Concurrent Programming with Pthreads

Clay Breshears
Rice University / ERDC MSRC
Henry A. Gabb
Nichols Research Corporation / ERDC MSRC

Acknowledgements

This work was funded by the DoD High Performance Computing Modernization Program US Army Engineer Research and Development Center Major Shared Resource Center (ERDC MSRC) through Programming Environment and Training (PET), Contract Number: DAHC 94-96-C0002, Nichols Research Corporation.

Outline

- What are Threads?
- Pthreads (Core functions)
  - Threading Serial Codes
- Concurrent Programming
- Numerical Computation
- Pthreads (Advanced functions)

Outline

- What are Threads?
- Pthreads (Core functions)
  - Threading Serial Codes
- Concurrent Programming
- Numerical Computation
- Pthreads (Advanced functions)

Multitasking in Unix

- Heavy-weight processes spawned with fork
  - Preemptive and priority-based scheduling
- Usually do not share common address space
  - Sharing data can be cumbersome
  - Synchronization is complicated by preemption
  - Resources can be exhausted quickly
  - Application doesn't scale well with load

Processes

- Unix process requires:
  - program code, program counter, heap memory, stack memory, stack pointer, file descriptors, virtual memory table, signal table, etc.
- Process embodies two characteristics
  - Unit of Resource Ownership
  - Unit of Dispatching

Processes & Threads

- Can these characteristics be dealt with separately?
  - Recent thought says 'YES'
- Unit of Resource Ownership is PROCESS
- Unit of Dispatching is THREAD

Threads

- Each thread is a separate execution stream
  - private program counter, stack memory, stack pointer, signal table
- Multiple threads may exist within a single process
- Sometimes called “lightweight processes”
- Other thread libraries:
  - Solaris threads, Linux threads, DCE threads, Win32 and OS/2 threads, GNU Portable threads

What is Pthreads?

- POSIX.1c standard
- C language interface
- All threads exist within same Unix process
- All threads are peers
  - No explicit parent-child model
  - Exception: “main thread” holds process information

Advantages of Multithreading

As opposed to multiple processes

- Resource use
- Shared memory
- Low overhead; context switch

Resource Use

- Threads share all resources of process
  - (virtual) memory,
  - files,
  - I/O channels, etc.
- Smaller memory usage for thread control structures
  - Fields default to process control structure

Shared Memory

- Not swapped out for thread context switch
- Threads read/write to shared variables for communication
  - cost of two memory accesses
- Process to process must go through external sharing mechanisms
  - system calls, sockets, network wires

Low Overhead

- Solaris process vs. Solaris thread:
  - 30 times longer to create a process
  - 10 time slower for synchronizing variables
  - 5 times slower for context switch

Pthreads vs. OpenMP

- Pthreads
  - any thread may create new threads
  - computations can be dynamically parallel
  - thus, nested parallelism is possible
- OpenMP
  - nested parallelism not yet supported
  - often implemented as high-level interface to Pthreads

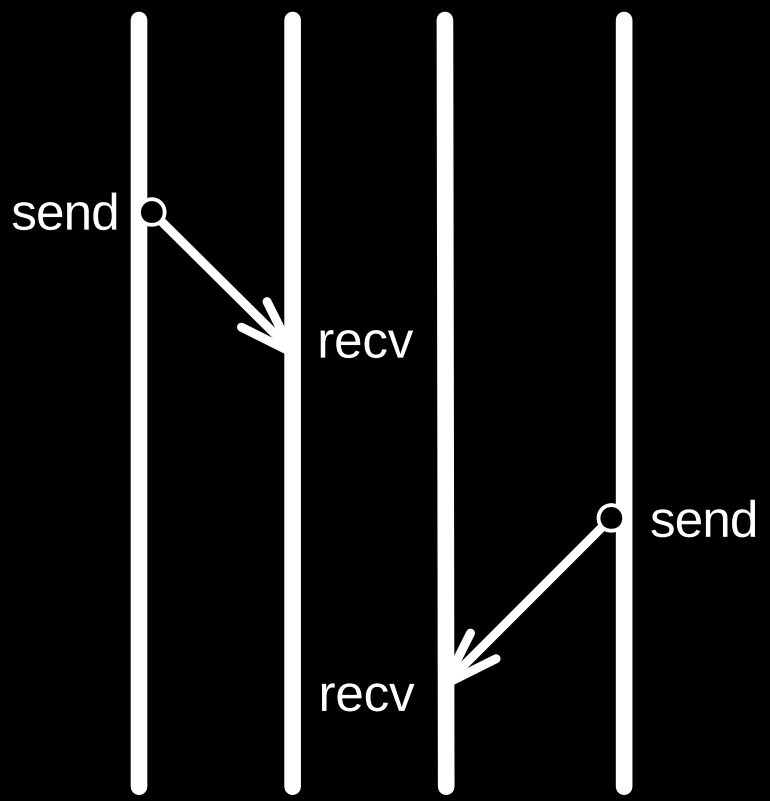
Parallelism or Concurrency?

- Parallelism: two or more threads are **executing** at the same time
  - multiple processors
- Concurrency: two or more threads are **in progress** at the same time
  - single or multiple processors
  - preemption of thread due to blocking or timeslice expiration

Thread Considerations

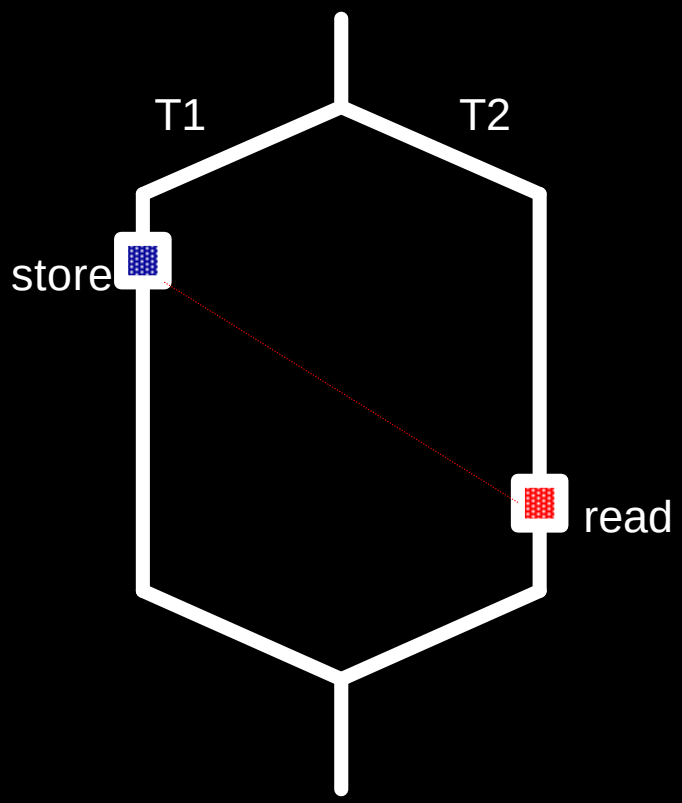
- Shared memory for communication
- Explicit synchronization needed
- Single processor model for development
- Race conditions
  - Read/Write, Write/Write conflicts
  - Models: Monitors, Rendezvous, Producer/Consumer, Readers/Writer

Parallel Communication



- Some parallel tasks wish to share data
- Messages become synchronization points between processes
  - implicitly synchronized

Thread Communication



- T1 stores value in global memory location
- T2 reads value out of global memory location
- Explicit synchronization needed to ensure read occurs after store

Race Condition

- Concurrent access of same variable by multiple threads
  - Read/Write conflict
  - Write/Write conflict
- Cause of other errors:
  - Execution order is assumed but is not guaranteed

Concurrent Execution

- Must consider all possible execution interleavings of thread operations
  - may be running on same processor
  - may be running on different processors
  - processors may have different speeds
- IF different output between separate runs:
  - THEN use some form of synchronization
  - may not show up 99 $^{44}/_{100}$% of the time

Concurrent Programming

- Concurrent programming requires skill
- "Standard" programming models available
  - Monitors
  - Rendezvous
  - Producer/Consumer
  - Readers/Writer

Traditional Thread Applications

- Operating Systems
- ATM (Cash Machine) Network
  - transactions at different locations are independent but act on shared data
- Database Search
  - threads can search different portions of data
  - each thread can satisfy a different query

Outline

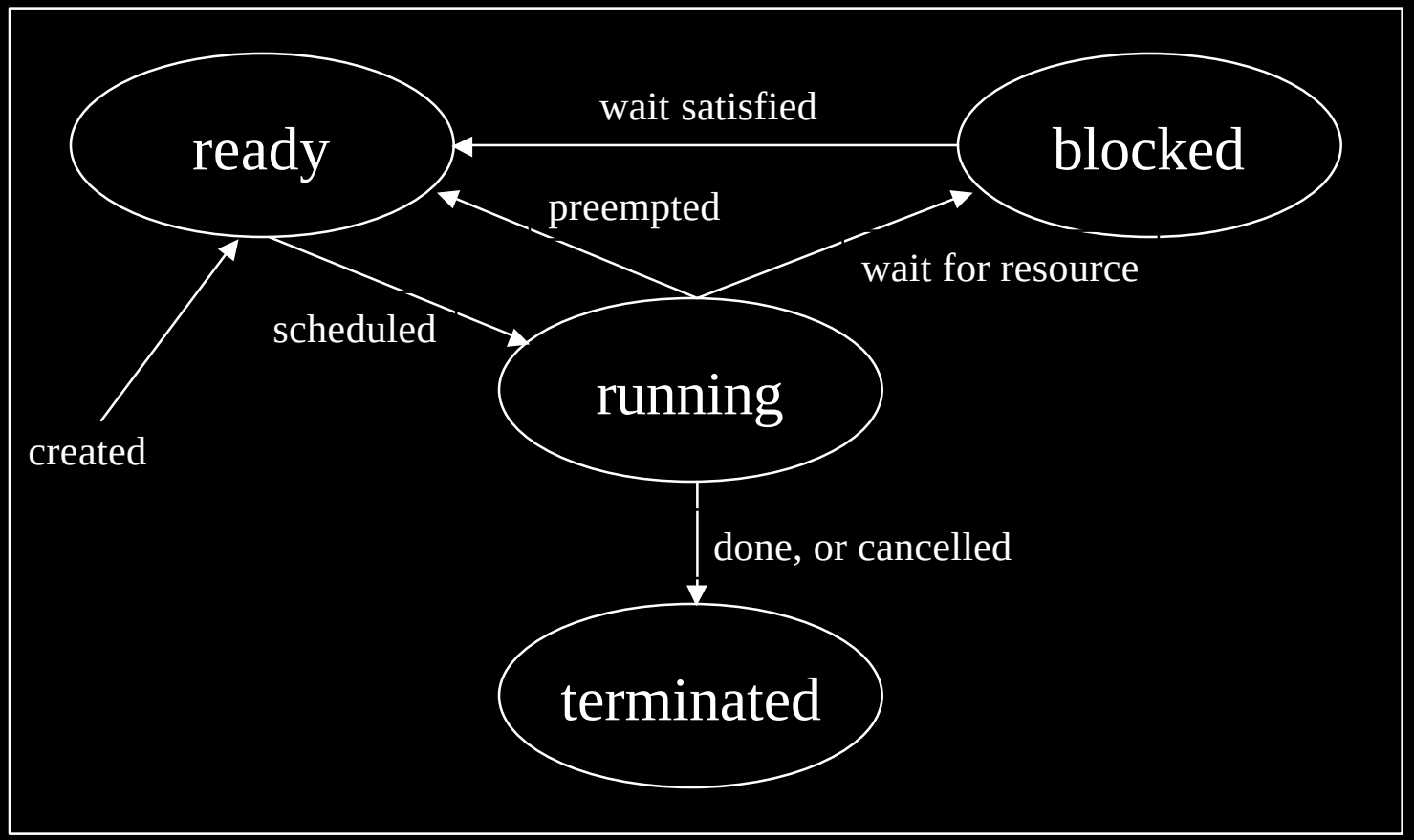
- What are Threads?
- Pthreads (Core functions)
  - Threading Serial Codes
- Concurrent Programming
- Numerical Computation
- Pthreads (Advanced functions)

Overview of Core Functions

- Thread Data Types
- Thread Management
- Mutual Exclusion
- Condition Variables
- Attributes and error codes

"The Life of a Thread"

taken from Programming with POSIX Threads, David R. Butenhof



Thread Management

- pthread_create
  - create a thread
  - start execution of function mapped to thread
- pthread_join
  - wait for thread to finish
  - retrieve exit code from joined thread
- pthread_detach
  - thread can no longer be joined or canceled
  - reclaim thread resources upon termination

Thread Management

- pthread_self
  - retrieve thread handle
- pthread_exit
  - halt execution of calling thread
  - report exit code

Thread Management Data Types

- pthread_t
  - thread handle
- pthread_attr_t
  - thread attributes:
    - detach state
    - stack size
    - stack address
    - specifying NULL gives default thread attributes

pthread_create

- int pthread_create(tid, attr, function, arg);

pthread_t *tid

handle of created thread

const pthread_attr_t *attr

attributes of thread to be created

void *(*function)(void *)

function to be mapped to thread

void *arg

single argument to function

pthread_create

- Spawn a thread running the function
- Thread handle returned via pthread_t structure
- Specify NULL to use default attributes
- Single argument sent to function
  - If no arguments to function, specify NULL
- Check error codes!

Error codes: pthread_create

- EAGAIN
  - insufficient resources to create thread
- EINVAL
  - invalid attribute

pthread_join

- int pthread_join(tid, val_ptr);

pthread_t tid

handle of thread to be joined

void **val_ptr

exit code reported by joined thread

pthread_join

- Calling thread waits for thread with handle **`tid`** to terminate
- Exit code is returned from joined thread if not NULL
- Threads are joinable by default

Error codes: pthread_join

- ESRCH
  - thread (pthread_t) not found
- EINVAL
  - thread (pthread_t) not joinable

Example: “Hello World”

```
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 4
main()
{
   pthread_t tid[NUM_THREADS];

   for (int i = 0; i < NUM_THREADS; i++)
       pthread_create(&tid[i], NULL, hello, NULL);

   for (int i = 0; i < NUM_THREADS; i++)
       pthread_join(&tid[i], NULL);
}
```

Example: “Hello World”

```
hello()
{
   printf(“Hello, World\n”);
}
```

pthread_detach

int pthread_detach(tid);

pthread_t tid

handle of thread to be detached

pthread_detach

- Detach thread **tid**
  - thread **tid** can no longer be joined or canceled
- Upon termination, detached thread resources are reclaimed by the system
- pthread_attr_{get|set}detachstate
  - PTHREAD_CREATE_JOINABLE
  - PTHREAD_CREATE_DETACHED

pthread_self

pthread_t pthread_self();

```
pthread_t tid;
int work, ierr;
if (work) {
   /* perform computation */
}
else {
   tid = pthread_self();
   ierr = pthread_detach( tid );   /* detach thread */
   /* begin independent processing */
}
```

pthread_exit

int pthread_exit(exitcode);

void *exitcode

value to be returned at join

- Terminates the calling thread
- Enables thread to report exit conditions to joining threads
- Allows main thread to exit without terminating process

Example: pthread_exit usage

```
#include <pthread.h>
#include <stdio.h>
#include <errno.h>
main () {
   int ierr;
   pthread_t tid;
   ierr = pthread_create(&tid, NULL, task, NULL);
   if (ierr != 0) {   /* failed to create thread */
      fprintf(stderr, “Error %d: %s\n”, ierr,
                                   strerror(ierr));
      pthread_exit( &ierr );
   }
}
```

Outline

- What are Threads?
- Pthreads (Core functions)
  - Threading Serial Codes
- Concurrent Programming
- Numerical Computation
- Pthreads (Advanced functions)

Getting Started

- Identify tasks for threading
- Identify computational model
  - Who does what?
  - Algorithms to use
- Identify how data will be accessed
  - What is global? What is local?
  - How is data to be assigned to threads?

Coding Considerations

- Create function to encapsulate computation
  - may be function that already exists
  - use single parameter (C structure for multiple arguments)
  - follow pthread_create template for types

Example: **`void *solve (void *arg)`**

Coding Considerations

- Recast parameter to local variable if needed
  - may be structure of several parameters
- Add code to determine task of thread
  - may need access to global variable
- Add code to access data for task
  - add code to protect global variables
  - shared data access may need to be restricted
- Add code to synchronize thread executions

Static Task Allocation

- Computations/Data are divided equally
  - based on number of threads and thread ID
- Typically access global data
  - must protect potential access overlaps
  - gather results into single location

Example: Numerical Integration

```
#define NUM_THREADS 1024
#define NUM_INTERVALS 65536
float p_sums[NUM_THREADS];
main()
{  pthread_t tid[NUM_THREADS];
   int i, t_num[NUM_THREADS];
   float sum = 0.0;
   for (i = 0; i < NUM_THREADS; i++){
        t_num[i] = i;
        pthread_create(&tid[i], NULL, do_calc, &t_num[i]);
   }
   for (i = 0; i < NUM_THREADS; i++){
        pthread_join(&tid[i], NULL);
        sum += p_sums[i];
   }
   printf("Sum = %f\n", sum);
}
```

Example: Numerical Integration

```
do_calc(void *num)
{  int i, h, myid, start, end;
   float lsum = 0.0, x;

   myid = (int)*num;
   h = 1.0 / NUM_INTERVALS;
   start = (NUM_INTERVALS / NUM_THREADS) * myid;
   end = start + (NUM_INTERVALS / NUM_THREADS);
   for (i = start; i < end; i++){
        x = h * ( (float)i - 0.5 );
        lsum += f(x);
   }
   p_sums[myid] = lsum;
}
float f(a) float a; { return ( 4.0 / (1.0 + a*a ) ); }
```

Dynamic Task Allocation

- Single thread "generates" tasks to be worked on [Boss thread]
- Other threads request new task when done with previous [Worker threads]
- Boss sends kill signal at end
- Workers terminate gracefully
- Good model for unequal amounts of computation between tasks

Outline

- What are Threads?
- Pthreads (Core functions)
  - Threading Serial Codes
- Concurrent Programming
- Numerical Computation
- Pthreads (Advanced functions)

Critical Sections

- Critical section
  - portions of program containing shared, modifiable data
  - data or code that must only be used or executed by a single thread at any time
- Example: Airline reservations

```
if (seat[num] == “EMPTY”) then
    seat[num] = customer_name
    print confirmation number
endif
```

Read Set / Write Set

- Read Set
  - those memory locations that a thread will access, but not modify
- Write Set
  - those memory locations that a thread will access and modify
- Applied to statements of code blocks

Access Conflicts

- Read/Write conflicts
  - Thread A reads from seat[57]
  - Thread B writes “Jones” to seat[57]
  - What value does Thread A get?
- Write/Write conflicts
  - Thread A writes “Smith” to seat[57]
  - Thread B writes “Jones” to seat[57]
  - What value is stored in seat[57]?

Race Conditions

- Concurrent access of same variable by multiple threads
  - Read/Write conflict
  - Write/Write conflict
- Execution order is assumed but cannot be guaranteed
- Most common error in concurrent programs
- May not be apparent at all times

Mutual Exclusion

- Enforces single thread access to a critical section
- Enables correct programming structures for avoiding race conditions
- Mechanism is a lock (mutex)
  - Atomic operations
  - Only one thread can "hold" mutex at any time
  - Lock/unlock is a paired operation

Pthread Mutual Exclusion

- pthread_mutex_init
  - initialize mutex variable
- pthread_mutex_lock
  - lock mutex if available, else wait for mutex
- pthread_mutex_unlock
  - return mutex to system or waiting thread
- pthread_mutex_destroy
  - destroy mutex, unavailable w/o initialization

Mutual Exclusion Data Types

- pthread_mutex_t
  - the mutex variable
- pthread_mutexattr_t
  - mutex attributes
    - process sharing
    - scheduling protocol
    - priority ceiling
    - specifying NULL gives default mutex attributes

pthread_mutex_init

int pthread_mutex_init(mutex, attr);

pthread_mutex_t *mutex

mutex to be initialized

const pthread_mutexattr_t *attr

attributes to be given to mutex

- Can also use the static, default initializer
  - PTHREAD_MUTEX_INITIALIZER
- Programmer must pay attention to mutex scope

Error codes: pthread_mutex_init

- ENOMEM
  - insufficient memory for mutex
- EAGAIN
  - insufficient resources (other than memory)
- EPERM
  - no privilege to perform operation

pthread_mutex_lock

- int pthread_mutex_lock(mutex);

  pthread_mutex_t *mutex

  mutex to attempt to lock

pthread_mutex_lock

- Lock mutex
- Mutex is held by calling thread until unlocked
- Mutex lock/unlock must be paired or deadlock occurs
- If mutex is locked by other thread, calling thread is blocked

Error codes: pthread_mutex_lock

- EINVAL
  - thread priority exceeds mutex priority ceiling
- EDEADLK
  - calling thread already owns mutex

pthread_mutex_unlock

int pthread_mutex_unlock(mutex);

pthread_mutex_t *mutex

mutex to be unlocked

pthread_mutex_destroy

- int pthread_mutex_destroy(mutex);

  pthread_mutex_t *mutex

  mutex to be uninitialized

- It is not necessary to destroy a statically initialized mutex

Example: Hello World

```
#include<pthread.h>
#define NUM_THREADS 4
pthread_mutex_t printlock = PTHREAD_MUTEX_INITIALIZER;
main()
{
   pthread_t tid[NUM_THREADS];

   for (int i = 0; i < NUM_THREADS; i++)
       pthread_create(&tid[i], NULL, hello, NULL);

   for (int i = 0; i < NUM_THREADS; i++)
       pthread_join(&tid[i], NULL);
}
```

Example: Hello World

```
hello()
{
   pthread_mutex_lock(&printlock);
    /* The printf function is not threadsafe */
   printf("Hello, World\n");
   pthread_mutex_unlock(&printlock);
}
```

Example: Numerical Integration

- Each thread updates a global variable
- The mutex variable has global scope
- The mutex lock/unlock protects critical section from write/write conflicts
- What might happen if **t_num** is replaced by the counter variable **i** in the main thread?

Example: Numerical Integration

```
#define NUM_THREADS 1024
#define NUM_INTERVALS 65536
float global_sum = 0.0;
pthread_mutex_t global_lock;
main()
{  pthread_t tid[NUM_THREADS];
   int i, t_num[NUM_THREADS];
   pthread_mutex_init(&global_lock, NULL);
   for (i = 0; i < NUM_THREADS; i++){
        t_num[i] = i;
        pthread_create(&tid[i], NULL, do_calc, &t_num[i]);
   }
   for (i = 0; i < NUM_THREADS; i++)
        pthread_join(&tid[i], NULL);
   printf("Sum = %f\n", global_sum);
}
```

Example: Numerical Integration

```
do_calc(void *num)
{  int i, myid, start, end;
   float x;

   myid = (int)*num;
   h = 1.0 / NUM_INTERVALS;
   start = (NUM_INTERVALS / NUM_THREADS) * myid;
   end = start + (NUM_INTERVALS / NUM_THREADS);
   for (i = start; i < end; i++){
        x = h * ( (float)i - 0.5 );
        pthread_mutex_lock(&global_lock);
        global_sum += f(x);
        pthread_mutex_unlock(&global_lock);
   }
}
float f(a) float a; { return ( 4.0 / (1.0 + a*a ) ); }
```

Condition Variables

- Semaphores (per Dijkstra)
  - integer variable (non-negative) with queue
  - wait(s): if s = 0 block, else s-- and proceed
  - signal(s): increment s, wake up thread waiting
- Condition variable is associated with an arbitrary conditional
- Provides mutual exclusion

Condition Variable and Mutex

- Mutex is associated with condition variable
  - protects evaluation of the conditional expression
- Prevents "Lost Signal" problem
  - no sleeping thread to catch signal
  - signal is not saved

Condition Variable Algorithm

- Acquire mutex
- While conditional is true, Wait
- Perform critical section computation
  - somehow update conditional
- Signal sleeping thread(s)
- Release mutex

Condition Variables

- pthread_cond_init, pthread_cond_destroy
  - initialize/destroy condition variable
- pthread_cond_wait
  - attempt to hold condition variable
- pthread_cond_signal
  - signal release of condition variable
- pthread_cond_broadcast
  - broadcast release of condition variable

Condition Data Types

- pthread_cond_t
  - the condition variable
- pthread_condattr_t
  - condition attributes
    - process sharing
    - specifying NULL gives default condition attributes

pthread_cond_init

int pthread_cond_init(cond, attr);

pthread_cond_t *cond

condition variable to be initialized

pthread_condattr_t *attr

condition variable attributes to be used

- Can also use the static, default initializer
  - PTHREAD_COND_INITIALIZER
- Programmer must pay attention to condition scope

Error codes: pthread_cond_init

- ENOMEM
  - insufficient memory for condition variable
- EAGAIN
  - insufficient resources (other than memory)

pthread_cond_destroy

- int pthread_cond_destroy(cond);

  pthread_cond_t *cond

  condition variable to be eliminated

- It is not necessary to destroy a statically initialized condition variable

pthread_cond_wait

- int pthread_cond_wait(cond, mutex);

pthread_cond_t *cond

condition variable attempted to be held

pthread_mutex_t *mutex

mutex associated with condition variable

pthread_cond_wait

- Releases associated mutex
- When signal is received, thread must reacquire mutex before function returns
- Prone to spurious wakeups, thus
  - Acquire mutex
  - Evaluate associated conditional expression
    - if true, block thread (release mutex) to await signal on condition variable
    - if false, release mutex and continue

pthread_cond_signal

int pthread_cond_signal(cond);

pthread_cond_t *cond

condition variable to be released

pthread_cond_signal

- Signal condition variable, wake one waiting thread
- If no threads waiting, no action taken
  - Signal is not saved for future threads
- Before signal, thread must have mutex
  - If not, race condition may result

Example: Denominator

- Two threads oversee a global variable
  - Thread 1 calculates a value
  - Thread 2 waits for this value
- A mutex controls access to this variable
- Thread 1 signals thread 2 (waiting)

Example: Denominator (thread1)

```
#include <pthread.h>

pthread_mutex_t denom_mtx = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t denom_cond = PTHREAD_COND_INITIALIZER;
float denominator = 0.0;

thread1() {
   pthread_mutex_lock( &denom_mtx );

   denominator = f();              /* calculate denominator */
   pthread_signal( &denom_cond );  /* signal waiting thread */

   pthread_mutex_unlock( &denom_mtx );
}
```

Example: Denominator (thread2)

```
thread2() {
   float local_denom;

   pthread_mutex_lock( &denom_mtx );

   /* wait for non-zero denominator */
   while( denominator == 0.0 )
      pthread_cond_wait( &denom_cond, &denom_mtx );
   local_denom = denominator;

   pthread_mutex_unlock( &denom_mtx );

   /* Use local copy of denominator for division */
}
```

pthread_cond_broadcast

- int pthread_cond_broadcast(cond);

  pthread_cond_t *cond

  condition variable to be released

pthread_cond_broadcast

- Wake all threads waiting on condition variable
- If no threads waiting, no action taken
  - Broadcast is not saved for future threads
- Before broadcast, thread must have mutex
  - If not, race condition may result

Example: Broadcast signal

- Main thread
  - Creates worker threads
  - Reads input data
  - Signals worker threads to begin computation
- Worker threads
  - Wait for signal that data is available
  - Begin work

Example: Main thread

```
#include <pthread.h>
pthread_t tid[N_THREADS];
pthread_mutex_t read_mtx = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t read_cond = PTHREAD_COND_INITIALIZER;
int ready = 0;

main() {
   for (int i = 0; i < N_THREADS; i++)
      pthread_create(&tid[i], NULL, worker, NULL);
   /* read input data */
   pthread_mutex_lock( &read_mtx );
   ready = 1;                          /* reset condition flag */
   pthread_broadcast( &read_cond );
   pthread_mutex_unlock( &read_mtx );
   pthread_exit();      /* exit without terminating process */
}
```

Example: Worker thread

```
worker() {
   pthread_mutex_lock( &read_mtx );

   /* wait until flag indicates that data is ready */
   while( ready == 0 )
      pthread_cond_wait( &read_cond, &read_mtx );

   pthread_mutex_unlock( &read_mtx );

   /* data is available, begin work */

   pthread_exit();   /* Exit when finished */
}
```