Thread Management

Agenda

Minor and/or optional Pthreads features

By end of today, you should have a near-complete overview of Pthreads

Warning: many features mentioned today are NOT implemented on EITHER Heineken or NetBSD

Topics

1. Initialization
2. Finalization
3. Thread-specific data
4. Scheduling

Mutex Initialization

A mutex has **state** — therefore, mutex must be initialized before use

Two methods:

1. Static
2. Dynamic

Static Mutex Initialization

Storage allocation:

```
pthread_mutex_t m;
```

Storage allocation and value initialization:

```
pthread_mutex_t m =
PTHREAD_MUTEX_INITIALIZER;
```

`PTHREAD_MUTEX_INITIALIZER` is an implementation-dependent struct

Heineken PTHREAD_MUTEX_INITIALIZER

Heineken /usr/include/pthread.h:

```
#define PTHREAD_MUTEX_INITIALIZER \
  {0, 0, 0, PTHREAD_MUTEX_TIMED_NP, __LOCK_INITIALIZER}
```

NetBSD PTHREAD_MUTEX_INITIALIZER

NetBSD /usr/include/pthread.h:

```
#include <pthread_types.h>
...
#define PTHREAD_COND_INITIALIZER   _PTHREAD_COND_INITIALIZER
```

NetBSD /usr/include/pthread_types.h:

```
#define _PTHREAD_MUTEX_INITIALIZER { _PT_MUTEX_MAGIC,      \
                                    __SIMPLELOCK_UNLOCKED, \
                                    __SIMPLELOCK_UNLOCKED, \
                                    NULL,                  \
                                    {NULL, NULL},          \
                                    NULL                   \
                                  }
```

Dynamic Mutex Initialization

Storage allocation and value initialization:

```
pthread_mutex_t m;
        ...
int ret = pthread_mutex_init(&m, NULL);
```

Second arg is a "`pthread_mutex_attr_t *`"

Specifying Mutex Attributes

1. Create `pthread_mutex_attr_t *` “object” by calling `pthread_mutexattr_init()`

2. Get/set values of “foo” attribute with calls to `pthread_mutexattr_getfoo()` and `pthread_mutexattr_setfoo()`

3. Provide `pthread_mutex_attr_t *` object as 2nd arg in call to `pthread_mutex_init()`

4. Later destroy `pthread_mutex_attr_t *` object by calling `pthread_mutexattr_destroy()`

Mutex Finalization

If statically allocated: do nothing

Statically allocated storage is automatically reclaimed:

- Local (i.e., on stack)—when function returns
- Global—when address space destroyed (e.g., `exit(3)`) or overwritten (e.g., `exec(2)`)

If dynamically allocated: call `pthread_mutex_destroy()`

Mutex Initialization Gotcha

Mutex must be initialized EXACTLY ONCE

- If initialized more than once: later initializations overwrite mutex state—obviously erroneous
- If NOT initialized: unfortunately sometimes works!

  (If implementation would initialize with all/mostly zeroes, and if allocated storage is all zeroes, uninitialized mutex can work!)

Mutex Attributes

Recall that `pthread_mutex_init` takes attribute argument of type `pthread_mutexattr_t`

`pthread_mutexattr_t` has “no mandatory attributes”

Aside: The Wild, Wacky World of POSIX Threads

1. There have been several official versions of the POSIX Thread standard (i.e., Pthreads), plus many "drafts," plus "real time extensions"

2. Later versions specify increasingly many optional features

3. Different implementors choose to implement different subsets of optional features

Bottom line:

- Must learn what your particular implementation supports
- For portability avoid optional features, esp. if they are not commonly implemented

Back to Mutex Attributes

Optional mutex attributes:

1. Type
2. Process-shared
3. Priority ceiling
4. Priority inheritance

Discuss priority ceiling/inheritance attributes later (with scheduling)

Mutex Type Attribute Values

`PTHREAD_MUTEX_NORMAL`

`PTHREAD_MUTEX_ERRORCHECK` — expanded error checking

`PTHREAD_MUTEX_RECURSIVE` — no deadlock if thread re-locks mutex it already has locked (number of unlocks must match number of locks)

`PTHREAD_MUTEX_DEFAULT` — set to `PTHREAD_MUTEX_NORMAL`

Get/set value of type attribute with `pthread_mutexattr_gettype()` and `pthread_mutexattr_settype()`

Mutex Process-Shared Attribute Values

`PTHREAD_PROCESS_SHARED` — mutex may be shared by separate processes

`PTHREAD_PROCESS_PRIVATE`

Get/set value of process-shared attribute with `pthread_mutexattr_getpshared()` and `pthread_mutexattr_setpshared()`

Process-Shared Mutex

These calls and above constants available only if compile-time constant `_POSIX_THREAD_PROCESS_SHARED` is defined

E.g.,

```
#ifdef _POSIX_THREAD_PROCESS_SHARED
    ... access process-shared mutex attribute ...
#endif
```

Process-shared mutex must be placed in shared memory segment (i.e., segment created using `shmget(2)`)

Condition Variable Initialization

Analogous to mutex ...

- Condition variable may be statically or dynamically allocated/initialized
- Initialization must happen exactly once
- `pthread_cond_init()` takes an attribute argument of type "`pthread_condattr_t *`"
- `pthread_condattr_t` object created using `pthread_condattr_init()`
- Values of individual attributes within `pthread_condattr_t` object set using functions `pthread_condattr_set*()`

The Only Condition Variable Attribute

If compile-time constant `_POSIX_THREAD_PROCESS_SHARED` is defined ...

Then condition variable's process-shared attribute can be get/set using `pthread_condattr_getpshared()` and `pthread_condattr_setpshared()`

Your Mileage May Vary

Remember: not all this stuff is implemented on either Heineken or NetBSD

A web site with complete overview of Pthreads attributes:
`http://cs.pub.ro/~apc/2003/resources/pthreads/uguide/document.htm`

Exactly-Once Initialization

Two basic approaches to initializing a module foo:

1. Provide `foo_init()` function that user must call

2. "Self initialization"

   User need NOT call an init function

   Instead, every interface function—`foo_a()`, `foo_b()`, `foo_c()`, etc.—initializes module if it is the first function to be called

Initialization Function

Very common (esp. in C programs) to see `foo_init` function

Nevertheless, considered poor design ...

1. User shouldn't have to initialize module—initialization is an implementation issue, not an interface/use issue

2. User may forget to call init function

3. Because user may forget, well designed software would check for initialization on every call anyway, e.g.,

```
void foo_a()
{
if (!initialized)
    initialize();
    ...
}
```

Self Initialization

Above was example of self initialization

Every interface function begins with initialization test:

```
void foo_a()
{
if (!initialized)
    initialize();
    ...
}
```

What happens if self initializing module is called simultaneously from several threads?

Danger of Concurrent Self Initialization

Consider:

1. Thread A reads `initialized` variable, concludes module is not initialized
2. Thread A de-scheduled, thread B scheduled
3. Thread B reads `initialized` variable, concludes module is not initialized
4. Thread B initializes
5. Thread B de-scheduled, thread A scheduled
6. Thread A initializes

Module has now been initialized twice

Second initialization could occur after much valuable state has been created

Pthread_once

Pthreads has mechanism to guarantee that a function executes EXACTLY ONCE

Intended use: init function of thread-safe self-initializing module

Function `pthread_once` called with “once block” argument

Using pthread_once

1. Declare once block variable
(type `pthread_once_t`)

2. Statically initialize once block:

```
pthread_once_t ob = PTHREAD_ONCE_INIT;
```

3. Call

```
pthread_once(&ob, func);
```

where "`func`" is no-arg function that should be executed only once

Semantics of pthread_once

I. After the first return of `pthread_once()`, function `func` has been executed

II. No matter how many additional times `pthread_once()` is called—simultaneously or in sequence—function `func` will not execute again

Rules of Use For pthread_once

1. `func` takes no arguments

2. `func` must not be called directly

3. It is OK to have many once-block/`func` pairs in a program, but each `func` must be associated 1-to-1 with a once block

To repeat: the intention is that `func` shoud be the init function of a thread-safe self-initializing module

Initialization Summary

Mutexes and condition variables can be initialized statically or dynamically

Dynamically initialized mutexes and condition variables may have attributes that can be assigned values

Available attributes and values vary across implementations

Use `pthread_once()` to call no-arg init function of self-initializing thread-safe module

Finalization

Recall:

- A thread may be *canceled*, i.e., killed
- Depending on its "cancelation state," a canceled thread may die immediately ("asynchronous" cancelation state) or later at a designed cancelation point ("deferred" cancelation state)

Asynchronous canceled thread may leave behind arbitrary state, i.e., a mess

Pthreads provides a mechanism to help you clean up the mess

Cleanup Push/Pop

`pthread_cleanup_push()` pushes a function argument onto a per-thread stack

`pthread_cleanup_pop()` pops from the calling thread's stack

Function accepts one "`void *`" argument

All functions popped and called when thread terminates

- Thread is canceled
- Thread calls `pthread_exit`

Example

```
int fd;
int old_state, old_type;
void *arg;

/* about to do action (file open) that will require cleanup
 * therefore, temporarily disable cancelation */
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &old_state);

/* open file */
fd = open("foo.dat", ...);

/* push cleanup function and arg onto stack */
arg = malloc(sizeof(void *));
arg = (void *) &fd;
pthread_cleanup_push(close_file, arg);

/* now once again OK for thread to be canceled */
pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, &old_type)
pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &old_state);
```


Push/Pop Trivia, I

`pthread_cleanup_pop()` takes `int` argument:

- zero: do not execute popped function
- non-zero: do execute popped function

Push/Pop Trivia, II

POSIX "requires" one pop for each push within common lexical scope (i.e., between matching braces)

Reason: so that push/pop can be easily implemented as macros

The "requirement" is not enforced, though certain implementations might cause compilation error if push/pop not matched

E.g., push macro ends with "{" while pop macro begins with "}"—if no pop to match earlier push, will not compile

Thread-specific Data

How to maintain long-lived per-thread data?

1. Store on the stack—pass data to each function called by thread

   Inelegant: adds unnecessary argument to possibly many functions

   Inefficient: arguments passed by value

2. Store data in global variable

   Either need to know number of threads in advance or need to dynamically create per-thread variable

Pthreads provide mechanism—called a "key"— for per-thread global storage

Key Creation

```
void *destructor(void *);
static pthread_key_t key;
pthread_key_create(&key, destructor);
```

`destructor()` will be called when:

- New value replaces old value associated with key, or
- Thread terminates

See below for `destructor`'s argument

Associating Value With Key

```
void *data;
data = (void *) malloc(...);
... assign value to data ...
pthread_setspecific(key, data);
```

Call to `pthread_setspecific` will ...

1. Call registered destructor if there was a previous value non-NULL associated with key

   Destructor's argument will be the previous value
2. Associate new value with key

Retrieving Value Associated With Key

```
void *data = pthread_getspecific(key);
```

What's Going On

Not as strange as it may seem!

- Mechanism is just a hash table

  Key is used to search hash table, find hash record

  Hash record contains data value

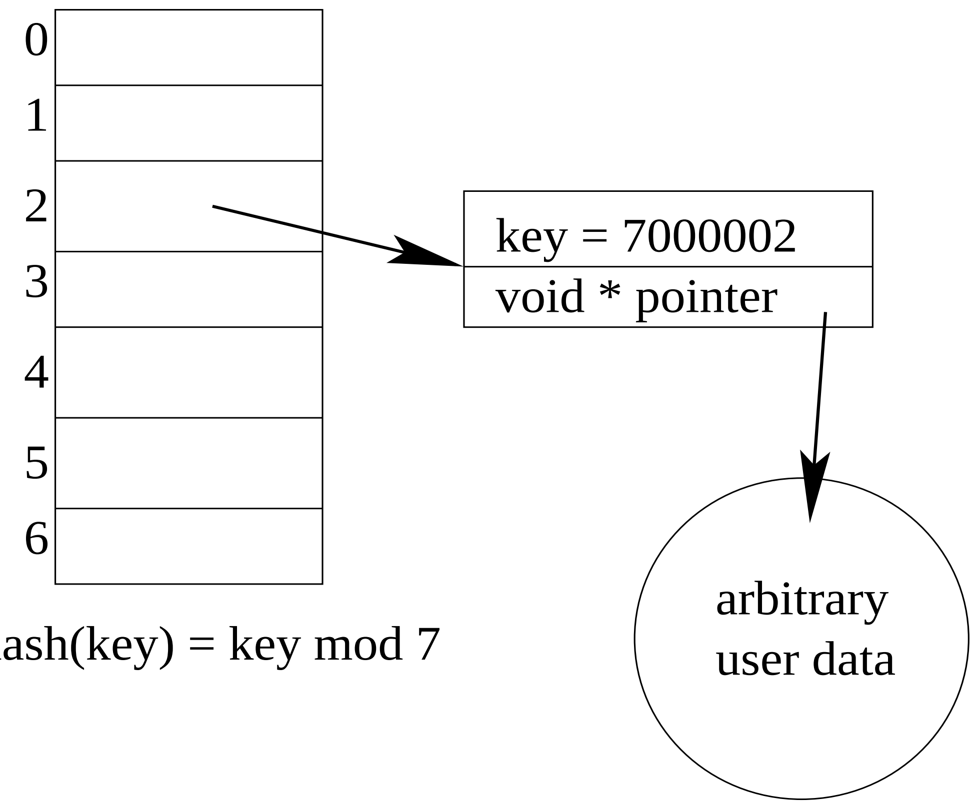
- Data value is "`void *`" so it can point to arbitrary amount of data

- Typical destructor is very simple:

  ```
  void *destructor(void *arg) {
      free(arg);
      return NULL;
  }
  ```

  `free()` appropriate when data value was created with `malloc()`

Depiction



hash(key) = key mod 7

Final Words About Keys

Yes, there is a `pthread_key_delete`

One thread may create any number of keys

Thread Scheduling

POSIX Pthreads committee wanted to accommodate:

- "Real time" applications

  Do not implement "hard" real time, just prevent uncontrolled preemption
- Multiprocessors

However, both scheduling considerations are OS-dependent and machine-dependent

Therefore, all Pthreads scheduling mechanisms (including "real time extensions") are optional

Scope, Allocation Domain, Priority

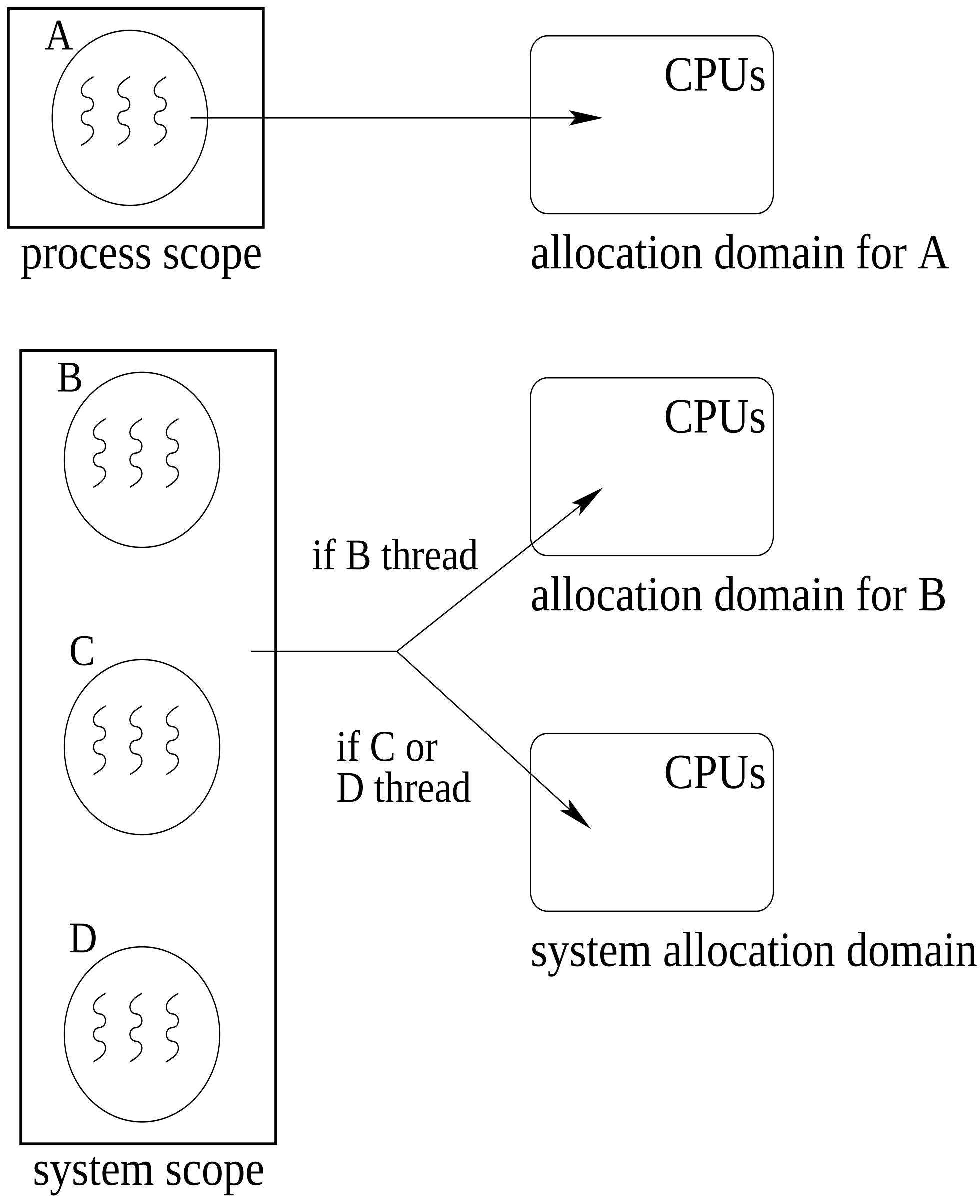
Three concepts key to scheduling:

1. **Scope**, aka **contention scope**, indicates which threads are in competition to be scheduled

2. An **allocation domain**, aka **processor set**, is set of processors available to service a given set of threads

3. Priority

Depiction



Threads in process A have process scope, others have system scope

Processes A and B have their own allocation domains

Pthreads and Scope

Scope is an attribute

A thread may have one of two scopes:

1. `PTHREAD_SCOPE_PROCESS` — thread competes against only other threads in same process
2. `PTHREAD_SCOPE_SYSTEM` — thread competes against all other "system" threads; i.e., threads not in the scope of some process

Use `pthread_attr_getscope()` and `pthread_attr_setscope()` to manipulate appropriate field in `pthread_attr_t` struct

Allocation Domain

Some multiprocessors (esp. those with many processors) can be "divided up" with sets of processors reserved for a set of threads

This allows *gang scheduling* — all relevant threads scheduled simultaneously

E.g., suppose a compute-bound application has 8 threads that communicate during their execution

- Want all 8 running simultaneously
- If even 1 is not running, whole program can slow down tremendously

An allocation domain is a set of processors (on multiprocessors that support such division)

Pthreads and Allocation Domains

Because the allocation domain concept is hardware-specific, Pthreads specifies no policies or interfaces for (1) creating allocation domain or (2) mapping contention scope to allocation domain

Some hardware+OS+Pthreads combinations implement non-portable interfaces

Priority

Priority implemented as small integer

The higher a thread's priority, the more likely thread is to be scheduled (or not be de-scheduled)

Support "soft" real time applications

Aside: Hard Real Time

Hard real time:

- OS API permits expression of quantitative time constraints

  E.g., "this thread must run every 60 seconds,"
  "thread X must get to this point in its execution 2 seconds before thread Y is scheduled"
- OS scheduler tries to accommodate threads' time constraints; failure to do so considered an error

Aside: Soft Real Time

Soft real time:

- OS API permits expression of relative importance

  E.g., "thread X is more important than thread Y but less important than thread Z"
- OS scheduler favors higher priority threads; no penalty for failure

Scheduling Algorithm, I

If compile-time constant `_POSIX_THREAD_PRIORITY_SCHEDULING` is defined then thread has two attributes:

1. Priority — thread's priority relative to other threads in same scope

2. Policy — how threads of *same priority* share access to CPU

Scheduling Algorithm, II

Scheduler maintains array of queues

One queue per priority level

(For this reason, priority range should not be too big; e.g., 0 to 31)

Each queue contains threads in “ready” state

Scheduling Algorithm, III

Priority scheduling: whenever a thread of higher priority within the same scope becomes ready, the current thread is preemptively de-scheduled and the higher priority thread is assigned to the CPU

(This assumes there are no unused CPUs in the allocation domain)

Scheduling Policies

There are three *policies*:

1. `SCHED_FIFO` — thread runs until it exits or blocks
2. `SCHED_RR` — thread runs until it exits/blocks, or until *time quantum* elapses, whichever comes first
3. `SCHED_OTHER` — implementation dependent

Notes About Scheduling Policies

1. When a once-blocked thread becomes unblocked (e.g., signaled by condition variable), it goes to end of FIFO at its priority level

2. Notice: within same priority, `SCHED_FIFO` threads are favored over `SCHED_RR` threads

3. `SCHED_OTHER` exists so that implementors can provide their system's standard algorithm

4. Default policy is `SCHED_OTHER`

SCHED_OTHER Policy

Most common `SCHED_OTHER` algorithm is UNIX-like “time sharing with priority adjustment:”

- Round-robin with time quantum, like `SCHED_RR`
- Threads that don’t use entire time quantum (i.e., they block) have priority increased by scheduler, to an extent

This policy favors I/O-bound (and synchronization-bound) threads over CPU-bound threads

How to Use Policy

Divide threads roughly into categories “time critical” and “not time critical”

Use `SCHED_FIFO` policy for time critical threads, use `SCHED_RR` policy for non time critical threads

Pthreads Priority/Policy API

Priority and policy are attributes

`pthread_attr_setschedpolicy()` to set policy

1st argument is `pthread_attr_t`,
2nd argument is `SCHED_FIFO`, `SCHED_RR`, or `SCHED_OTHER`

`pthread_attr_setschedparam()` to set priority

1st argument is `pthread_attr_t`,
2nd argument is of type `struct sched_param`, which contains one `int` field: `sched_priority`

One Last Attribute

Don't have to set priority/policy of every thread explicitly

Can specify that thread should "inherit" same priority/policy as thread that creates it:

```
pthread_t th_ID;
pthread_attr_t th_attr;
    ...
pthread_attr_init(&th_attr);
pthread_attr_setinheritsched(&th_attr,
                      PTHREAD_INHERIT_SCHED);
    ...
pthread_create(&th_ID, &th_attr, ...);
```

Other value is `PTHREAD_EXPLICIT_SCHED`

Interaction Between Scheduling and Synchronization

Synchronization is a form of scheduling among a restricted set of threads—those accessing common data

OS scheduler schedules ALL threads according to some system-wide fairness policy

Threads that synchronize access to some shared data structure are scheduling THEMSELVES ONLY according to some private-to-them synchronization policy

Bad Interactions

Two classic bad interactions:

- **Priority inversion**
- **Convoy phenomenon**

These phenomena arise from harmful interaction between scheduler's decisions and how threads "schedule" access to shared data

Won't discuss convoy phenomenon

Priority Inversion: The Problem

Occurs in environments where threads have priorities and scheduler invariant is: “the highest priority ready thread always runs immediately”

Problem: low-priority thread holds lock & is de-scheduled while high-priority thread waits for lock

Priority Inversion: Example

Consider:

1. `low priority thread Z has lock`
2. `Z preempted by high priority thread X`
3. `X tries to get lock & blocks`
4. `medium priority thread Y runs before thread Z can run again`

Highest priority thread, X, could be *starved* out of lock if steady stream of threads Y come along to prevent Z from running again

High priority thread *runs* (until it exhausts its time quantum) but does not *make progress* — spends all its time spin-waiting for lock

What's Going On

Unintended & harmful interaction:

1. OS thread scheduler schedules ALL threads according to priority
2. Subset of threads schedule access to shared data among themselves according to which has lock

Two "scheduling policies" don't know about each other

Priority Inversion: The Solution

Simply preventing preemption inside critical section is unacceptable:

- Critical section could be quite long
- Violates OS scheduler policy that highest priority thread always runs next

Solution: **priority inheritance**

Thread in critical section inherits — for duration of critical section only! — priority of highest-priority blocked thread

Priority Inheritance

Low-priority thread in critical section inherits high priority so it can finish critical section ASAP

Higher priority thread IS blocked, but only for duration of one thread's critical section

Low-priority thread's priority level returns to low when it exits critical section

Requires implementations of `pthread_mutex_lock()` and `pthread_mutex_unlock()` that know about priority of calling threads

Pthreads and Priority Inheritance

Inheritance setting is a mutex attribute

Use `pthread_mutexattr_setprotocol` to affect appropriate field of `pthread_mutex_attr_t` struct

Attribute values are:

- `PTHREAD_PRIO_NONE` — no change-of-priority protocol
- `PTHREAD_PRIO_PROTECT` — inferior "priority ceiling protocol"
- `PTHREAD_PRIO_INHERIT` — priority inheritance

To determine what is implemented, test these compile-time symbols:

- `_POSIX_THREAD_PRIO_PROTECT`
- `_POSIX_THREAD_PRIO_INHERIT`