SHARCNET™

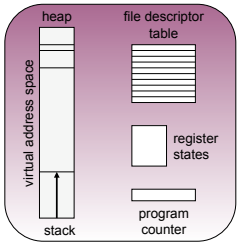
POSIX Threads

David McCaughan, *HPC Analyst*
SHARCNET, University of Guelph
dbm@sharcnet.ca

SHARCNET™

Processes

- DEF'N: a process is a program in execution, as seen by the operating system
- A *process* imposes a specific set of management responsibilities on the OS
  - a unique virtual address space (stack, heap, code)
  - file descriptor table
  - program counter
  - register states, etc.

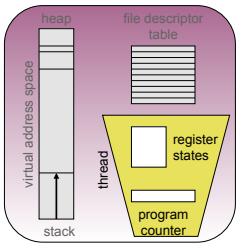


Conceptual View of a Process (simplified)

SHARCNET™

Threads

- DEF'N: a thread is a sequence of executable code *within* a process
- A serial process can be seen, at its simplest, as a single thread (a single "thread of control")
  - represented by the program counter
  - sequence (increment PC), iteration/ conditional branch (set value of PC)
- In terms of record-keeping, only a small subset of a process is relevant when considering a thread
  - register states; program counter



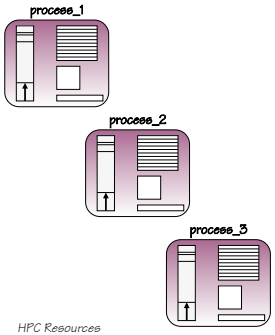
Conceptual View of a Thread (simplified)

SHARCNET™

Issues for Parallelization

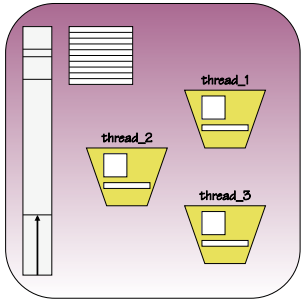
- What does this have to do with HPC exactly?
  - multiprocessor machines are now relatively common in the consumer market
  - multi-core machines are already available, and are set to become commodity computing resources in the immediate future
- How do you take advantage of multiple CPUs/cores in a single system image?
  - *multi-programming*:
    - start multiple processes to perform work simultaneously
  - *multi-threading*:
    - introduce multiple threads of control within a single process to perform work simultaneously

Multi-programming



- Distribute work by spawning multiple processes
  - e.g. fork(2), exec(2)
- Advantages
  - conceptually simple: each process is completely independent of others
  - process functionality can be highly cohesive
  - easily distributed
- Disadvantages
  - high resource cost
  - sharing file descriptors requires care and effort

Multi-threading



- Distribute work by defining multiple threads to do the work
  - e.g. pthreads
- Advantages
  - all process resources are implicitly shared (memory, file descriptors, etc.)
  - overhead incurred to manage multiple threads is relatively low
  - looks much like serial code
- Disadvantages
  - all data being implicitly shared creates a world of hammers, and your code is the thumb
  - exclusive access, contention, etc.

Pthreads

- There have been a number of threading models historically (some of which are still used)
  - mach threads, etc.
- Lack of standards led to IEEE to define POSIX 1003.1c standard for threading
  - POSIX threads, or Pthreads
- Note:
  - threads are peers
  - POSIX threads run in user space
    - contrast with what would be implied by kernel threads
    - trade-offs?

Threaded code on a Single Processor

- Multi-threading your code is not strictly an issue for multi-processor/core system
  - even on a single processor system, it is possible to interleave work and improve performance
    - efficient handling of asynchronous events
    - allow computation to occur during long I/O operations
    - permit fine grained scheduling of different operations performed by the same process
  - provides “upward compatibility”
    - threading a suitable piece of code does not break it on a single processor system, but has built-in potential for speed-up if multiple processors/cores are available

Pthreads vs. OpenMP

- OpenMP is a language extension for parallel programming in a SMP environment
  - allows the programmer to define "parallel regions" in code which are executed in separate threads (typically found around loops with no loop-carried dependencies)
  - the details of the thread creation are hidden from the user
- OpenMP is considered fairly easy to learn and use, so why bother with Pthreads at all?
  1. Right tool, right job: if OpenMP will service your needs you should be using it
  2. OpenMP supports parallelism in a very rigid sense and lacks versatility
     - Pthreads allows far more complex parallel approaches which would be difficult or impossible to implement in OpenMP

Pthreads Programming Basics

- Include Pthread header file
  - `#include "pthread.h"`
- Compile with Pthreads support/library
  - `cc -pthread …`
    - compiler vendors may differ in their usage to support pthreads (link a library, etc.)
    - GNU, Intel and Pathscale use the `-pthread` argument so this will suffice for our purposes
    - when in doubt, consult the man page for the compiler in question

Pthreads Programming Basics (cont.)

- Note that all processes have an implicit "main thread of control"
- We need a means of creating a new thread
  - `pthread_create()`
- We need a way to terminating a thread
  - threads are terminated implicitly when the function that was the entry point of the thread returns, or can be explicitly destroyed using `pthread_exit()`
- We may need to distinguish one thread from another at run-time
  - `pthread_self()`, `pthread_equal()`

pthread_create

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

- Create a new thread with a function as its entry point

- `thread`
  - handle (ID) for the created thread
- `attr`
  - attributes for the thread (if not NULL)
- `f_start`
  - pointer to the function that is to be called first in the new thread
- `arg`
  - the argument provided to f_start when called
  - consider: how would you provide multiple arguments to a thread?

pthread_self, pthread_equal

```
pthread_t pthread_self
(
    void
);
```

- Returns the handle of the calling thread
- This value can be saved for later use in identifying a thread

```
int pthread_equal
(
    pthread_t t1,
    pthread_t t2
);
```

- Compares two thread handles for equality
- e.g. conditional execution based on which thread is in a given section of code

pthread_exit

```
void pthread_exit
(
    void *status
);
```

- Terminate the calling thread and performs any necessary clean-up

- `status`
  - exit status of the thread
  - made available to any join with the terminated thread
- Note:
  - if `pthread_exit()` is not called explicity, the exit status of the thread is the return value of `f_start`

Synchronization

- There are several situations that arise where synchronization between threads is important
  - execution dependency
    - thread(s) must wait for other threads to complete their work before proceeding
  - mutual exclusion
    - a shared data structure must be protected from simultaneous modification by multiple threads
  - critical sections
    - a region of code must be executed by only one thread at a time
- Pthreads provides support for handling each of these situations

pthread_join

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

- Suspends execution of the current thread until the specified thread is complete

- `thread`
  - handle (ID) of the thread we are waiting on to finish
- `status`
  - value returned by `f_start`, or provided to `pthread_exit()` (if not NULL)

Example: *join.c*

```
…
int main()
{
    int i1 = 5, i2 = 2, r1, r2;
    pthread_t t1, t2;

    pthread_create(&t1, NULL, (void *)fact, (void *)&i1);
    pthread_create(&t2, NULL, (void *)fact, (void *)&i2);

    /*
     * resulting sum must wait for results
     */
    pthread_join(t1,(void **)&r1);
    pthread_join(t2,(void **)&r2);

    printf("Sum of fact = %d\n", r1 + r2);

    return(0);
}
…
```

Example: *join.c (cont.)*

```
…
int fact(int *n)
{
    int i, sum = 1;

    for (i = 1; i <= (*n); i++)
        sum *= i;

    return(sum);
}
```

Fundamental coding issues

- There are a couple of issues that tend to catch novice pthreads programmers (although not strictly pthread issues):
  - ***don't use a loop index directly as "thread id" passed to thread***
    - providing a pointer to the loop index variable as the thread argument will result in all threads having a pointer to the same integer
    - where this is necessary, use an array of integers
      - store loop index in position *i*, pass address of that array entry to thread
      - not usually inconvenient as it's likely you're storing the thread handles in an array as well
  - ***don't allow main thread to exit while child threads are still pending***
    - may appear that threads are doing nothing (or terminating early)
    - most implementations will terminate child threads when parent thread exits
    - use `pthread_join` to have `main()` wait for thread completion
- The following "Hello, world!" example illustrates both of these…

Example: "Hello, world!"

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

void output (void *);

int main(int argc, char *argv[])
{
    int id, nthreads = atoi(argv[1]);
    int id_array[nthreads];
    pthread_t thread[nthreads];

    /*
     * note: in order for a thread to receive an integer "id" which is
     * it's thread number, we can't pass &id to every one...as that would
     * be a pointer to the same variable and the number would change
     */
    for (id = 0; id < nthreads; id++)
    {
        id_array[id] = id;
        pthread_create(&thread[id], NULL, (void *)output, &id_array[id]);
    }
…
```

Example (cont.)

```
…
    for (id = 0; id < nthreads; id++)
        pthread_join(thread_array[id], NULL);

    return(0);
}

void output(void *thread_num)
{
    printf("Hello, world! from thread %d\n", *((int *)thread_num));
}
```

Exercise: Threaded "Hello, world!"

The purpose of this exercise is to allow you to work with simple thread operations, and begin to consider some of the issues that arise with basic pthread funcationality.

Exercise

1) The `thello.c` file in `~dbm/public/exercises/pthreads` is a copy of the one used in the earlier example

2) Modify this program so that a different string is output for each "Hello" message, in addition to the thread # that is already output
   - have the program invoked as "thello <nthreads> <str1> <str2> … <strN>", where nthreads is the number of threads to be created, and str1, str2, … strN is the string to be appended to "Hello, " in each thread
   - you will need to find a way to pass both the integer and string as you create each thread.

3) Modify the program further so that even numbered threads output "Hello, …", while the odd numbered threads output "Goodbye, …"
   - there is more than one way to accomplish this

Exercise (cont.)

- Compile and run this program with
  - 2 threads
  - 4 threads
  - 8 threads
- Answer the following questions:
  - *how did you accomplish passing multiple parameters to the thread? be critical of how you did this….is there a better way?*
  - *how did you get the threads distinguishing their behaviour based on even/odd thread number? what other ways can you think of to accomplish this? which way do you think would be more efficient, or is it even significant?*

Mutual Exclusion

- Consider a data structure such as a linked list
  - there are no problems with multiple threads reading the list simultaneously
  - what if addition/deletion of list elements is to occur in a thread?
    - need to control access to the list when it is modified - only one thread should be allowed to modify it at a time
    - *race conditions* abound in parallel code
- Pthreads allows you to declare mutex variables
  - mutex variables can be "locked" in order to control concurrent access to data or code by multiple threads
  - note that these are only advisory locks - if you use them appropriately it works; if you ignore your locks random bad things will happen

Mutual Exclusion (cont.)

- Need for mutex
  - shared data structure, buffers between pipeline stages, etc.
- Mutex variables are necessary, but are very restrictive
  - only lock what needs locking, and only for as long as required
  - Mutex tends to either be overused, or underused
    - what are the consequences of both?
- Thread Saftey
  - refers to a particular library or other collection of source code being *safe* to use with threads
    - consider that you are not *only* using code you wrote in a program
  - consider: *Is MPI Thread Safe?*

pthread_mutex_lock, pthread_mutex_unlock

```
int pthread_mutex_lock
(
    pthread_mutex_t *lock
);
```

```
int pthread_mutex_lock
(
    pthread_mutex_t *lock
);
```

- Obtain and release a lock on a mutex
  - note calling thread will block until it can obtain the lock
  - see also: `pthread_mutex_trylock()`
- Mutex variables can be initialized statically or dynamically
  - we only consider static initialization here
  - see `pthread_mutex_init()`
- CAUTION: potential for deadlock
  - two process block on a mutex waiting for the other

Example: *mutex.c*

```
…
pthread_mutex_t mutexvar = PTHREAD_MUTEX_INITIALIZER; /* static */
…

thread1()
{
    pthread_mutex_lock(mutexvar);
    access_shared_data();
    pthread_mutex_unlock;
}

thread2()
{
    pthread_mutex_lock(mutexvar);
    access_shared_data();
    pthread_mutex_unlock(mutexvar);
}

…
```

More Sophisticated Mutex issues

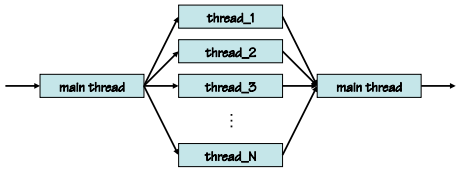
- Conditional Behaviour
  - Concurrent readers/single writer
    - it is usually acceptable to have multiple threads reading at one time, however only a single thread can have access when it is being modified
  - Counting semaphores
    - allow up to a given number of concurent accesses, but no more
    - need a counter such that when a thread requests a lock it increments a counter, with threads blocking only once the limit is reached
  - see `pthread_cond_wait()`, `pthread_cond_signal()`, `pthread_cond_broadcast()`, `pthread_rwlock_init()`, `pthread_rwlock_rdlock()`, `pthread_rwlock_wrlock()`, etc.
    - this is also how you would implement the thread pool model where all slave "sleep" until awoken by the master
- Initialization routines in threads
  - `pthread_once()`

Threading Models

- In principle, you can introduce arbitrary parallelism with multi-threading
- In practice there are only a few common models by which threaded parallelism is typically accomplished
  - fork/join
  - pipeline
  - master/slave
- Note that pthreads imposes no innate structure to parallel code
  - it is worthwhile to be familiar with how we implement these models

Fork/Join Threading

- Multiple threads are created, executing the same or different routines concurrently
  - also referred to as a Peer model
    - classic model of data parallelism
    - most of what OpenMP does for you is encapsulated here
  - synchronization upon thread exit is a common requirement

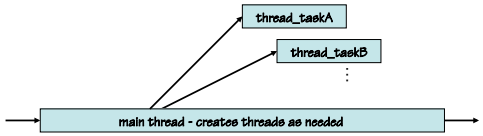


Fork/Join Implementation

```
    do_things();

    pthread_create(&thread_1, …, func_1);
    pthread_create(&thread_2, …, func_2);
    …

    pthread_join(…);

    do_more_things();
    …

func_1()
{
    do required work, synchronize with other threads as needed
}
func_2()
{
    do required work, synchronize with other threads as needed
}
…
```

Master/Slave Threading

- Threads are created on demand as the main thread requires work to be done
  - appropriate threads are created to service tasks, and run concurrently once created

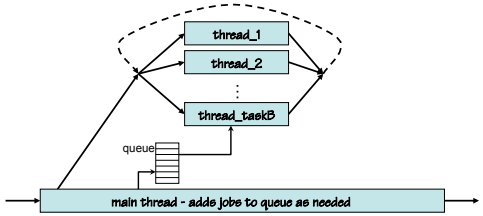


Master/Slave Implementation

```
    …
    while(1)
    {
        task_type = get_task();
        switch(task_type)
        {
            case A: pthread_create(…, do_task_A); break;
            case B: pthread_create(…, do_task_B); break;
            …
    }

do_job_A()
{
    perform job A, synchronize with other threads as needed
}
do_job_B()
{
    perform job B, synchronize with other threads as needed
}
…
```

Thread Pool

- A number of threads are created capable of doing work
  - the main thread adds jobs to a queue of work to be done and signals slave threads that there is work to be done
  - slave threads consume tasks from the shared queue and perform the work



Thread Pool Implementation

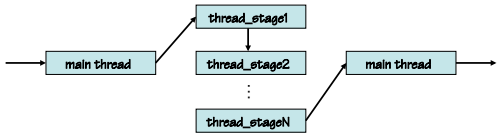
```
    …

    /*
     * note: common to create identical slaves in the thread pool
     */
    for (i = 0; i < NUM_SLAVES; i++)
        pthread_create(…, slave_thread);

    …

    while(1)
    {
        receive work
        add work to queue
        wake thread pool
    }

    …
```

Thread Pool Implementation (cont.)

```
…

slave_threads()
{
    while(1)
    {
        block until awoken by master
        task = next task from queue
        switch(task)
        {
            case A: do_task_A(); break;
            case B: do_task_B(); break;
            …
        }
    }
}
```

Pipeline Threading

- Multiple threads are created, each performing a discrete stage in some execution sequence
  - new input is fed to the first thread, and the output from each subsequent stage is passes to the next thread
  - after N time steps, the pipleline is filled and a result is produced every time step, with all stages processing concurrently
  - buffering and synchronization are significant here



Pipeline Implementation

```
    …
    pthread_create(…, do_stage1);
    pthread_create(…, do_state2);
    …
    pthread_join(do_stage1);
    pthread_join(so_stage2);
    …
    cleanup();
    …

do_stage1()
{
    while(!done)
    {
        get_input();
        do_stage1_work();
        pass_result_to_stage2();
    }
}
…
```

Pipeline Implementation

```
…

do_stage2()
{
    while(!done)
    {
        get_input_from_stag1();
        do_stage2_work();
        pass_result_to_stage3();
    }
}
…


/*
 * consider: what issues arise in coordinating "passing data"
 *           between stages in the pipeline
 */
```

SHARCNET™

Exercise: A Thread Safe Data Structure

The purpose of this exercise is to allow you to build a small threaded application to put some of the concepts we have applied here to use in practice.

SHARCNET™

Exercise

1) Pick a data structure that you know well (e.g. Linked List, matrix, etc.) and implement it in a thread safe way
   - consider where you store the mutex variable (could be part of the data structure, or initialized globally - it matters which way you do it (from either a software or pragmatic point of view)
   - strive to only have minimal critical sections protected
   - think about how multiple threads can possibly access this data structure and ensure you do not produce a deadlock

2) Implement a multi-threaded program to do large numbers of concurrent accesses to your data structure and test your implementation
   - design a testing situation that allows you to measure the efficiency of the parallel code in terms of overhead implied by the mutex operations

SHARCNET™

Food For Thought

- Efficiency issues
  - much of what we have discussed is straightforward enough to comprehend, but nothing in life (or programming) is free
  - what overhead is implied by critical sections in a given implementation?
- Debugging issues
  - threaded programming can introduce subtle bugs
  - not all debuggers are thread-aware
- Advanced issues
  - thread Safety and you
  - signal handling in threads

SHARCNET™

Food For Thought

3) Note that it is not necessary to strictly do the recursive doubling approach that we used for the example
   - think about how you are going to distribute the work and the data and ensure that process 0 outputs the result

- Answer the following questions:
  - *how did you choose to parallelize the work and data?*
  - *what sort of speed-up would you expect from the approach you have taken?*
  - *what would you have to take into account if you were going to distribute the data from process 0 (rather than have all processes read it)?*