POSIX Threads

Copyright © 2007 K Computing

PThread Programming

- Creating threads
- Advantages/disadvantages of threads
- Thread creation
- Thread attributes
- Thread initialization
- Joining
- Terminating
- Comparing thread ids
- Scheduling
- Native Posix Threads

Copyright © 2007 K Computing

Threads Overview

- POSIX Threads represent a standard, lightweight concurrency mechanism
- Several implementations exist for Linux
  - GNU Standard C library
  - IBM Next Generation Threads
  - RTLinux
  - RTAI
  - Florida State University
- POSIX Threads are available on many Unix systems
- Threads are not necessarily designed for parallelism, just concurrency
- POSIX Threads contain convenient data sharing and numerous synchronization methods

Copyright © 2007 K Computing

Advantages of Threads

- Higher performance – make continuous use of the CPU
- More natural algorithms

Copyright © 2007 K Computing

Disadvantages of Threads

- Data dependency issues
- Greater difficulty in understanding a program
- Differences, sometimes subtle, from processes

Copyright © 2007 K Computing

PThread Creation

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

- **Return value is zero or the error code**
- **`thread` gets the thread_id of the newly created thread**
- **`attr` specifies special attributes for the newly created thread**
- **`start_routine` is the function called in the new thread. When this function returns the thread is destroyed.**
- **`arg` is passed to the subroutine as its one and only parameter.**

Copyright © 2007 K Computing

Thread Attributes

- Attributes for threads are set by filling in an attr structure and passing it to pthread_create.
- The attr struct must be initialized - which sets all of the default values, and then can be modified via a set of attribute setting functions.
- Attribute blocks are only consulted on thread creation.
- Attributes settable:
  - detachstate
  - schedpolicy
  - schedparam
  - inheritsched
  - scope
- Functions have names of the form pthread_attr_set*. (e.g., pthread_attr_setsdetatchstate)

Copyright © 2007 K Computing

Initialization

- PThreads supply a convenient mechanism to make sure that initialization is done only once, and in the beginning when the threads first need it to be done.
- Use a pthread_once_t data item initialized to PTHREAD_ONCE_INIT and the function pthread_once.

```
#include <pthread.h>
static pthread_once_t cookie = PTHREAD_ONCE_INIT;
void init_it()
{// initialize data here}
void * mythread(void *p)
{pthread_once(&cookie, init_it);}
int main ()
{pthread_t tid;
pthread_create(&tid, NULL, mythread, NULL);
}
```

Copyright © 2007 K Computing

Joining

```
int pthread_join(pthread_t th, void **thread_return);
```

- pthread_join() suspends the calling thread until the referenced thread terminates.
- If the thread had already terminated then the join returns immediately
- Any sibling thread can join another.
- thread_return is used to obtain the return value - a pointer to a pointer.
- Call pthread_join only once for each thread.
- pthread_join "cleans up" resources - should be called for each created thread.
- Not needed for detached threads.

Copyright © 2007 K Computing

Terminating

- A thread can terminate another thread by "canceling" it.
- Threads can specify when they are cancelable.
- Threads can specify a list of functions to be executed if a thread is canceled.
- Threads can defer cancellation or allow it to be asynchronous
- Cancellation is meant to provide a convenient means to terminate threads cleanly that are no longer needed

Copyright © 2007 K Computing

Comparing

- Pthread ID's are meant to be opaque
- Pthread ID's cannot be simply compared
- The only function defined to compare Pthread ID's is
  - int pthread_equal(pthread_t thread1, pthread_t thread2);
  - pthread_equal returns 0 if thread1 and thread2 refer to the same thread, otherwise non-zero is returned
- A thread can obtain its own ID with
  - pthread_t pthread_self(void);
  - pthread_self() returns the thread ID of the calling thread

Copyright © 2007 K Computing

Scheduling

- Threads are scheduled with the POSIX sched_ functions just like processes.
- Threads support SCHED_OTHER, SCHED_FIFO, and SCHED_RR.
- Use sched_get_priority_min(int policy) and sched_get_priority_max(int policy) to determine minimum and maximum allowed priority values, respectively.
- There is no guarantee that SCHED_FIFO and SCHED_RR use the same range. (e.g., a high FIFO may run before a low RR or vice versa).

Copyright © 2007 K Computing

Signals With Threads

- Threads are not the same as processes and signals were meant to be used with processes.
- Threads can get, block or unblock signals with pthread_sigmask()
- pthread_sigmask() can also be used to fetch the current signal mask.
- To send a signal to a thread use pthread_kill()
- Some implementations may terminate all of the threads in a process when one of the threads receives a signal.
- Most pthread functions, e.g., pthread_mutex_unlock(), can not safely be called from signal handlers.
- Have a special thread do a sigwait() instead of having a signal handler function.

Copyright © 2007 K Computing

Native POSIX Thread Library

- Efficient 1-1 implementation: One kernel thread per one pthread.
- Takes advantage of newer kernel features:
  - O(1) scheduler - it is efficient to have thousands of threads
  - set_thread_area() system call to provide an efficient means to store local thread data
  - makes use of futex which is a fast synchronization method
- Is not quite POSIX compliant
  - waking not done by priority
  - some other calls, e.g., setuid(), setgid(), and nice() do not behave properly.

See: http://people.redhat.com/drepper/nptl-design.pdf

http://64.233.167.104/search?q=cache:lNmLNdILrokJ:www.ibm.com/developerworks/linux/library/l-inside.html+tls+%22system+call%22+linux&hl=en

Copyright © 2007 K Computing

Checking The Implementation

- Sysconf() with a variety of option names can be used to query about several pthread related values.
  - `_SC_THREADS` - are threads supported?
  - `_SC_THREAD_STACK_MIN` - minimum limit on stack size
  - `_SC_THREAD_PRIORITY_SCHEDULING`
  - `_SC_THREAD_THREADS_MAX`
- Environment variable `LD_ASSUME_KERNEL` can be set to an earlier kernel version to force *Linux Threads*.

Copyright © 2007 K Computing

Header file constants

- All pthread functions begin with `pthread_`
- All pthread constants begin with `PTHREAD_`
- Functions or constants that begin with these that are not standard should end with _np which means "*non-portable*."
- POSIX dictates a number of header file constants. E.g.,
  - `_POSIX_THREADS`
  - `_POSIX_THREAD_PROCESS_SHARED`

Copyright © 2007 K Computing

Debugging Pthreads

- Lots of printfs
- GDB
- Test single-threaded whenever possible
- Test with as few threads as possible if it can't be single threaded.
- Different on single and multi-cpu systems?

Copyright © 2007 K Computing

GDB Thread Features

- Thread command
- Info command
- Thread apply
- Notification of new threads
- Break ... thread

Copyright © 2007 K Computing

Some trickier things to consider:

- –Weakly ordered memory operations: different processors may not see variables in the same order. Some POSIX functions (e.g., pthread_mutex_lock) do synchronize memory.
- –Be careful of all the usual: race conditions, data dependencies ...
- –Linux is not POSIX compliant so there may be subtle differences.
- –Testing on one Linux system (kernel+library) does not, of course, mean that the behavior will be the same with different versions. Things have changed dramatically over time and may do so again.

Copyright © 2007 K Computing

Attachment

- Threads can be either “attached” or “detached”.
- Attached, the default, threads must be joined.
- Detached threads must not be joined.
- Detachment can be done while run or when created
  - `pthread_detach()`
- Create a detached thread by using an attribute to pthread_create.
  - pthread_attr_setdetachstate(attr_ptr,PTHREAD_CREATE_DETACHED)
- **Failure to join or detach is a resource leak.**

Copyright © 2007 K Computing

Synchronization

- Pthreads provide several means for synchronizing threads.
- As is usual with concurrent/parallel programming beware of:
  - –race conditions -> result depends upon relative speeds of threads
  - –data dependencies -> more than one thread can update a shared data item
  - –deadlock -> two or more threads are blocking waiting for each other
  - –priority inversion -> a lower priority thread is holding up a higher priority thread
- Pthreads provide:
  - –mutexes
  - –condition variables
  - –locks
  - –Semaphores (named and unnamed)
  - –barriers

Copyright © 2007 K Computing

Data Dependency Example

Global

sum = 0

Thread 1

.
.
.
sum ++
.
.
.

Thread 2

.
.
.
sum ++
.
.
.

What will be the value of sum when thread 1 and thread 2 finish?

Copyright © 2007 K Computing

Mutexes

- Initialization
- Locking and unlocking
- Mis-use
  - Not unlocking
  - Locking wrong one
  - Recursive locking
- Priority inheritance

Copyright © 2007 K Computing

More Mutexes

- **Mutex gets its name from *mut*ual *ex*clusion**
- **Mutexes are either locked or unlocked**
- **Locking and unlocking are thread-safe operations**
- **A thread attempting to lock a locked mutex will block**
- **Mutexes may support priority ceiling or priority inheritance.**

```
pthread_mutex_lock(&mymutex);

sum++;

pthread_mutex_unlock(&mymutex);
```

Copyright © 2007 K Computing

Condition variables

- Initialization
- Associated mutex
- Blocking
- Waking Up
  - Wakeup order
  - Broadcast
- Producer-Consumer

Copyright © 2007 K Computing

Read-Write Locks

- When shared data is read much more frequently than written a mutex can cause much unnecessary serialization.
- A read-write lock can be obtained by any number of concurrent reader threads.
- Obtaining the lock for writing is possible by only one thread at a time.
- read-write locks may be slower than mutexes and thus should be used only when required.

```
int pthread_rwlock_rdlock(&myrwlock);

int pthread_rwlock_wrlock(&myrwlock);
```

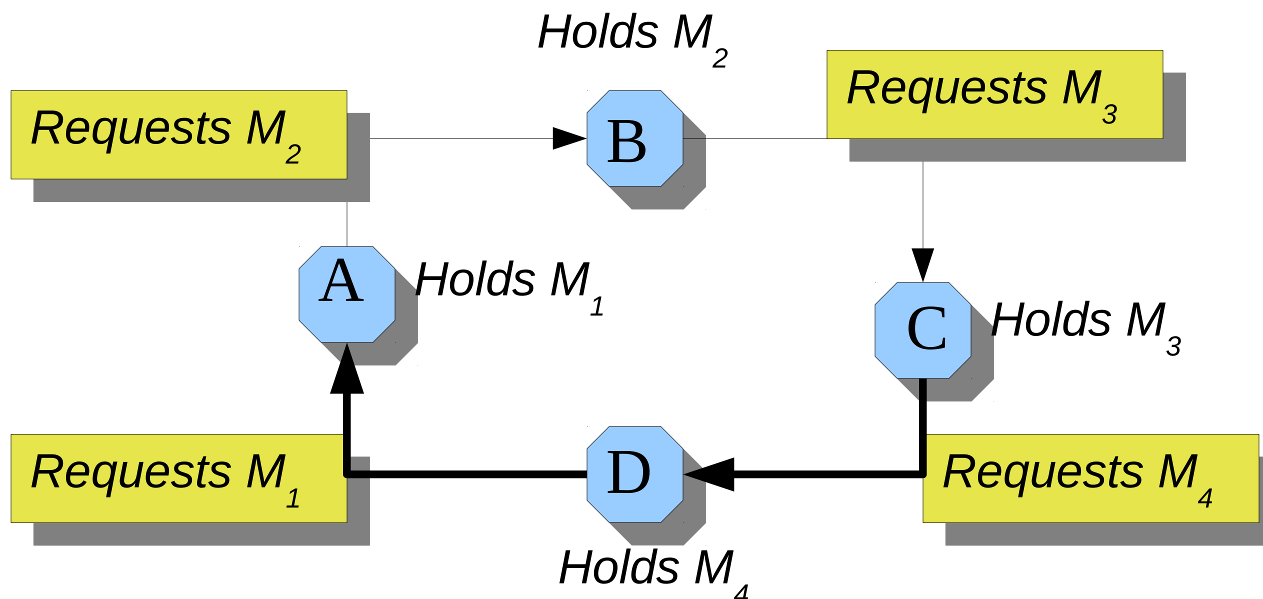
Copyright © 2007 K Computing

Semaphores

- Initialization: sem_init()
- Named/Unamed
- Blocking: sem_wait()
- Waking Up: sem_post()
  - Wake up order should be priority based – probably is not on Linux
- Don't copy semaphores – behavior when using a copy is undefined.
- Semaphore operations set errno and return 0 on success, -1 on failure.

Copyright © 2007 K Computing

- Two or more tasks holding a synchronization item and waiting for one held by another in a circular chain.
- Impossible to deadlock if tasks request the resource (e.g., a mutex or semaphore) in a specified order.
- Order may be their address (as long as the different tasks use the same address).
- Deadlocks may not be determinate or reproducible.



Copyright © 2007 K Computing

Priority inversion

- A higher priority task is postponed by a lower priority task.
- Occurs when a lower priority task holds a mutex that the higher priority task requests, yet, the lower priority task can't return it because the task is preempted.
- Priority inheritance would temporarily boost the priority of the lower priority task until it unlocks the mutex. Thus, the higher priority task can then acquire it.
- With *priority inheritance* the higher priority task is postponed by the length of time of the lower priority task's critical sections as opposed to the length of time of some middle priority tasks execution path.
- Note that priority inheritance is not possible for semaphores, for example, when it is not possible to determine which thread will increase the semaphore. Mutexes, not semaphores, support priority inheritance.

Copyright © 2007 K Computing

Lab Exercises

Copyright © 2007 K Computing