reached
  // before this routine is run by the waiting thread, the loop will be
  // skipped to prevent pthread_cond_wait from being called and never return.
  pthread_mutex_lock(&count_mutex);
  while (count < COUNT_LIMIT) {
    printf("add_100(): thread %ld, count = %d, Going into wait...\n", my_id,count);
    pthread_cond_wait(&count_cond, &count_mutex);
    printf("add_100(): thread %ld, Condition signal received, count = %d\n", my_id,count);
  }
  printf("add_100: thread %ld, Updating the value of count...\n", my_id);
  count += 100;
  printf("add_100: thread %ld, count now = %d\n", my_id, count);
  printf("add_100: thread %ld Unlocking mutex\n", my_id);
  pthread_mutex_unlock(&count_mutex);

  printf("add_100: thread %ld, Ending\n", my_id);
  pthread_exit(NULL);
}
```

Waiting and Signaling on Condition Variables

- **Example 14:**

The main thread creates three worker threads: T1,T2,T3. Threads T2 and T3 increment a global "count" variable COUNT_INCREMENTS times, one unit at a time. Thread T1 awaits for “count” to reach a COUNT_LIMIT value; when that happens, T1 is signaled by T2 or T3, awakens, adds 100 to “count” and ends. T2 and T3 then continue incrementing “count” until both reach COUNT_INCREMENTS increments. The main thread awaits for all workers to finish and prints the final value of “count”.

```
int main()
{
  pthread_t threads[3];

  /* Initialize mutex and condition variable objects */
  pthread_mutex_init(&count_mutex, NULL);
  pthread_cond_init (&count_cond, NULL);

  pthread_create(&threads[0], NULL, add_100, (void *)1);
  pthread_create(&threads[1], NULL, add_1, (void *)2);
  pthread_create(&threads[2], NULL, add_1, (void *)3);

  /* Wait for all threads to complete */
  for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(threads[i], NULL);
  printf ("Main(): Waited and joined with %d threads. Final value of count = %d. Done.\n",
           NUM_THREADS, count);

  /* Clean up and exit */
  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_cond);
  pthread_exit (NULL);
}
```

Barriers

Overview

- **Barriers:** yet another way for threads to synchronize
  - ensures a certain number of threads reaches a certain synchronization point in the program (the barrier) before progressing to the remaining of their instructions
  - threads reaching a barrier block until all threads supposed to reach the barrier have done so; then, all threads unblock
  - a multi-threaded program may have several barriers; each barrier acts as a rendezvous point for a number of threads
  - an *active barrier* could be implemented with a global counter protected by a mutex and having involved threads doing busy waiting; this approach wastes CPU, specially if one or more threads get delayed in reaching the barrier
- Barriers are declared with type **`pthread_barrier_t`**

Creating and Destroying Barriers

- **Dynamic Initializing Barriers:**
  - no static initializer provided
  - `int` **`pthread_barrier_init`**`(pthread_barrier_t *`**`barrier`**`, const pthread_barrierattr_t *`**`attr`**`, unsigned` **`count`**`)`
    - initializes the specified **`barrier`** with the attributes in **`attr`** and the necessary number of caller threads before returning in **`count`**
    - **default** attribute values are assumed if **`attr`** is `NULL`
      - or if **`attr`** is initialized with **`pthread_barrierattr_init`**
- **Destroying Barriers:**
  - `int` **`pthread_barrier_destroy`**`(pthread_barrier_t *`**`barrier`**`)`
    - destroys the barrier **`barrier`** (frees up its resources)
      - the object may be later re-initialized with `pthread_barrier_init`
    - should not be invoked if there are threads blocked in the **`barrier`**

Waiting on Barriers

- **Waiting on a Barrier:**
  - `int` **`pthread_barrier_wait`**`(pthread_barrier_t *`**`barrier`**`)`

    - the calling thread *blocks* until the required number of threads (count) have called `pthread_barrier_wait` specifying the **`barrier`**

    - when the required number of threads is reached, on one of them returns `PTHREAD_BARRIER_SERIAL_THREAD` and 0 on the others

    - after `pthread_barrier_wait` returns in all threads, the **`barrier`** is reset to the state of its last initialization (including the counter)

Waiting on Barriers

- **Example 15:**

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

pthread_barrier_t GLOBAL_barrier;

void *aThread(void *arg)
{
 long tid=(long)arg;

 for (int i=0; i<10; i++) {
    printf("thread %ld: i=%d\n", tid, i);
    pthread_barrier_wait(&GLOBAL_barrier);
 }

 pthread_exit(NULL);
}
```

```
int main()
{
 pthread_t threads[NUM_THREADS];
 long t; int ret;

 ret = pthread_barrier_init(&GLOBAL_barrier, NULL, NUM_THREADS);
 if (ret){
   printf("ERROR; return code from pthread_barrier_init() is %d\n", ret);
   exit(ret);
 }

 for(t=0;t<NUM_THREADS;t++){
   printf("thread main: creating thread %ld\n", t);
   ret = pthread_create(&threads[t], NULL, aThread, (void *)t);
   if (ret){
     printf("ERROR; return code from pthread_create() is %d\n", ret);
     exit(ret);
   }
 }

 for(t=0; t<NUM_THREADS; t++) {
   ret = pthread_join(threads[t], NULL);
   if (ret) {
     printf("ERROR; return code from pthread_join() is %d\n", ret);
     exit(ret);
   }
   printf("thread main: joined with thread %ld\n", t);
 }

 pthread_barrier_destroy(&GLOBAL_barrier); // safe only after joining
 printf("thread main: exiting\n");
 pthread_exit(NULL);
}
```

Appendix

Creating n processes vs n Threads

- **appendix_1:**

```
// appendix_1-create-many-processes.c

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/wait.h>

#define NUMPROCS 20000

int main()
{
 pid_t pid;

 for (int i=0; i<NUMPROCS; i++) {
   pid=fork();
   if (pid==0) { exit(0); }
   else if (pid==-1) {
     printf("Failed on the creation of process
i=%d\n",i);
     perror("fork"); exit(errno);
   }
 }

 while(wait(NULL)!=-1);
 return(0);
}
```

```
// appendix_1-create-many-threads.c

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NUM_THREADS 20000

void *aThread(void *tid)
{
  //pthread_detach(pthread_self()); // (*1)
  pthread_exit(NULL);
}

int main()
{
 pthread_t threads[NUM_THREADS];
 int ret;

 for(long t=0;t<NUM_THREADS;t++){
   ret = pthread_create(&threads[t], NULL, aThread, (void *)t);
   if (ret!=0){
     printf("Failed on the creation of thread t=%ld\n", t);
     printf("Return code from pthread_create() is %d\n", ret);
     exit(ret);
   }
   // pthread_detach(threads[t]); // (*2)
 }

 pthread_exit(NULL);
}
```

- to execute in a single core:
time taskset --cpu-list n ./create-many-... #where n is the core selected (n=0,1,…)
- in this example, invoking pthread_detach, at (*1) or (*2), will in fact increase the execution time

Managing Thread Attribute Objects

- **POSIX Thread Attributes:**
  - **Detachstate**: controls whether the thread is created detached; if the thread is created detached, then use of the ID of the newly created thread by `pthread_detach` or `pthread_join` functions is an error
    - `pthread_attr_getdetachstate, pthread_attr_setdetachstate`
  - **Schedpolicy**: specific scheduling policy (FIFO, RR, OTHER, ...)
    - `pthread_attr_getschedpolicy, pthread_attr_setschedpolicy`
  - **Schedparam**: specific scheduling parameters (priority, period,...)
    - `pthread_attr_getschedparam, pthread_attr_setschedparam`
  - **Inheritsched**: determines how other scheduling attributes of the created thread shall be set (explicit or inherited from the creator thread)
    - `pthread_attr_getinheritsched, pthread_attr_setinheritsched`
  - **Contentionscope**: the thread scheduling scope (process or system)
    - `pthread_attr_getscope, pthread_attr_setscope`

Managing Thread Attribute Objects

- **POSIX Thread Attributes (cont):**
  - **Stackaddr**: the thread's stack base address
    - `pthread_attr_getstackaddr, pthread_attr_setstackaddr`
      - Linux: obsolete, use (*) instead
  - **Stacksize**: the thread's stack size
    - `pthread_attr_getstacksize, pthread_attr_setstacksize`
  - **Stack**: both the thread's stack base address and size
    - (*) `pthread_attr_getstack, pthread_attr_setstack`
  - **Guardsize**: controls the size of the guard area for the thread's stack; the guardsize provides protection against overflow of the stack pointer
    - `pthread_attr_getguardsize, pthread_attr_setguardsize`
    - ignored if the caller is allocating the thread's stack

Managing Thread Attribute Objects

- **Initializing Thread Attribute Objects**
  - `int` **`pthread_attr_init`**`(pthread_attr_t *`**`attr`**`)`
    - initializes the thread attribute object **`attr`** with **default** values (note: for default values it is enough to pass `NULL` as **`attr`** in `pthread_create`)
    - default attribute values:
      - **Detachstate**: `PTHREAD_CREATE_JOINABLE`
      - **Schedpolicy**: `SCHED_OTHER`; **Schedparam**: (implementation dependent)
      - **Inheritsched**: `PTHREAD_INHERIT_SCHED` (inherit from the creator)
      - **Contentionscope**: (implementation dependent); Linux only supports `PTHREAD_SCOPE_SYSTEM`
      - **Stack*, Guardsize**: (implementation dependent); in Linux, default **Stacksize** is 2MB (mininum is 16K), and **Guardsize** is page size
- **Destroying Thread Attribute Objects**
  - `int` **`pthread_attr_destroy`**`(pthread_attr_t *`**`attr`**`)`
    - destroys the **`attr`** object, which must not be reused until reinitialized

Managing Thread Attribute Objects

- **appendix_2:**

Shows how to use the `pthread_attr_get*/set*` functions to query and set the different thread attributes. Shows default values unless the // code is uncommented.

```
#include <stdio.h>
#include <pthread.h>
#include <limits.h> // PTHREAD_STACK_MIN
#include <stdlib.h>
#include <errno.h> // EINVAL/ENOSTUP
#include <unistd.h> // getpagesize

void show_detachstate (int detachstate)
{
 if (detachstate == PTHREAD_CREATE_JOINABLE)
   printf("detachstate is PTHREAD_CREATE_JOINABLE\n");
 else if (detachstate == PTHREAD_CREATE_DETACHED)
   printf("detachstate is PTHREAD_CREATE_DETACHED\n");
}

void show_schedpolicy (int schedpolicy)
{
 if (schedpolicy == SCHED_OTHER)
   printf("schedpolicy is SCHED_OTHER\n");
 else if (schedpolicy == SCHED_FIFO)
   printf("schedpolicy is SCHED_FIFO\n");
 else if (schedpolicy == SCHED_RR)
   printf("schedpolicy is SCHED_RR\n");
}
```

```
void show_inheritsched (int inheritsched)
{
 if (inheritsched == PTHREAD_INHERIT_SCHED)
   printf("inheritsched is PTHREAD_INHERIT_SCHED\n");
 else if (inheritsched == PTHREAD_EXPLICIT_SCHED)
   printf("inheritsched is PTHREAD_EXPLICIT_SCHED\n");
}

void show_contentionscope (int contentionscope)
{
 if (contentionscope == PTHREAD_SCOPE_SYSTEM)
   printf("contentionscope is PTHREAD_SCOPE_SYSTEM\n");
 else if (contentionscope == PTHREAD_SCOPE_PROCESS)
   printf("contentionscope is PTHREAD_SCOPE_PROCESS\n");
}

void show_stack (void *stackaddr, size_t stacksize)
{
 printf("stackaddr: %p\t stacksize: %lu\n",
         stackaddr, stacksize);
}

void show_guardsize (size_t guardsize)
{
 printf("guardsize: %lu\n", guardsize);
}
```

Managing Thread Attribute Objects

- **appendix_2 (cont.):**

Shows how to use the `pthread_attr_get*/set*` functions to query and set the different thread attributes. Shows default values unless the // code is uncommented.

```
int main ()
{
 pthread_attr_t attr;
 int ret, detachstate, schedpolicy, inheritsched, contentionscope;
 size_t stacksize_set; void *stackaddr_set;
 size_t stacksize_get; void *stackaddr_get;
 size_t guardsize;

 ret = pthread_attr_init(&attr);
 if (ret != 0) return ret;

 //ret = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
 //if (ret != 0) return ret;
 ret = pthread_attr_getdetachstate(&attr, &detachstate);
 if (ret != 0) return ret; else show_detachstate(detachstate);

 //ret = pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
 //if (ret != 0) return ret;
 ret = pthread_attr_getschedpolicy(&attr, &schedpolicy);
 if (ret != 0) return ret; else show_schedpolicy(schedpolicy);

 //ret = pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
 //if (ret != 0) return ret;
 ret = pthread_attr_getinheritsched(&attr, &inheritsched);
 if (ret != 0) return ret; else show_inheritsched(inheritsched);
```

Managing Thread Attribute Objects

- **appendix_2 (cont.):** Shows how to use the `pthread_attr_get*/set*` functions to query and set the different thread attributes. Shows default values unless the // code is uncommented.

```
 //ret = pthread_attr_setscope(&attr, PTHREAD_SCOPE_PROCESS);
 //if (ret != 0) return ret; // EINVAL/ENOSTUP error: unsupported
 ret = pthread_attr_getscope(&attr, &contentionscope);
 if (ret != 0) return ret; else show_contentionscope(contentionscope);

 //stacksize_set=PTHREAD_STACK_MIN;
 //stackaddr_set=(void*)malloc(stacksize_set);
 //ret = pthread_attr_setstack(&attr, stackaddr_set, stacksize_set);
 //if (ret != 0) return ret;
 ret = pthread_attr_getstack(&attr, &stackaddr_get, &stacksize_get);
 if (ret != 0) return ret; else show_stack(stackaddr_get, stacksize_get);

 //guardsize = 2*getpagesize();
 //ret = pthread_attr_setguardsize(&attr, guardsize);
 //if (ret != 0) return ret;
 ret = pthread_attr_getguardsize(&attr, &guardsize);
 if (ret != 0) return ret; else show_guardsize(guardsize);

 pthread_attr_destroy(&attr);
 //free(stackaddr_set);
 return 0;
}
```

Managing Mutex Attribute Objects

- **POSIX Mutex Attributes:**
  - **Type**: deadlocking, deadlock-detecting, recursive, …
    - `pthread_mutexattr_gettype, pthread_mutexattr_settype`
  - **Robustness**: what happens when a thread A tries to acquire a mutex and another thread B died possessing it (A deadlocks ? A acquires it ?)
    - `pthread_mutexattr_getrobust, pthread_mutexattr_setrobust`
  - **Process-shared**: for sharing a mutex across process boundaries
    - `pthread_mutexattr_getpshared, pthread_mutexattr_setpshared`
  - **Protocol**: how a thread behaves in terms of priority when a higher-priority thread wants the mutex (does it keep or change its priority?)
    - `pthread_mutexattr_getprotocol, pthread_mutexattr_setprotocol`
  - **Priority ceiling**: minimum priority at which the critical section will run; >= priority of all threads that may lock the mutex; avoids priority inversion
    - `pthread_mutexattr_getprioceiling, pthread_mutexattr_setprioceiling`

[https://stackoverflow.com/questions/4252005/what-is-the-attribute-of-a-pthread-mutex]

Managing Mutex Attribute Objects

- **Initializing Mutex Attribute Objects**
  - `int pthread_mutexattr_init(pthread_mutexattr_t *attr)`
    - initializes the mutex attribute object **`attr`** with **default** attribute values (note that for default values it is enough to pass `NULL` as **`attr`** in `pthread_mutex_init` or use the static initializer `PTHREAD_MUTEX_INITIALIZER`)
    - default attribute values:
      - **Type**: `PTHREAD_MUTEX_DEFAULT` (deadlocking)
      - **Robustness**: `PTHREAD_MUTEX_STALED` (deadlocking)
      - **Process-shared**: `PTHREAD_PROCESS_PRIVATE` (same process)
      - **Protocol**: `PTHREAD_PRIO_NONE` (keeps priority)
      - **Priority ceiling**: (implementation dependent)
- **Destroying Mutex Attribute Objects**
  - `int pthread_mutexattr_destroy(pthread_mutexattr_t *attr)`
    - destroys the **`attr`** object, which must not be reused until reinitialized

[https://stackoverflow.com/questions/29624365/what-is-the-default-mutex-attributes-of-the-pthread-mutex]

Managing Mutex Attribute Objects

- **appendix_3:**

Shows how to use the `pthread_mutexattr_get*/set*` functions to query and set the different mutex attributes. Shows default values unless the // code is uncommented.

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


void show_type (int type)
{
 if (type == PTHREAD_MUTEX_DEFAULT)
   printf("type is PTHREAD_MUTEX_DEFAULT\n");
 if (type == PTHREAD_MUTEX_NORMAL)
   printf("type is PTHREAD_MUTEX_NORMAL\n");
 if (type == PTHREAD_MUTEX_ERRORCHECK)
   printf("type is PTHREAD_MUTEX_ERRORCHECK\n");
 if (type == PTHREAD_MUTEX_RECURSIVE)
   printf("type is PTHREAD_MUTEX_RECURSIVE\n");
}

void show_robustness (int robust)
{
 if (robust == PTHREAD_MUTEX_STALLED)
   printf("robustness is PTHREAD_MUTEX_STALLED\n");
 else if (robust == PTHREAD_MUTEX_ROBUST)
   printf("robustness is PTHREAD_MUTEX_ROBUST\n");
}
```

```
void show_pshared (int pshared)
{
 if (pshared == PTHREAD_PROCESS_PRIVATE)
   printf("pshared is PTHREAD_PROCESS_PRIVATE\n");
 else if (pshared == PTHREAD_PROCESS_SHARED)
   printf("pshared is PTHREAD_PROCESS_SHARED\n");
}

void show_protocol (int protocol)
{
 if (protocol ==  PTHREAD_PRIO_NONE)
   printf("protocol is PTHREAD_PRIO_NONE\n");
 else if (protocol == PTHREAD_PRIO_INHERIT)
   printf("protocol is PTHREAD_PRIO_INHERIT\n");
 else if (protocol ==  PTHREAD_PRIO_PROTECT)
   printf("protocol is PTHREAD_PRIO_PROTECT\n");
}

void show_prioceiling (int prioceiling)
{
 printf("prioceiling is %d\n", prioceiling);
}
```

Managing Mutex Attribute Objects

- **appendix_3 (cont.):**

Shows how to use the `pthread_mutexattr_get*/set*` functions to query and set the different mutex attributes. Shows default values unless the // code is uncommented.

```
int main () {
 pthread_mutexattr_t attr;
 int ret, type, robust, pshared, protocol, prioceiling;

 ret = pthread_mutexattr_init(&attr);
 if (ret != 0) return ret;

 //ret = pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_ERRORCHECK);
 //if (ret != 0) return ret;
 ret = pthread_mutexattr_gettype(&attr, &type);
 if (ret != 0) return ret; else show_type(type);

 //ret = pthread_mutexattr_setrobust(&attr, PTHREAD_MUTEX_ROBUST);
 //if (ret != 0) return ret;
 ret = pthread_mutexattr_getrobust(&attr, &robust);
 if (ret != 0) return ret; else show_robustness(robust);

 //ret = pthread_mutexattr_setpshared(&attr, PTHREAD_PROCESS_SHARED);
 //if (ret != 0) return ret;
 ret = pthread_mutexattr_getpshared(&attr, &pshared);
 if (ret != 0) return ret; else show_pshared(pshared);

 //ret = pthread_mutexattr_setprotocol(&attr, PTHREAD_PRIO_INHERIT);
 //if (ret != 0) return ret;
 ret = pthread_mutexattr_getprotocol(&attr, &protocol);
 if (ret != 0) return ret; else show_protocol(protocol);

 //ret = pthread_mutexattr_setprioceiling(&attr, 2);
 //if (ret != 0) return ret;
 ret = pthread_mutexattr_getprioceiling(&attr, &prioceiling);
 if (ret != 0) return ret; else show_prioceiling(prioceiling);
```

```
 pthread_mutex_t mutex;
 ret = pthread_mutex_init(&mutex, &attr);
 if (ret != 0) return ret;
 else
  printf("pthread_mutex_init: SUCCESS\n");

 pthread_mutex_destroy(&mutex);
 pthread_mutexattr_destroy(&attr);
 return 0;
}
```

Managing Mutex Attribute Objects

- **Specifics of the mutex attribute Type in the Linux NPTL**
  - uses an alternative (equivalent) name: **Kind (= Type)**
    - in functions of previous slides consider **`kind`** instead of **`type`**
  - several kinds available; some map directly into POSIX types
  - **fast** kind (`PTHREAD_MUTEX_FAST_NP`)
    - also represented as `PTHREAD_MUTEX_TIMED_NP`
    - maps to POSIX type `PTHREAD_MUTEX_NORMAL/DEFAULT`
    - static initializer is = POSIX: `PTHREAD_MUTEX_INITIALIZER`
  - **error checking** kind (`PTHREAD_MUTEX_ERRORCHECK_NP`)
    - maps to POSIX type `PTHREAD_MUTEX_ERRORCHECK`
    - static initializer available, non-portable (no POSIX equivalent): `PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP`

Managing Mutex Attribute Objects

- **Specifics of the mutex attribute Type in the Linux NPTL**
  - **recursive** kind (`PTHREAD_MUTEX_RECURSIVE_NP`)
    - maps to POSIX type `PTHREAD_MUTEX_RECURSIVE`
    - static initializer available, non-portable (no POSIX equivalent): `PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP`
  - **adaptive** kind (`PTHREAD_MUTEX_ADAPTIVE_NP`)
    - doesn't map to any POSIX type
    - static initializer available, non-portable (no POSIX equivalent): `PTHREAD_ADAPTIVE_MUTEX_INITIALIZER_NP`
    - if a thread does not immediately acquire a lock, making it spin for a while before giving up and blocking the thread; the duration of the spinlock is adaptive, learning from previous attempts; for an explanation of the original programmer of this mutex kind see https://stackoverflow.com/questions/19863734/what-is-pthread-mutex-adaptive-np

Managing Mutex Attribute Objects

- **Specifics of the mutex attribute Type in the Linux NPTL**
  - **appendix_4:**

```
#include <stdio.h>
// to access NPTL non-portbable features
#define __USE_GNU
#include <pthread.h>

void show_type (int type)
{
 if (type == PTHREAD_MUTEX_FAST_NP)
   printf("type is PTHREAD_MUTEX_FAST_NP\n");
 if (type == PTHREAD_MUTEX_TIMED_NP)
   printf("type is PTHREAD_MUTEX_TIMED_NP\n");
 if (type == PTHREAD_MUTEX_ERRORCHECK_NP)
   printf("type is PTHREAD_MUTEX_ERRORCHECK_NP\n");
 if (type == PTHREAD_MUTEX_RECURSIVE_NP)
   printf("type is PTHREAD_MUTEX_RECURSIVE_NP\n");
 if (type == PTHREAD_MUTEX_ADAPTIVE_NP)
   printf("type is PTHREAD_MUTEX_ADAPTIVE_NP\n");
}

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;     // POSIX and NPTL
pthread_mutex_t mutex2 = PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP; // NPTL
pthread_mutex_t mutex3 = PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;  // NPTL
pthread_mutex_t mutex4 = PTHREAD_ADAPTIVE_MUTEX_INITIALIZER_NP;   // NPTL

int main ()
{
 pthread_mutexattr_t attr;
 int ret, type;
```

Managing Mutex Attribute Objects

- **Specifics of the mutex attribute Type in the Linux NPTL**
  - **appendix_4 (cont.)**

```
ret = pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_FAST_NP);
if (ret != 0) return ret;
ret = pthread_mutexattr_gettype(&attr, &type);
if (ret != 0) return ret; else show_type(type);

ret = pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_TIMED_NP);
if (ret != 0) return ret;
ret = pthread_mutexattr_gettype(&attr, &type);
if (ret != 0) return ret; else show_type(type);

ret = pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_ERRORCHECK_NP);
if (ret != 0) return ret;
ret = pthread_mutexattr_gettype(&attr, &type);
if (ret != 0) return ret; else show_type(type);

ret = pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE_NP);
if (ret != 0) return ret;
ret = pthread_mutexattr_gettype(&attr, &type);
if (ret != 0) return ret; else show_type(type);

ret = pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_ADAPTIVE_NP);
if (ret != 0) return ret;
ret = pthread_mutexattr_gettype(&attr, &type);
if (ret != 0) return ret; else show_type(type);
```

```
 pthread_mutex_t mutex;
 ret = pthread_mutex_init(&mutex, &attr);
 if (ret != 0) return ret;
 else
   printf("pthread_mutex_init: SUCCESS\n");

 pthread_mutex_destroy(&mutex);
 pthread_mutexattr_destroy(&attr);
 return 0;
}
```

Managing Condition Variables Attribute Objects

- **POSIX Condition Variables Attributes:**
  - **Clock**: the clock ID of the clock that shall be used to measure the timeout service of `pthread_cond_timedwait` function
    - `pthread_condattr_getclock, pthread_condattr_setclock`
    - the default value of the clock attribute shall refer to the system clock
  - **Process-shared**: for sharing a condition variable across processes
    - `pthread_condattr_getpshared, pthread_condattr_setpshared`
    - recall that a mutex has a similar attribute; the mutex accompanying a condition variable should have the same value for this attribute

Managing Condition Variables Attribute Objects

- **Initializing Condition Variables Attribute Objects**
  - `int` **`pthread_condattr_init`**`(pthread_condattr_t *`**`attr`**`)`
    - initializes the condition variable attribute object **`attr`** with **default** values (note that for default values it is enough to pass `NULL` as **`attr`** in `pthread_cond_init` or use the static initializer `PTHREAD_COND_INITIALIZER`)
    - default attribute values:
      - **Clock**: (implementation dependent). Linux: clock id `0`
      - **Process-shared**: `PTHREAD_PROCESS_PRIVATE` (same process).

- **Destroying Condition Variables Attribute Objects**
  - `int` **`pthread_condattr_destroy`**`(pthread_condattr_t *`**`attr`**`)`
    - destroys the **`attr`** object, which must not be reused until reinitialized

Managing Condition Variables Attribute Objects

- **appendix_5:**

Shows how to use the `pthread_condattr_get* / set*` functions to query and set the different condition variable attributes. Shows default values unless // code is uncommented.

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

void show_clock_id (int clock_id)
{
 printf("clock_id is %d\n", clock_id);
}

void show_pshared (int pshared)
{
 if (pshared == PTHREAD_PROCESS_PRIVATE)
   printf("pshared is PTHREAD_PROCESS_PRIVATE\n");
 else if (pshared == PTHREAD_PROCESS_SHARED)
   printf("pshared is PTHREAD_PROCESS_SHARED\n");
}
```

```
int main ()
{
 pthread_condattr_t attr;
 int ret, clock_id, pshared;

 ret = pthread_condattr_init(&attr);
 if (ret != 0) return ret;

 //ret = pthread_condattr_setclock(&attr, 1);
 //if (ret != 0) return ret;
 ret = pthread_condattr_getclock(&attr, &clock_id);
 if (ret != 0) return ret; else show_clock_id(clock_id);

 //ret = pthread_condattr_setpshared(&attr,
                                     PTHREAD_PROCESS_SHARED);
 //if (ret != 0) return ret;
 ret = pthread_condattr_getpshared(&attr, &pshared);
 if (ret != 0) return ret; else show_pshared(pshared);

 pthread_cond_t cond;
 ret = pthread_cond_init(&cond, &attr);
 if (ret != 0) return ret;
 else
   printf("pthread_cond_init: SUCCESS\n");

 pthread_cond_destroy(&cond);
 pthread_condattr_destroy(&attr);
 return 0;
}
```

Managing Barriers Attribute Objects

- **POSIX Barriers Attributes:**
  - **Process-shared**: for sharing a barrier across processes
    - `pthread_barrierattr_getpshared, pthread_barrierattr_setpshared`
      - see example 13
- **Initializing Barrier Attribute Objects**
  - `int` **`pthread_barrierattr_init`**`(pthread_barrierattr_t *`**`attr`**`)`
    - initializes the barrier attribute object **`attr`** with **default** values (for default values is enough to pass `NULL` as **`attr`** in `pthread_barrier_init` or to use the static initializer `PTHREAD_BARRIER_INITIALIZER(count)`)
    - default attribute values:
      - **Process-shared**: `PTHREAD_PROCESS_PRIVATE` (same process)
- **Destroying Barrier Attribute Objects**
  - `int` **`pthread_barrierattr_destroy`**`(pthread_condattr_t *`**`attr`**`)`
    - destroys the **`attr`** object, which must not be reused until reinitialized

Managing Barriers Attribute Objects

- **appendix_6:**

Shows how to use the `pthread_barrierattr_get* / set*` functions to query and set the different barrier attributes. Shows default values unless // code is uncommented.

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

void show_pshared (int pshared)
{
 if (pshared == PTHREAD_PROCESS_PRIVATE)
   printf("pshared is PTHREAD_PROCESS_PRIVATE\n");
 else if (pshared == PTHREAD_PROCESS_SHARED)
   printf("pshared is PTHREAD_PROCESS_SHARED\n");
}
```

```
int main ()
{
 pthread_barrierattr_t attr;
 int ret, pshared;

 ret = pthread_barrierattr_init(&attr);
 if (ret != 0) return ret;

 //ret = pthread_barrierattr_setpshared(&attr,
                                        PTHREAD_PROCESS_SHARED);
 //if (ret != 0) return ret;
 ret = pthread_barrierattr_getpshared(&attr, &pshared);
 if (ret != 0) return ret; else show_pshared(pshared);

 pthread_barrier_t barrier;
 ret = pthread_barrier_init(&barrier, &attr, 1);
 if (ret != 0) return ret;
 else printf("pthread_barrier_init: SUCCESS\n");

 pthread_barrier_destroy(&barrier);
 pthread_barrierattr_destroy(&attr);
 return 0;
}
```