COMP605/CS605: Scientific Computing, Spring 2023
POSIX Threads or Pthreads

Pthreads specifies library to be linked with C programs. Pthreads API only available in POSIX systems.

1 Compilation and Execution

gcc -g -Wall -o ph_hello pth_hello.c -lpthread (tells the compiler to link to Pthreads library)

./pth_hello -lpthread < number of threads >

2 Hello program

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

int thread_count; /* global variable: accessible to all threads */

void* Hello(void* rank); /* Thread function */

int main(int argc, char* argv[])
{
    long thread; /* use long in case of a 64-bit system */
    pthread_t* thread_handles;

    thread_count = strtol(argv[1], NULL, 10); /* # of threads (command line) */

    thread_handles = malloc(thread_count*sizeof(pthread_t));

    for (thread = 0; thread < thread_count; thread++)
        pthread_create(&thread_handles[thread], NULL, Hello, (vpoid*) thread);

    printf("Hello from the main thread\n");

    for (thread = 0; thread < thread_count; thread++)
        pthread_join(thread_handles[thread], NULL);

    free(thread_handles);

    return 0;
}

void* Hello(void* rank)
{
    long my_rank = (long) rank; /* use long in case 64-bit system */

    printf("Hello from thread %ld of %d\n", my_rank, thread_count);

    return NULL;
}
```

3 Output

- To run the program with one thread:
  ./pth_hello.c -lpthread 1
  and the output will look something like this:

```
Hello from the main thread
Hello from thread 0 of 1
```

- To run the program with four threads:
  ./pth_hello.c -lpthread 4
  and the output will look something like this:

```
Hello from the main thread
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
```

4 Observations

- Just a C program.
- pthread.h header file which declares the various Pthreads functions, constants, types and so on.
- Global variable thread_count.
- In line 14, program gets the number of threads it should start from the command line.
- Convention about how to specify the number of threads.
- Function *strtol* converts a string into a *long int*.

```
long strtol( /* returns long int corresponding to number_p string */
           const char* number_p /* in */,
           char**      end_p /* out: NULL or 1st invalid number_p char */,
           int         base  /* in: base of number representation */);
```

- In line 16, allocate storage for opaque pthread_t data structure for thread-specific info.
- In lines 18-19, function to start the threads.

```
int ptreads_create( /* return value for most Pthreads functions indicate if
     there is an error in the function call (in general ignore it) */
    pthread_t* thread_p /* out: pointer to appropriate pthread_t object; not
        allocated by the call; it must be allocated before the call */
    const pthread_attr_t      attr_p /* in: NULL; not using it */,
    void* (*start_routine)(void*)  /* in: function thread is to run */,
    void* arg_p  /* in: pointer to arguments to be passed to function */);
```

- The function prototype that is started by *pthread_create* looks like: void* thread_function(void* args_p);
- The final argument in the call to *pthread_create* effectively assigns each thread a unique integer *rank*.
- Result of carrying casts is system-defined.

- The thread running main is the textitmain thread. After starting the threads it prints a message.
- In the meantime, the threads started by the calls to *pthreads_ create* are also running.
- The threads get their ranks by casting in line 33.
- When a thread is done, since the type of its function has a return value, the thread should return something.
- In Pthreads, the programmer does not directly control where the threads are run.
- In lines 23-24, the function *pthread_join* is called once per thread. A single call to *thread_join* will wait for the thread associated with the *pthread_t* object to complete

```
int ptreads_join( /* main thread call this for it to complete termination */
    pthread_t thread /* in */
    void** ret_val_p /* out: receive any return value computed by thread */
```

- User specifies the number of threads at command line. Main thread creates subsidiary threads. While threads are running, main thread prints a message, and then waits for the other threads to terminate.
- No error checking for simplicity.

5 Matrix-Vector Multiplication Example

$A = (a_{ij})$ a $m \times n$ matrix. $x = (x_0, x_1, \cdots, x_{n-1})^T$ then $y = Ax$ with $y = (y_0, y_1, \cdots, y_{m-1})^T$ given by

$$y_i = \sum_{j=0}^{n-1} a_{ij} x_j$$

Serial code

```
for (i = 0; i < m; i++)
{
    y[i] = 0.0;
    for (j = 0; j <n; j++)
        y[i] += A[i][j]*x[j];
}
```

Parallel code (at a minimum x shared, but in addition make A and y global)

```
void* Pth_mat_vect(void* rank)
{ /* it is assumed that A, x, y, m, and n are global and shared */
    long my_rank = (long) rank;
    int i, j;
    int local_m = m/thread_count;
    int my_first_row = my_rank*local_m;
    int my_last_row = (my_rank+1)*local_m - 1;

    for (i = my_first_row; i <= my_last_row; i++)
    {
        y[i] = 0.0;
        for (j = 0; j <n; j++)
            y[i] += A[i][j]*x[j];
    }

    return NULL;
} /* thread function that carries out matrix-vector multiplication */
```

6 Estimation of π (example of critical section)

Formula (needs a lot of terms to be accurate)

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + (-1)^n \frac{1}{2n+1} + \cdots\right).$$

Serial code

```
double factor = 1.0;
double sum = 0.0;

for (i = 0; i < n; i++, factor = -factor)
    sum += factor/(2*i+1);

pi = 4.0*sum;
```

Parallel code (divide iterations among thread_count threads; *sum* a shared variable; $n = \#$ of terms in sum)

```
void* Thread_sum(void* rank)
{
    long my_rank = (long) rank;
    double factor;
    long long i;
    long long my_n = n/thread_count;
    long long my_first_i = my_n*my_rank;
    long long my_last_i = my_first_i + my_n;

    if (my_first_i % 2 == 0)
        factor = 1.0;
    else
        factor = -1.0;

    for (i = my_first_i; i < my_last_i; i++, factor = -factor)
        sum += factor/(2*i+1);

    return NULL;
}
```

- While increasing n one can observe that when utilizing two threads the result gets worse.
- In fact each time it is run, the approximation for π changes.
- The addition of two values, $x = x + y$, is not a simple operation.
  1. x and y are stored in main memory, which has not circuitry for carrying out arithmetic operations.
  2. Before the addition is carried out, x and y are transferred from main memory to CPU registers.
  3. Once the values are in registers, the addition can be carried out.
  4. After the addition is completed, the result may have to be transferred back to main memory.
- Suppose there are two threads, and each computes a value that is stored in private variable y.
- Also, suppose one wants to add these private y's to a shared variable x (initialized to 0 by the main thread).
- If thread 1 copies x from memory to a register before thread 0 stores its result, the computation carried out by thread 0 will be overwritten by thread 1. Or, if thread 1 races ahead of thread 0 then its result may be overwritten by thread 0. Unless one of the threads stores its result before the other thread starts reading x from memory, the result of the faster thread will be overwritten by the result of the slower thread.

7 Busy-Waiting

Pthreads global sum with busy-waiting

1. Suppose 0 wants to execute a statement, it needs to make sure 1 is not already executing it;
2. once makes sure of this, it needs to provide some way for 1 to determine that; so 1 will not attempt starting execution until 0 is done;
3. when done, provides some way to 1 to determine that, so it can start executing the statement.

Simple approach: Use a flag

Suppose *flag* is a shared *int* variable, set to 0 by the main thread.

Suppose we have the next code

```
1 y = Compute(my_rank);
2 while (flag != my_rank);
3 x = x + y;
4 flag++;
```

- If thread 1 finishes line 1 before thread 0, what happens when it reaches line 2?
- Main thread had initialized *flag* to 0, thread 1 will not proceed to line 3 until thread 0 executes line 4.
- Thread 1 cannot enter the critical section until thread 0 has completed the execution of *flag++*.
- The *while* loop is an example of busy-waiting.
- A thread repeatedly tests a condition, but effectively, does no useful work until the condition has the appropriate value.
- If compiler optimization is turned on, it is possible that the compiler will make changes that will affect the correctness of busy-waiting (the compiler is unaware that the program is multithreaded, so it does not know that variables *flag* and x can be changed by another thread).

```
1 y = Compute(my_rank);
2 x = x + y;
3 while (flag != my_rank);
4 flag++;
```

- Turn off compiler optimizations completely? Other alternative solutions.
- Busy-waiting not an ideal solution: If thread 0 is delayed ... this can seriously degrade performance.
- In busy-waiting, one should precede a critical section with a busy-wait loop. However, when a thread is done with the critical section, if it simply increments *flag*, eventually *flag* will be greater than the number of threads, and none of the threads will be able to return to the critical section. That is, after executing the critical section once, all the threads will be stuck forever in the busy-wait loop.
- Rather than simply incrementing *flag*, the last thread should reset the *flag* to zero. This is accomplished with

```
1 y = Compute(my_rank);
2 x = x + y;
3 while (flag != my_rank);
4 flag = (flag + 1) % thread_count;
```

So, with this change, the estimation of π looks like

```
void* Thread_sum(void* rank)
{ /* shared flag variable */
    long my_rank = (long) rank;
    double factor;
    long long i;
    long long my_n = n/thread_count;
    long long my_first_i = my_n*my_rank;
    long long my_last_i = my_first_i + my_n;

    if (my_first_i % 2 == 0)
        factor = 1.0;
    else
        factor = -1.0;
    for (i = my_first_i; i < my_last_i; i++, factor = -factor)
    {
        while (flag != my_rank);
        sum += factor/(2*i+1);
        flag = (flag + 1) % thread_count;
    }
    return NULL;
}
```

- If program is compiled and run with two threads, it gives the correct results but it lasts much longer.
- Is this due to overhead due to starting up and joining threads? Estimate the overhead by

```
void* Thread_function(void* ignore)
{ return NULL; }
```

- The threads alternate between executing the critical section and waiting and executing, because of the loop.

Global sum function with critical section after loop

```
void* Thread_sum(void* rank)
{
    long my_rank = (long) rank;
    double factor, my_sum = 0.0;
    long long i;
    long long my_n = n/thread_count;
    long long my_first_i = my_n*my_rank;
    long long my_last_i = my_first_i + my_n;

    if (my_first_i % 2 == 0)
        factor = 1.0;
    else
        factor = -1.0;
    for (i = my_first_i; i < my_last_i; i++, factor = -factor)
        my_sum += factor/(2*i+1);

    while (flag != my_rank);
    sum += my_sum;
    flag = (flag + 1) % thread_count;

    return NULL;
}
```

8 Mutexes

- Threads continually use the CPU in programs with busy-waiting.
- mutex is an abbreviation for mutual exclusion.
- A mutex is a special type of variable that, together with a couple of special functions, can be used to restrict access to a critical section to a single thread at a time.
- The Pthreads standard includes a special type for mutexes: *pthread_mutex_t*.
- This type of variable needs to be initialized by the system before it is used.
- This can be done with a call to

```
int pthread_mutex_init(
    pthread_mutex_t*  mutex_p /* out */,
    const pthread_mutexattr_t attr_p /* in: won't use it, pass NULL */ );
```

- When program finishes using a mutex, it should call

```
int pthread_mutex_destroy(pthread_mutex_t* mutex_p /* in/out */ );
```

- To gain access to a critical section (the thread waits until no other thread is in the critical section)

```
int pthread_mutex_lock(pthread_mutex_t* mutex_p /* in/out */ );
```

- When a thread is finished executing the code in a critical section, it should call (notifies it is done)

```
int pthread_mutex_unlock(pthread_mutex_t* mutex_p /* in/out */ );
```

Global sum function that uses a mutex

```
void* Thread_sum(void* rank)
{
    long my_rank = (long) rank;
    double factor;
    long long i;
    long long my_n = n/thread_count;
    long long my_first_i = my_n*my_rank;
    long long my_last_i = my_first_i + my_n;
    double my_sum = 0.0;

    if (my_first_i % 2 == 0)
        factor = 1.0;
    else
        factor = -1.0;

    for (i = my_first_i; i < my_last_i; i++, factor = -factor)
        my_sum += factor/(2*i+1);

    pthread_mutex_lock(&mutex);
    sum += my_sum;
    pthread_mutex_unlock(&mutex);

    return NULL;
}
```

- The first thread to call *pthread_mutex_lock* will, effectively, "lock the door" to the critical section.
- Any other thread that attempts to execute the critical section code must first also call *pthread_mutex_lock*, and until the first thread calls *pthread_mutex_unlock*, all the threads that have called *pthread_mutex_lock* will block their calls, they will just wait until the first thread is done.
- After the first thread calls *pthread_mutex_unlock*, the system will choose one of the blocked threads and allow it to execute the code in the critical section.
- This process will be repeated until all the threads have completed executing the critical section.
- With mutexes, the order in which the threads execute the code in the critical section is more or less random. Eventually all threads will obtain the lock.

Comparing performance of busy-waiting with the version of mutexes

- there might be no significance difference provided the number of threads does not exceed that of cores, since each thread enters the critical section once (assuming the critical is not very long, or Pthreads functions are very slow) one would not expect that the threads to be delayed very much by waiting to enter the critical section.
- However, if the number of threads increases beyond the number of cores, the busy-wait version performance degrades, because the order that CPU schedule threads is not necessarily the same as the order given by the master thread.

9 Producer-Consumer Synchronization and Semaphores

Although busy-waiting generally wastes CPU resources, it has the property that threads execute in order a critical section and this might be useful in situations where one needs that the threads must executed the critical section in their rank order.

Consider the case a thread needs to send a message to the next in rank thread ($0 \rightarrow 1 \rightarrow 2 \rightarrow \cdots \rightarrow t-1 \rightarrow 0$). After "receiving" the message the next in rank thread should print the message.

Consider a shared array of *char**. Then each thread can allocate storage for the message it is sending, and after initializing it, it sets a pointer in the shared array to refer to it. To avoid dereferencing undefined pointers, the main thread can set the shared array individual entries to NULL.

```c
/* messages has type char**. It is allocated in main */
/* Each entry is set to NULL in main */
void* Send_msg(void* rank)
{
    long my_rank = (long) rank;
    long dest = (my_rank + 1) % thread_count;
    long source = (my_tank + thread_count - 1) % thread_count;
    char* my_msg = malloc(MSG_MAX*sizeof(char));

    sprintf(my_msg, "Hello to %ld from %ld, dest, my_rank);
    messages[dest] = my_msg;

    if (messages[my_rank] != NULL)
        printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
    else
        printf("Thread %ld > No message from %ld\n", my_rank, source);

    return NULL;
}
```

Problems and potential fixes

- In a system with more threads t than cores, thread 0 may finish before thread $t-1$ has copied the message into the messages array (dereferencing a null pointer).
- Fixing with busy-wait: Replace the *if* statement with

```c
while (messages[my_rank] == NULL);
printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
```

- This solution has the same problem that any busy-wait solution.
- Fixing with mutexes: After the shared array is made to refer the message, one would like to notify the next thread that it can proceed to print the message. Something like this

```
...
messages[dest] = my_msg;
Notify thread dest that it can proceed;
...
Await notification from thread source;
printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
...
```

- Not clear how to do it with mutexes. Maybe

```c
...
pthread_mutex_lock(mutex[dest]); // mutexes init by master to be unlocked
...
messages[dest] = my_msg;
pthread_mutex_unlock(mutex[dest]); // Notify thread dest that it can proceed
...
pthread_mutex_lock(mutex[my_rank]); // Await notification from thread source
printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
...
```

- This will be a problem since one does not know when threads will reach the calls to *pthread_mutex_lock*: There might be two critical sections. If thread 0 reaches the second call to *pthread_mutex_lock* before thread 1 reaches the first, thread 0 will acquire the lock and print the message dereferencing a null pointer.
- Fixing with semaphores:
  - Semaphores can be thought of a special type of *unsigned int*, so they can take values $0, 1, \cdots$.
  - In most cases one is interested in binary semaphores (the ones that take values 0 and 1).
  - Roughly: 0 corresponds to a locked mutex and 1 corresponds to an unlocked mutex.
  - To use a binary semaphore as a mutex, initialize it to be 1.
  - Before the critical section one needs to protect place a call to a function *sem_wait*.
  - A thread that executes *sem_wait* will block if the semaphore is 0.
  - If semaphore is not zero, the thread will decrement the semaphore and proceed.
  - After executing the critical section, the thread calls *sem_post*, which increments the semaphore, and a thread waiting in *sem_wait* can proceed.
  - The crucial difference between semaphores and mutexes is that there is no ownership associated with a semaphore.

- The main thread can initialize all of the semaphores to 0, and then any thread can execute a *sem_post* on any of the semaphores, and similarly, any thread can execute *sem_wait* on any of the semaphores.

```c
/* messages has type char**. It is allocated in main */
/* semaphores is allocated and initialized to 0 (locked) in main */
void* Send_msg(void* rank)
{
    long my_rank = (long) rank;
    long dest = (my_rank + 1) % thread_count;
    long source = (my_tank + thread_count - 1) % thread_count;
    char* my_msg = malloc(MSG_MAX*sizeof(char));

    sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
    messages[dest] = my_msg;

    sem_post(&semaphores[dest]); // unlock the semaphore of dest

    sem_wait(&semaphores[my_rank]); // wait for semaphore to be unlocked

    printf("Thread %ld > %s\n", my_rank, messages[my_rank]);

    return NULL;
}
```

- Syntax of various semaphore functions:

```c
int sem_init(
    sem_t* semaphore_p /* out */,
    int shared /* in */,
    unsigned initial_val /*in */);

int sem_destroy(sem_t*, semaphore_p, /* in/out */);
int sem_post(sem_t*, semaphore_p /* in/out */);
int sem_wait(sem_t* semaphore_p /* in/out */);
```

- Will not use the second argument to *sem_init* (a constant 0 can be passed in).
- Semaphores are not part of Pthreads. Need to add #include <semaphore.h>.
- There was no critical section in the message-sending problem.
- Rather, thread *my_rank* could not proceed until thread *source* has finished creating the message.
- This type of synchronization, when a thread cannot proceed until another thread has taken some action is called produce-consumer synchronization.

10 Barriers and Condition Variables

Another problem in shared-memory programming: Synchronizing the threads by making sure that they all are at the same point in a program.

Such a point of synchronization is called a barrier because no thread can proceed beyond until all threads have reach it.

Applications:

- Timing some part of a multithreaded program: One would like for all the threads to start the timed code at the same instant

```
/* Shared */
double elapsed_time;
...
/* Private */
double my_start, my_finish, my_elapsed;
...
Synchronize threads;
Store current time in my_start;
/* Execute timed code */
...
Store current time in my_finish;
my_elapsed = my_finish - my_started;

elapsed = Maximum of my elapsed times;
```

  With this, one is sure that all of the threads will record *my_start* approximately at the same time.

- Debugging: Usually difficult. One could have each thread print a message indicating which point is has reached in the program but the volume of the output could become overwhelming. Barriers provide an alternative

```
point in the program one wants to reach
barrier;
if (my_rank == 0)
{
    printf("All threads reached this point\n");
    fflush(stdout);
}
```

Many implementations of Pthreads do not provide barriers. So, one needs to develop its own implementations if one wants to have a portable code.

There are some options:

- Busy-waiting and a mutex: Use a shared counter protected by the mutex. When the counter indicates that every thread has entered the critical section, threads can leave a busy-wait loop.

```
/* Shared and initialized by the main thread */
int counter; /* initialize to 0 */
int thread_count;
pthread_mutex_t barrier_mutex;
...
void* Thread_work(...)
{
    ...
    /* Barrier */
    pthread_mutex_lock(&barrier_mutex);
    counter++;
    pthread_mutex_unlock(&barrier_mutex);

    while (counter < thread_count);
    ...
}
```

This implementation will have the usual problems that other busy-waiting codes do.

Another issue is the shared variable *counter*. What happens if one needs a second barrier and try to reuse the counter?

- When the first barrier is completed, *counter* will have the value *thread_count*.
- It needs to be reset to 0. Otherwise, the *while* condition for the first barrier will be false and the barrier will not cause the threads to block.
- Furthermore, any attempt to reset *counter* is doomed to failure.
- If the last thread to enter the loop tries to reset it, some thread in the busy-wait may never see the fact that *counter* == *thread_count*, and that thread may hang in the busy-wait.
- If some thread tries to reset the counter after the barrier, some other thread may enter the second barrier before the counter is reset and its increment to the *counter* will be lost. The all threads will hang in the second busy-wait loop.
- One needs a *counter* variable for each instance of the barrier.

• Semaphores: It is possible to implement a barrier with semaphores in such a way it has less problems than the busy-wait and mutex barrier approach

```
/* Shared variables */int counter; /* initialized to 0 */
sem_t count_sem /* initialized to 1 */
sem_t barrier_sem /* initialized to 0 */
...
void* Thread_work
{
    ...
    /* Barrier */
    sem_wait(&count_sem);

    if (counter == thread_count - 1)
    {
        counter = 0;
        sem_post(&count_sem);
        for (j = 0; j < thread_count - 1; j++)
            sem_post(&barrier_sem);
    }
    else
    {
        counter++;
        sem_post(&count_sem);
        sem_wait(&barrier_sem);
    }
    ...
}
```

- *counter* variable used to determine how many threads have entered the barrier.
- There are two semaphores: *count_sem* protects the counter and *barrier_sem* to block threads that have entered the barrier.
- *count_sem* is initialized to 1, so the first thread to reach the barrier will proceed past the call to *sem_wait*. Subsequent threads, will block until they can exclusively access to the *counter*.
- When a thread has exclusive access to the *counter*, it checks to see if *counter* < *thread_count - 1*.

  - It it is, the thread increments *counter*, relinquishes the lock (*sem_post(&count_sem)*) and blocks in *sem_wait(&barrier_sem)*.
  - If *counter* == *thread_count - 1*, the thread is the last to enter the barrier, it resets the *counter* to 0 and unlock *count_sem* by calling *sem_post(&count_sem)*. Now, it notifies all other threads that they can proceed by *sem_post(&barrier_sem)* for each of the *thread_count - 1* threads that are blocked in *sem_wait(&barrier_sem)*.
  - Notice that a semaphore is an *unsigned int*, and the calls to *sem_post* increment it, while the calls to *sem_wait* decrement it – unless it is already 0 in which case the calling threads will block until it is positive again, and they will decrement it when they unblock.
  - Can one reuse the data structures from the first barrier if one wants to execute a second barrier?em
    * *counter* has been reset before releasing any of the threads from the barrier. It can be reused.
    * *count_sem* has been rest to 1 before any thread can leave the barrier. It can be reused.
    * Since there is exactly one *sem_post* for each *sem_wait*, it seems that the value of *barrier_sem* will be 0 when the threads start executing a second barrier.

      However, suppose there are two threads.

      1. Suppose thread 0 is block in *sem_wait(&barrier_sem)* in he first barrier, while thread 1 is executing the loop of *sem_post*.
      2. Also suppose, that OS has seen thread 0 idle, and descheduled it out.
      3. Then thread 1 can go on to the second barrier.
      4. Since *counter* == 0, it will execute the *else* clause.
      5. After incrementing *counter*, it executes *sem_post(&count_sem)*, and then *sem_wait(&barrier_sem)*.
      6. However, it thread 0 is still descheduled, it will not have decremented *barrier_sem*.
      7. When thread 1 reaches *sem_wait(&barrier_sem)*, *barrier_sem* will still be 1, so it will decrement the barrier and proceed.
      8. When thread 0 starts executing again, it will still be blocked in the first *sem_wait(&barrier_sem)*, and thread 1 will proceed to the second barrier before thread 0 has entered it.
      9. Reusing *barrier_sem* results in a race condition.

- Condition variables: It is a better approach for creating barriers in Pthreads.
  - A condition variable is a dat object that allows a thread to suspend execution until a certain condition occurs.
  - When the condition occurs another thread can signal the thread to "wake up".
  - A condition variable is always associated to a mutex.
  - Typically, condition variables are used in constructs similar to

```
lock mutex;

if condition has occurred
    signal thread(s);
else
    unlock the mutex and block; // when thread is unblocked, mutex is
        relocked

unlock mutex;
```

- Condition variables in Pthreads have a type *pthread_cond_t*
- Syntax of various functions:

```
// unblocks one of the blocked threads
int pthreads_cond_signal(pthread_cond_t* cond_var_p /* in/out */);
// unblocks all of the blocked threads
int  pthreads_cond_broadcast(pthread_cond_t* cond_var_p /* in/out */);

/*
    unblock the mutex referred to by mutex_p and cause the executing thread
    to block until it is unblocked by another's thread call to
    pthread_cond_signal or pthread_cond_broadcast.

    When the thread is unblocked, it reacquires the mutex.
*/
int pthread_cond_wait(
    pthread_cond_t* cond_var_p /* in/out */,
    pthread_mutex_t* mutex_p /* in/out */);
```

- In effect, *pthread_cond_wait* implements the following sequence of functions:

```
pthread_mutex_unlock(&mutex_p);
wait_on_signal(&cond_var_p);
pthread_mutex_lock(&mutex_p);
```

- The following implements a barrier with a condition variable:

```
/* Shared */
int counter = 0;
pthread_mutex_t mutex;
pthread_cont_t cond_var;
...
void * Thread_work(...)
{
    /* Barrier */
    pthread_mutex_lock(&mutex);
    counter++;

    if (counter == thread_count)
    {
        counter = 0;
        pthread_cond_broadcast(&cond_var);
    }
    else
        while (pthread_cond_wait(&cond_var, &mutex) != 0);

    pthread_mutex_unlock(&mutex);
    ...
}
```

  - It is possible that events other than the call to *pthread_cond_broadcast* can cause a suspended thread to unlock.
  - Hence, the call to *pthread_cond_wait* is usually placed in a *while* loop. If some other event than *pthread_cond_signal* or *pthread_cond_broadcast* unlocks a thread, then the return value of *pthread_cond_wait* will be nonzero, and the unlocked thread will call *pthread_cond_wait* again.

  - For a barrier to function correctly, it is essential that the call to *pthread_cond_wait* unlock the mutex. If it does not, then only one thread could enter the barrier; all other threads would block in the call to *pthread_mutex_lock*, the first thread to enter the barrier would block in the call to *pthread_cond_wait*, and the program will hang.
  - Also, the mutex has to be relocked before one returns from the call to *pthread_cond_wait*. One obtains the lock when returned from the call to *pthread_mutex_lock*. Hence, at some point one should relinquish the lock through a call to *pthread_mutex_unlock*.

- Condition variables should be initialized and destroyed.

```c
int pthread_cond_init(
    pthread_cond_t* cond_p /* out */,
    const pthread_condattr_t* cond_attr_p /* in */);

int pthread_cond_destroy(pthread_cond_t* cond_p /* in/out */);
```

  We will not use the second argument of *pthread_cond_init*. It will be call with NULL as second argument.

11 Read-Write Locks

Problem: Control access to a large, shared data structure, which can be searched or updated by the threads.

For simplicity, assume a sorted linked list of *int*s, with operations of interest *Member, Insert, Delete*.

A list is compossed of a collection of nodes. Assume each node is a structure like

```c
struct list_node_s
{
    int data;
    struct list_node_s* next;
}
```

A pointer *head_p*, with type *struct list_node_s** refers to the first node in the list.

The *next* member of the last node is *NULL*.

Typically, the headers of functions to perform operations on the list are:

```c
int Member(int value, struct list_node_s* head_p)
int Insert(int value, struct list_node_s* head_p)
int Delete(int value, struct list_node_s* head_p)
```

1. *Member* uses a pointer to traverse the list until it either finds the desired value or determines that it cannot be in the list. Since the list is sorted, the latter condition occurs when the current node pointer points to *NULL* or its value is greater than the desired value.
2. *Insert* begins by searching for the correct position in which to insert the new node. It must search until it finds a node whose *value* is greater than the *value* to be inserted. When it finds this node, it needs to insert the new node before the node found. Since it is not possible to back up the list, one could keep track of the predecessor node by a pointer, and to insert the new node.
3. *Delete* is similar to *Insert* function in that it also needs to keep track of the predecessor of the current node while it is searching for the node to be deleted.

Now, define these functions in a Pthreads program:

- Define *head_p* to be a global variable. That simplify the function headers.

- Multiple threads can simultaneously execute *Member* since only reading operations are involved.
- *Insert* and *Delete* also write to memory locations, and this might cause problems if one tries to execute either of these operations at the same time as another operation.

How to solve this problem?

- lock the list: any time a thread attempts to access it. For example, a call to each of the three functions can be protected by a mutex

```
Pthread_mutex_lock(&list_mutex);
Member(value);
Ptread_mutex_unlock(&list_mutex);
```

  This serializes access to the list. It fails to exploit the opportunity for parallelism if most calls are to *Member*.

- "finer-grain" locking: instead of locking the entire list, lock individual nodes. For example,

```
struct list_node_s
{
    int data;
    struct list_node_s* next;
    pthread_mutex_t mutex;
}
```

  Now each time one tries to access a node, it is needed first to look the mutex associated to the node. Notice, that this will also require that one has a mutex associated to *head_p*. This will generate a more complex implementation of each of the functions and certainly it will not only slow down the program, since each time a node is accessed, a mutex must be locked and unlocked, but also the amount of storage for each node will increase substantially.

- Pthreads read-write locks: it is like a mutex except that it provides two lock functions.
  - The first function locks the read-write lock for reading, while the second locks it for writing.
  - Multiple threads can simultaneously obtain the lock by calling the read-lock function
  - Only one thread can obtain the lock by calling the write-lock function.
  - If any threads own the lock for reading, any threads that want to obtain the lock for writing will block in the call to the write-lock function.
  - If any thread owns the lock for writing, any threads that want to obtain the lock for reading or writing will block in their respective locking functions.
  - The syntax for these functions is

```
int pthread_rwlock_rdlock(pthread_rwlock_t* rwlock_p /* in/out */); //
    locks for reading
int pthread_rwlock_wrlock(pthread_rwlock_t* rwlock_p /* in/out */); //
    locks for writing
int pthread_rwlock_unlock(pthread_rwlock_t* rwlock_p /* in/out */); //
    unlocks
int pthread_rwlock_init(
    pthread_rwlock_t* rwlock_p /* out */,
    const pthread_rwlockattr_t* attr_p /* in */); // use NULL for the
        second argument
int pthread_rwlock_destroy(pthread_rwlock_t* rwlock_p /* in/out */);
```

  - Using Pthreads one can protect the linked list functions by (ignoring function return values)

```
pthread_rwlock_rdlock(&rwlock);
Member(value);
pthread_rwlock_unlock(&rwlock);
...
pthread_rwlock_wrlock(&rwlock);
Insert(value);
pthread_rwlock_unlock(&rwlock);
...
pthread_rwlock_wrlock(&rwlock);
Delete(value);
pthread_rwlock_unlock(&rwlock);
```

Performance of each implementation depends on the proportion of calls to *Member, Insert, Delete.*

12 Cache Coherence, and False Sharing

- Cache coherence: Consider the matrix-vector multiplication example
  - Main thread initialized a $m \times n$ matrix A, and an n-dimensional vector x.
  - There are t threads.
  - Each thread is responsible for computing m/t components of the product $y = Ax$.
  - Data structures representing A, x, y, m, n, are all shared.
  - Suppose p is the total number of floating point additions and multiplications.
  - Notice that $p = m(2n+1) = \mathcal{O}(mn)$.

```
void* Pth_mat_vect(void* rank)
{
    long my_rank = long(rank);
    int i, j;
    int local_m = m/thread_count;
    int my_first_row = my_rank*local_m;
    int my_last_row = (my_rank+1_*m_local - 1;

    for (i = my_first_row; i <= my_last_row; i++)
    {
        y[i] = 0.0;

        for (j = 0; j < n; j++)
        y[i] += A[i][j]*x[j];
    }

    return NULL;
}
```

    * Single thread: The performance depends on the relationship between m and n.
      1. Case m large, n small: Many *write-misses* because of y initialization.
      2. Case n large, m small: Many *read-misses* because loading of x.
      3. Case m, n medium size: Not as many *write-misses* and *read-misses*.

4. Virtual memory: how frequently does the CPU need to access the page table in main memory?

∗ Multiple thread: cache coherence is enforced at the "cache-line" level (each time any value in a cache line is written, if the line is also stored in another processor's cache, the entire line will be invalidated). This triggers false sharing.

1. Case m large, n small: unlike false sharing of y near *my_ last_ row* and *my_first_ row*.
2. Case n large, m small: Assuming that y is stored in a single cash. Every write to same element of y will invalidate the line in the other processors' cache. They will have to reload y. This will happens p/t times in each thread.
3. Case m, n medium size: unlike false sharing of y near *my_ last_ row* and *my_first_ row*.
4. Virtual memory: how frequently does the CPU need to access the page table in main memory?

To avoid false-sharing

∗ one may "pad" the y vector with dummy elements to insure that any update by one thread won't affect another thread's cache line.

∗ another possibility is to have each thread to use its own private storage during the multiplication loop, and then update the shared storage when they are done.

13 Thread Safety

A block is tread-safe if it can be simultaneously executed by multiple threads without causing problems.

Example: Tokenize file of English words separated by a white space (a space, a tab, or a newline).

- Simple approach, divide the input file into lines of text and assign the lines to threads in round-robin fashion.
- One can serialize access to the lines of input using semaphores.
- After a thread has read a single line of input, it can tokenize it.
- One way is with the *strtok(char* string, const char*)* function in *string.h*.
- The usage of this function is a little bit unusual: the first time it is called the string argument should be the text to be tokenized and the second argument are the possible separators. For subsequent calls the first argument should be *NULL*.
- This is because in the first call, the functions catches a pointer to *string*, and for subsequent calls it returns successive tokens taken from the cached copy.
- The behavior of *strtok* is because it declares a variable that caches the input line to have *static* storage class, and since it is shared, it is not private.
- *strtok* is not thread-safe.
- Neither *random* in *stdlib.h* not *localtime* in *time.h* are thread-safe.
- In some cases there exists a thread-safe version called *strtok_ r* (the *_ r* suggests the function is reentrant).