Mutexes

Since a thread that is busy-waiting may continually use the CPU, busy-waiting is generally not an ideal solution to the problem of limiting access to a critical section.

- Two better solutions are mutexes and semaphores.

Mutex is an abbreviation of mutual exclusion, and 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.

- Thus a mutex can be used to guarantee that one thread "excludes" all other threads while it executes the critical section.
  - Hence, the mutex guarantees mutually exclusive access to the critical section.
- The Pthreads standard includes a special type for mutexes: pthread_mutex_t.
  - A variable of type pthread_mutex_t needs to be initialized by the system before it's used.
  - This can be done by

```
int pthread_mutex_init(
      pthread_mutex_t*             mutex_p   /* out */,
      const pthread_mutexattr_t*   attr_p    /* in  */);
```

    - We won't make use of the second argument, so we'll just pass in NULL to use the default attributes.
    - Sometimes you will see

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

      Which does the same thing
- When a Pthreads program finishes using a mutex it should call

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

- The point of a mutex is to protect a critical section from being entered by more than one thread at a time. To gain access to a critical section, a thread will lock the mutex, do its work, and then unlock the mutex to let other threads execute the critical section.
  - To lock the mutex:

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

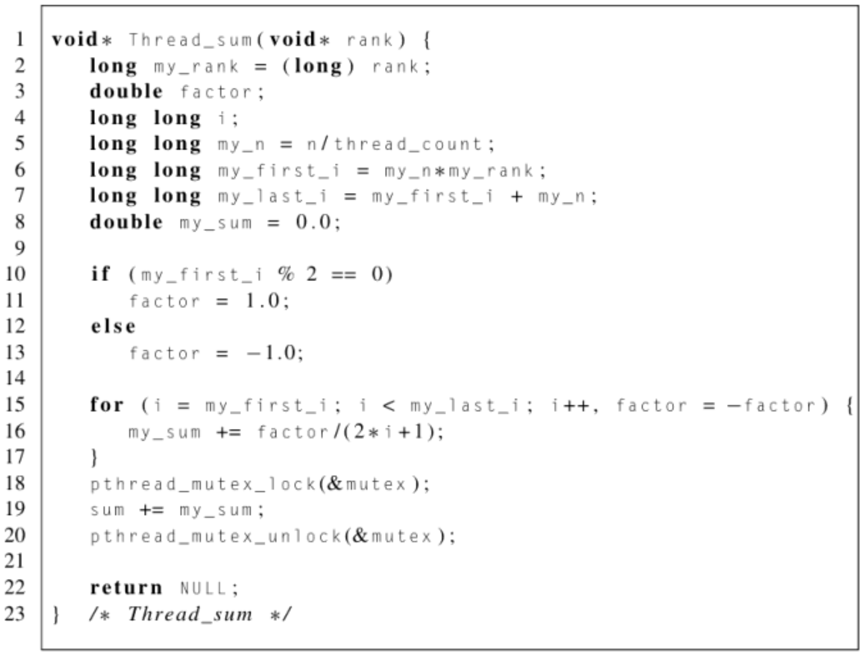
  - To unlock mutex:

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

  - Pthread_mutex_lock will cause the thread to wait until no other thread is in the crit section and the call to pthread_mutex_unlock notifies the system that the calling thread has completed execution of the code in crit section
- We can use mutexes instead of busy-waiting in our global sum program by declaring a global mutex variable, having the main thread initialize it, and then, instead of busy-waiting and incrementing a flag, the threads call pthread_mutex_lock before entering the

critical section, and they call pthread_mutex_unlock when they're done with the critical section.

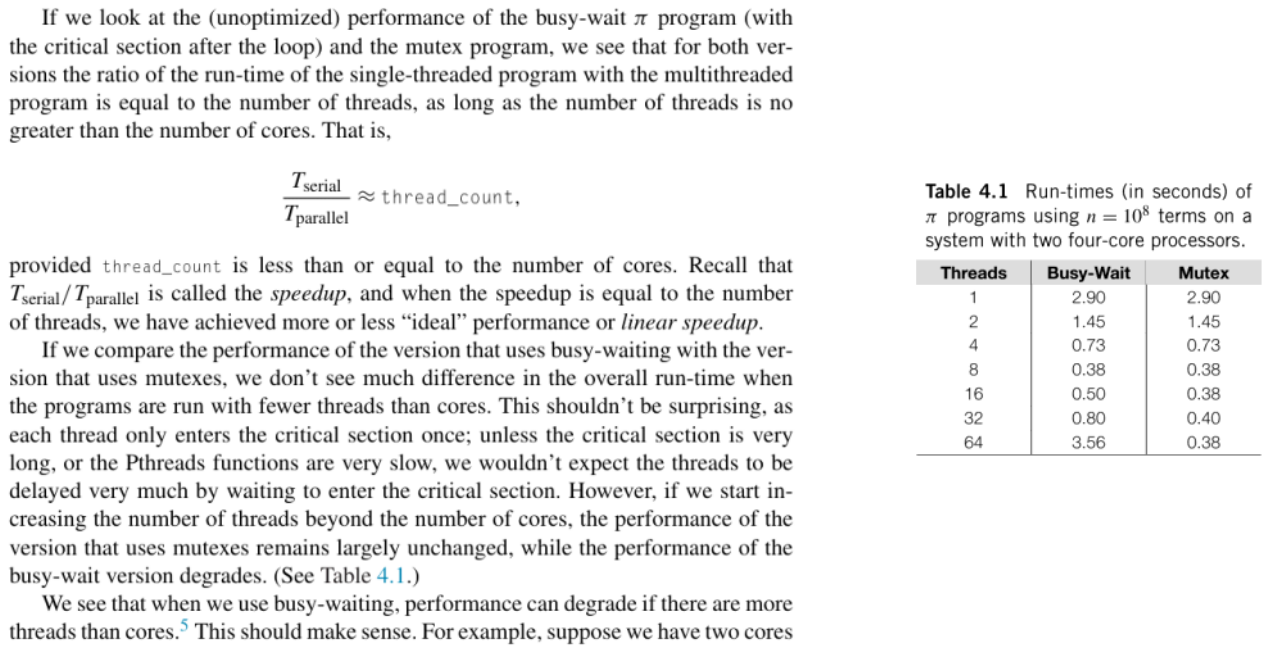
- Example:



```
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;
}  /* Thread_sum */
```

Program 4.6: Global sum function that uses a mutex.

- We see that when we use busy-waiting, performance can degrade if there are more threads than cores while mutexes don't suffer this.



If we look at the (unoptimized) performance of the busy-wait π program (with the critical section after the loop) and the mutex program, we see that for both versions the ratio of the run-time of the single-threaded program with the multithreaded program is equal to the number of threads, as long as the number of threads is no greater than the number of cores. That is,

$$\frac{T_{\text{serial}}}{T_{\text{parallel}}} \approx \texttt{thread\_count},$$

provided thread_count is less than or equal to the number of cores. Recall that $T_{\text{serial}}/T_{\text{parallel}}$ is called the *speedup*, and when the speedup is equal to the number of threads, we have achieved more or less "ideal" performance or *linear speedup*.

If we compare the performance of the version that uses busy-waiting with the version that uses mutexes, we don't see much difference in the overall run-time when the programs are run with fewer threads than cores. This shouldn't be surprising, as each thread only enters the critical section once; unless the critical section is very long, or the Pthreads functions are very slow, we wouldn't expect the threads to be delayed very much by waiting to enter the critical section. However, if we start increasing the number of threads beyond the number of cores, the performance of the version that uses mutexes remains largely unchanged, while the performance of the busy-wait version degrades. (See Table 4.1.)

We see that when we use busy-waiting, performance can degrade if there are more threads than cores.[5] This should make sense. For example, suppose we have two cores

Table 4.1 Run-times (in seconds) of π programs using $n = 10^8$ terms on a system with two four-core processors.

Threads	Busy-Wait	Mutex
1	2.90	2.90
2	1.45	1.45
4	0.73	0.73
8	0.38	0.38
16	0.50	0.38
32	0.80	0.40
64	3.56	0.38

- Notes:
  - In busy waiting we have to use more of the CPU to run operations that will keep us in busy-waiting, all these calls add up as well as all the waiting that has to be done
  - When we have less threads than cores we see that performance between busy-waiting and mutexes are the same but when there are more threads than cores we see a poor performance by busy-waiting while mutexes stay the same
    - This is because when there are less threads than cores there isnt too much

waiting and less jumps between cores. This is all amplified when there are more threads than cores.