Critical sections

- Matrix-vector multiplication was very easy to code because the shared-memory locations were accessed in a highly desirable way.
- After initialization, all of the variables—except y—are only *read* by the threads.
- Although threads do make changes to y, only one thread makes changes to any individual component.
  - No attempts by two (or more) threads to modify any single component.
- What happens if this isn't the case? What happens when multiple threads update a single memory location?

Serial code

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

```
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

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

- We can try to parallelize this in the same way we parallelized the matrix-vector multiplication program
  - divide up the iterations in the **for** loop among the threads and make sum a shared variable.

- Assume that the number of threads, *thread_count* or *t*, evenly divides the number of terms in the sum, *n*.
  - Thread 0, calculates the first n/t terms
  - Thread 1, calculates the second n/t terms etc.

A thread function for computing π

```
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)  /* my_first_i is even */
      factor = 1.0;
   else  /* my_first_i is odd */
      factor = -1.0;

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

   return NULL;
}  /* Thread_sum */
```

Using a dual core processor

	n			
	10^5	10^6	10^7	10^8
π	3.14159	3.141593	3.1415927	3.14159265
1 Thread	3.14158	3.141592	3.1415926	3.14159264
2 Threads	3.14158	3.141480	3.1413692	3.14164686

- Note that as we increase n, the estimate with one thread gets better and better.
- In fact, if we ran the program several times with two threads and the same value of *n*, we would see that the result computed by two threads *changes* from run to run. Why?

Race condition

- Remember that the addition of two values is typically not a single machine instruction.

- For example, although we can add the contents of a memory location y to a memory location x with a single C statement,

  x = x + y;

  what the machine does is typically more complicated

- Get x and y from memory → Put values in registers → calculate the sum → store the result in memory

Race condition

- Suppose that we have two threads, and each thread will execute the following code:

```
y = Compute(my_rank);
x = x + y;
```

- Let's also suppose that thread 0 computes y = 1 and thread 1 computes y = 2

Possible race condition

Time	Thread 0	Thread 1
1	Started by main thread	
2	Call `Compute()`	Started by main thread
3	Assign `y = 1`	Call `Compute()`
4	Put `x=0` and `y=1` into registers	Assign `y = 2`
5	Add 0 and 1	Put `x=0` and `y=2` into registers
6	Store 1 in memory location `x`	Add 0 and 2
7		Store 2 in memory location `x`

Race condition - comments

- This example illustrates a fundamental problem in shared-memory programming:
  - when multiple threads attempt to update a shared resource—in our case a shared variable—the result may be unpredictable.

- When multiple threads attempt to access
  1. a shared resource
  2. at least one of the accesses is an update and
  3. the accesses can result in an error,

  we have a **race condition**.

- In our example, the code x = x + y is a **critical section**, that is, it’s a block of code that updates a shared resource that can only be updated by one thread at a time.

Busy-waiting

- When thread 0 wants to execute the statement x = x + y, it needs to first make sure that thread 1 is not already executing the statement.

- Once thread 0 makes sure of this, it needs to provide some way for thread 1 to determine that it is executing the statement

- Finally, after thread 0 has completed execution of the statement, it needs to provide some way for thread 1 to determine that it is done

- A simple approach that doesn't involve any new concepts is the use of a flag variable. Suppose flag is a shared **int** that is set to 0 by the main thread.

```
y = Compute(my_rank);
while (flag != my_rank);
x = x + y;
flag++;
```

Waste a lot of CPU cycles and

beware of optimizing compilers, though!

Pthreads global sum with busy-waiting

```
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) {
      while (flag != my_rank);
      sum += factor/(2*i+1);
      flag = (flag+1) % thread_count;
   }

   return NULL;
}  /* Thread_sum */
```

Pthreads global sum with busy-waiting

```
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) {
      while (flag != my_rank);
      sum += factor/(2*i+1);
      flag = (flag+1) % thread_count;
   }

   return NULL;
}  /* Thread_sum */
```

Is it efficient?

On a dual-core system:

Parallel code = 19.5 seconds

Serial code = 2.8 seconds

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;
}  /* Thread_sum */
```

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;
}  /* Thread_sum */
```

Parallel code = 1.5 seconds

Serial code = 2.8 seconds

Mutexes

- A thread that is busy-waiting may continually use the CPU accomplishing nothing.

- Mutex (mutual exclusion) is a special type of variable that can be used to restrict access to a critical section to a single thread at a time.

Thread Management - Mutexes

- A typical sequence in the use of a mutex is as follows:
  - Create and initialize a mutex variable
  - Several threads attempt to lock the mutex
  - Only one succeeds and that thread owns the mutex
  - The owner thread performs some set of actions
  - The owner unlocks the mutex
  - Another thread acquires the mutex and repeats the process
  - Finally the mutex is destroyed

Mutexes

- Used to guarantee that one thread “excludes” all other threads while it executes the critical section.

- The Pthreads standard includes a special type for mutexes: pthread_mutex_t.

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

Mutexes

- When a Pthreads program finishes using a mutex, it should call

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

- In order to gain access to a critical section a thread calls

```
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

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

```c
pthread_t tid[2];
int counter;
pthread_mutex_t lock;

int main(void)
{
    int i = 0;
    int err;

    if (pthread_mutex_init(&lock, NULL) != 0)
    {
        printf("\n mutex init failed\n");
        return 1;
    }

    while(i < 2)
    {
        err = pthread_create(&(tid[i]), NULL, doSomeThing, NULL);
        if (err != 0)
            printf("\ncan't create thread :[%s]", strerror(err));
        i++;
    }

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);

    return 0;
}
```

```
void* doSomeThing(void *arg)
{
    pthread_mutex_lock(&lock);
    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d started\n", counter);

    for(i=0; i<(0xFFFFFFFF);i++);
    printf("\n Job %d finished\n", counter);

    pthread_mutex_unlock(&lock);
    return NULL;
}
```

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;
}  /* Thread_sum */
```

Performance

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

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

On a system with two four-core processors.

Busy-wait when t >> n

Time	`flag`	Thread 0	Thread 1	Thread 2	Thread 3	Thread 4
0	0	crit sect	busy wait	susp	susp	susp
1	1	terminate	crit sect	susp	busy wait	susp
2	2	—	terminate	susp	busy wait	busy wait
⋮	⋮			⋮	⋮	⋮
?	2	—	—	crit sect	susp	busy wait

Possible sequence of events with busy-waiting and more threads than cores.

Issues

- Busy-waiting enforces the order threads access a critical section.
- Using mutexes, the order is left to chance and the system.
- There are applications where we need to control the order threads access the critical section.

Send messages to threads in sequence

```
void* Send_msg(void* rank) {
   long my_rank = (long) rank;
   long dest = (my_rank + 1) % thread_count;
   long source = (my_rank + 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;
}  /* Send_msg */
```

Send messages to threads in sequence

```
void* Send_msg(void* rank) {
   long my_rank = (long) rank;
   long dest = (my_rank + 1) % thread_count;
   long source = (my_rank + 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;

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

   return NULL;
}  /* Send_msg */
```

Send messages to threads in sequence

```
void* Send_msg(void* rank) {
   long my_rank = (long) rank;
   long dest = (my_rank + 1) % thread_count;
   long source = (my_rank + 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;

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

   return NULL;
}  /* Send_msg */
```

What about mutexes?

Semaphores

- The semaphore is a special integer number that has only the following three actions:
  - Initialization
  - Wait
  - Signal - Post

- Functionality:
  - Initialize to an integer value. After initialization only wait() and signal() functions are allowed

  - wait(): If the semaphore <= 0, the process stalls. If the semaphore is > 0 then the semaphore is decreased by 1 and the process continues its actions.

  - signal() – post(): The semaphore is increased by 1

Semaphores - actions

```
wait (S) {
   while (S <= 0)
         ;
   S--;
}
```

```
signal(S) {
   S++;
}
```

Semaphores - actions

```
wait (S) {
   while (S <= 0)
         ;
   S--;
}
```

```
signal(S) {
   S++;
}
```

All changes to the semaphore are atomic operations!

Semaphores - actions

```
wait (S) {
  while (1) {
     lock(S_lock);
     if (S > 0) {
        S--;
        unlock(S_lock);
        break;
     }
     unlock(S_lock);
  }
}
```

```
signal (S) {
  lock(S_lock);
  S++;
  unlock(S_lock);
}
```

Semaphores - Notes

- A process cannot know beforehand the value of the semaphore. Thus, it does not know if it will need to stall during wait()

- There is no rule that guarantees which waiting process will acquire the semaphore after calling the signal()

- One mutex is equivalent to a semaphore that it is initialized to 1 (unlocked)

- The functions of semaphores are mentioned as P() (wait()) and V() (signal()) for historical reasons

Ordering problem

- We want process P1 to execute function S1 before process P2 executes the function S2
- We initialize the semaphore to 0

```
// P1
S1();
signal(sema);
```

```
// P2
wait(sema);
S2();
```

Producer-Consumer problem

- Structure: Array/Buffer (N size)
- Consumer: Removes data from the “end”
- Producer: Inserts data to “start”

```
#define NEXT(x) ((x + 1) % N)
item_t buffer[N]; int in=0, out=0, count=0;
```

Producer-Consumer problem

- Structure: Array/Buffer (N size)
- Consumer: Removes data from the “end”
- Producer: Inserts data to “start”

```
#define NEXT(x) ((x + 1) % N)
item_t buffer[N]; int in=0, out=0, count=0;
```

```
void enqueue(item_t item){
    item_t item;


    buffer[in] = item;
    in = NEXT(in);
    count++;
}
```

```
item_t dequeue(void){
    item_t item;


    item = buffer[out];
    out = NEXT(out);
    count--;
    return item
}
```

Producer-Consumer problem

- Structure: Array/Buffer (N size)
- Consumer: Removes data from the “end”
- Producer: Inserts data to “start”

```
#define NEXT(x) ((x + 1) % N)
item_t buffer[N]; int in=0, out=0, count=0;
```

```
void enqueue(item_t item){
    item_t item;
    while (count == N)
        ; // wait
    buffer[in] = item;
    in = NEXT(in);
    count++;
}
```

```
item_t dequeue(void){
    item_t item;
    while (count == 0)
        ; // wait
    item = buffer[out];
    out = NEXT(out);
    count--;
    return item
}
```

Producer-Consumer problem

- Structure: Array/Buffer (N size)
- Consumer: Removes data from the “end”
- Producer: Inserts data to “start”

```
#define NEXT(x) ((x + 1) % N)
item_t buffer[N]; int in=0, out=0, count=0;
```

```
void enqueue(item_t item){
    item_t item;
    while (count == N)
        ; // wait
    buffer[in] = item;
    in = NEXT(in);
    count++;
}
```

```
item_t dequeue(void){
    item_t item;
    while (count == 0)
        ; // wait
    item = buffer[out];
    out = NEXT(out);
    count--;
    return item
}
```

Producer-Consumer problem

```
#define NEXT(x) ((x + 1) % N)
item_t buffer[N]; int in=0, out=0, count=0;
```

```
void enqueue(item_t item){

    item_t item;
    while (count == N)
        ; // wait
    buffer[in] = item;
    in = NEXT(in);
    count++;
}
```

```
item_t dequeue(void){

    item_t item;
    while (count == 0)
        ; // wait
    item = buffer[out];
    out = NEXT(out);
    count--;
    return item
}
```

Producer-Consumer problem

```
#define NEXT(x) ((x + 1) % N)
item_t buffer[N]; int in=0, out=0, count=0;
sema_t mutex = semaphore(1);
```

```
void enqueue(item_t item){
    wait(mutex);
    item_t item;
    while (count == N)
        ; // wait
    buffer[in] = item;
    in = NEXT(in);
    count++;
    signal(mutex);
}
```

```
item_t dequeue(void){
    wait(mutex);
    item_t item;
    while (count == 0)
        ; // wait
    item = buffer[out];
    out = NEXT(out);
    count--;
    signal(mutex);
    return item
}
```

Producer-Consumer problem

```
#define NEXT(x) ((x + 1) % N)
item_t buffer[N]; int in=0, out=0, count=0;
sema_t mutex = semaphore(1);
```

```
void enqueue(item_t item){
    wait(mutex);
    item_t item;
    while (count == N)
        ; // wait
    buffer[in] = item;
    in = NEXT(in);
    count++;
    signal(mutex);
}
```

```
item_t dequeue(void){
    wait(mutex);
    item_t item;
    while (count == 0)
        ; // wait
    item = buffer[out];
    out = NEXT(out);
    count--;
    signal(mutex);
    return item
}
```

Does it work? Is there any deadlock?

Producer-Consumer solution

```
#define NEXT(x) ((x + 1) % N)
item_t buffer[N]; int in=0, out=0, count=0;
sema_t mutex = semaphore(1);
sema_t items = semaphore(0);
sema_t space = semaphore(N);
```

```
void enqueue(item_t item){
    item_t item;
    wait(space);
    wait(mutex);
    while (count == N)
        ; // wait
    buffer[in] = item;
    in = NEXT(in);
    count++;
    signal(mutex);
    signal(items);
}
```

```
item_t dequeue(void){
    item_t item;
    wait(items);
    wait(mutex);
    while (count == 0)
        ; // wait
    item = buffer[out];
    out = NEXT(out);
    count--;
    signal(mutex);
    signal(space);
    return item }
```

Syntax of the various semaphore functions

Semaphores are not part of Pthreads;
you need to add this.

```
#include <semaphore.h>

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  */);
```

Send messages to threads in sequence

```
/* messages is allocated and initialized to NULL 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;
   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 */

   /* Wait for our semaphore to be unlocked */
   sem_wait(&semaphores[my_rank]);
   printf("Thread %ld > %s\n", my_rank, messages[my_rank]);

   return NULL;
}  /* Send_msg */
```

Barriers

- Synchronizing the threads to make sure that they all are at the same point in a program is called a barrier.

- No thread can cross the barrier until all the threads have reached it.

Using barriers to time the slowest thread

```
/* 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_start;

elapsed = Maximum of my_elapsed values;
```

Using barriers for debugging

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

Busy-waiting and a Mutex

- Implementing a barrier using busy-waiting and a mutex is straightforward.
- We use a shared counter protected by the mutex.
- When the counter indicates that every thread has entered the critical section, threads can leave the critical section.

Busy-waiting and a Mutex

```
/* 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);
   . . .
}
```

We need one counter variable for each instance of the barrier, otherwise problems are likely to occur.

Implementing a barrier with semaphores

```
/* Shared variables */
int counter;          /* Initialize to 0 */
sem_t count_sem;      /* Initialize to 1 */
sem_t barrier_sem;    /* Initialize 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);
   }
   . . .
```

Condition Variables

- A condition variable is a data object that allows a thread to suspend execution until a certain event or condition occurs.
- When the event or condition occurs another thread can signal the thread to "wake up."
- A condition variable is always associated with a mutex.

Condition Variables

- Mutexes implement synchronization by controlling thread access to data
- Condition variables allow threads to synchronize based upon the actual value of data
- Without condition variables threads continually polling to check if the condition is met.
  - A condition variable is a way to achieve the same goal without polling.

Condition Variables

```
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

- Condition variables in Pthreads have type pthread_cond_t.

- Unblock *one* of the blocked threads:

```
int pthread_cond_signal(pthread_cond_t*  cond_var_p /* in/out */);
```

- Unblock *all* of the blocked threads

```
int pthread_cond_broadcast(pthread_cond_t* cond_var_p /* in/out */);
```

Condition Variables

- Unlock the mutex referred to by mutex_p and cause the executing thread to block
  - until it is unblocked by another thread's call to pthread_cond_signal or
  - pthread cond broadcast

```
int pthread_cond_wait(
      pthread_cond_t*     cond_var_p   /* in/out */,
      pthread_mutex_t*    mutex_p      /* in/out */);
```

- Initialization of a condition variable

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

Implementing a barrier with condition variables

```
/* Shared */
int counter = 0;
pthread_mutex_t mutex;
pthread_cond_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);
    . . .
}
```

```
#define NUM_THREADS  3
#define TCOUNT 10
#define COUNT_LIMIT 12

int     count = 0;
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;

int main(int argc, char *argv[])
{
  int i, rc;
  long t1=1, t2=2, t3=3;
  pthread_t threads[3];

  pthread_mutex_init(&count_mutex, NULL);
  pthread_cond_init (&count_threshold_cv, NULL);

  pthread_create(&threads[0], NULL, watch_count, (void *)t1);
  pthread_create(&threads[1], NULL, inc_count, (void *)t2);
  pthread_create(&threads[2], NULL, inc_count, (void *)t3);

  /* Wait for all threads to complete */
  for (i = 0; i < NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }
  printf ("Main(): Waited and joined with %d threads. Final value of count = %d. Done.\n",
          NUM_THREADS, count);

 pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_threshold_cv);
  pthread_exit (NULL); }
```

```
void *watch_count(void *t)
{
  long my_id = (long)t;
  printf("Starting watch_count(): thread %ld\n", my_id);

  pthread_mutex_lock(&count_mutex);
  while (count < COUNT_LIMIT) {
    printf("watch_count(): thread %ld Count= %d. Going into wait...\n", my_id,count);
    pthread_cond_wait(&count_threshold_cv, &count_mutex);
    printf("watch_count(): thread %ld Condition signal received. Count= %d\n", my_id,count)
    printf("watch_count(): thread %ld Updating the value of count...\n", my_id,count);
    count += 125;
    printf("watch_count(): thread %ld count now = %d.\n", my_id, count);
    }
  printf("watch_count(): thread %ld Unlocking mutex.\n", my_id);
  pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}
```

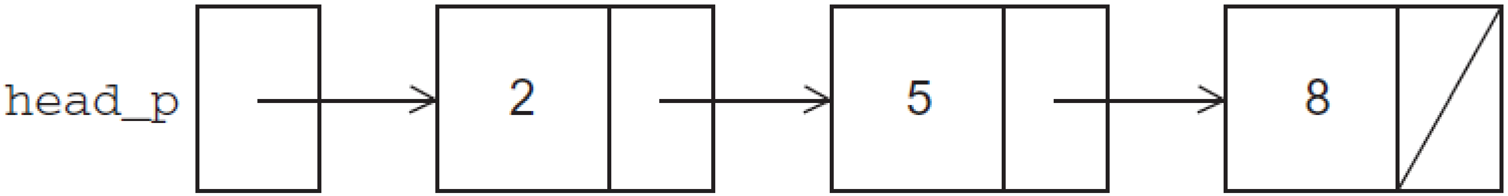
```
void *inc_count(void *t)
{
  int i;
  long my_id = (long)t;
  for (i=0; i < TCOUNT; i++) {
    pthread_mutex_lock(&count_mutex);
    count++;
     if (count == COUNT_LIMIT) {
      printf("inc_count(): thread %ld, count = %d  Threshold reached. ",my_id, count);
      pthread_cond_signal(&count_threshold_cv);
      printf("Just sent signal.\n");
      }
    printf("inc_count(): thread %ld, count = %d, unlocking mutex\n",my_id, count);
    pthread_mutex_unlock(&count_mutex);
    sleep(1);
    }
  pthread_exit(NULL);
}
```

Controlling access to a large, shared data structure

- Let's look at an example.

- Suppose the shared data structure is a sorted linked list of ints, and the operations of interest are Member, Insert, and Delete.

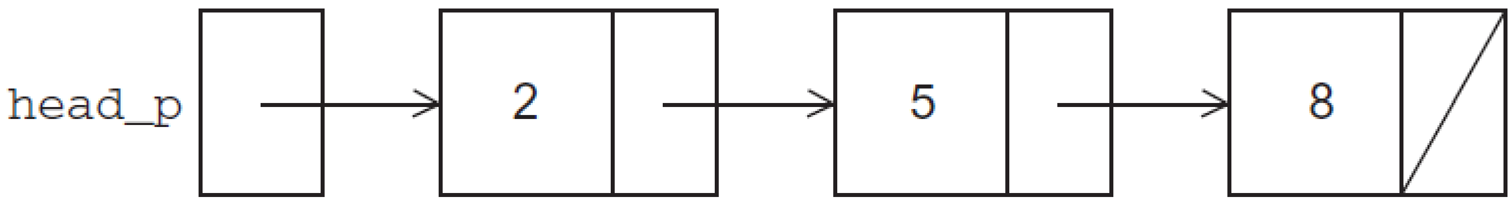
Linked Lists




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

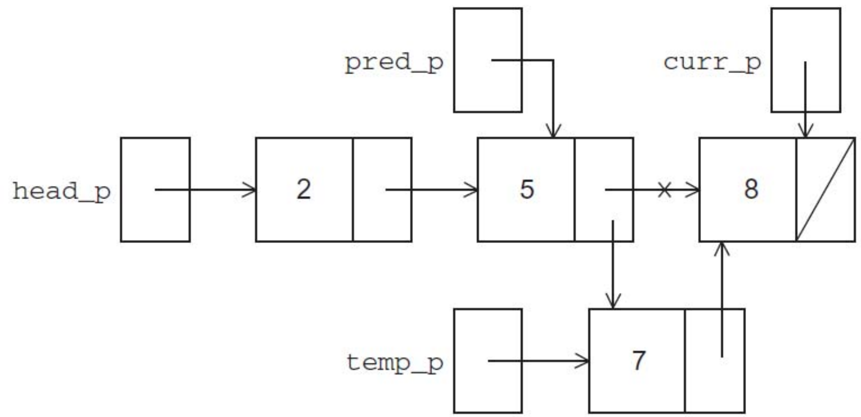
Linked List Membership

```
int  Member(int value, struct list_node_s* head_p) {
   struct list_node_s* curr_p = head_p;

   while (curr_p != NULL && curr_p->data < value)
      curr_p = curr_p->next;

   if (curr_p == NULL || curr_p->data > value) {
      return 0;
   } else {
      return 1;
   }
}  /* Member */
```



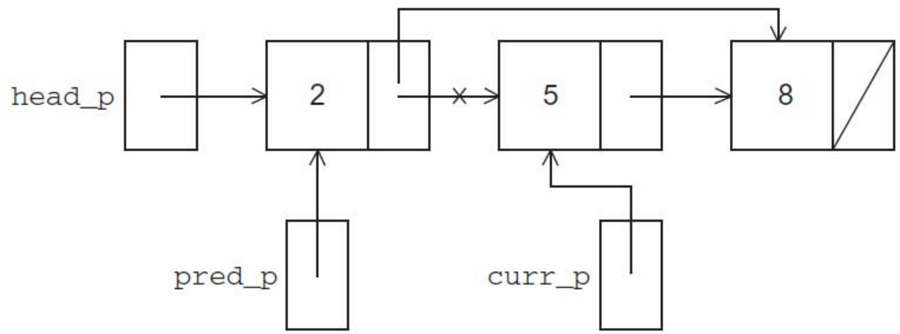
Inserting a new node into a list

```
int Insert(int value, struct list_node_s** head_pp) {
   struct list_node_s* curr_p = *head_pp;
   struct list_node_s* pred_p = NULL;
   struct list_node_s* temp_p;

   while (curr_p != NULL && curr_p->data < value) {
      pred_p = curr_p;
      curr_p = curr_p->next;
   }

   if (curr_p == NULL || curr_p->data > value) {
      temp_p = malloc(sizeof(struct list_node_s));
      temp_p->data = value;
      temp_p->next = curr_p;
      if (pred_p == NULL) /* New first node */
         *head_pp = temp_p;
      else
         pred_p->next = temp_p;
      return 1;
   } else { /* Value already in list */
      return 0;
   }
}  /* Insert */
```



Deleting a node from a linked list

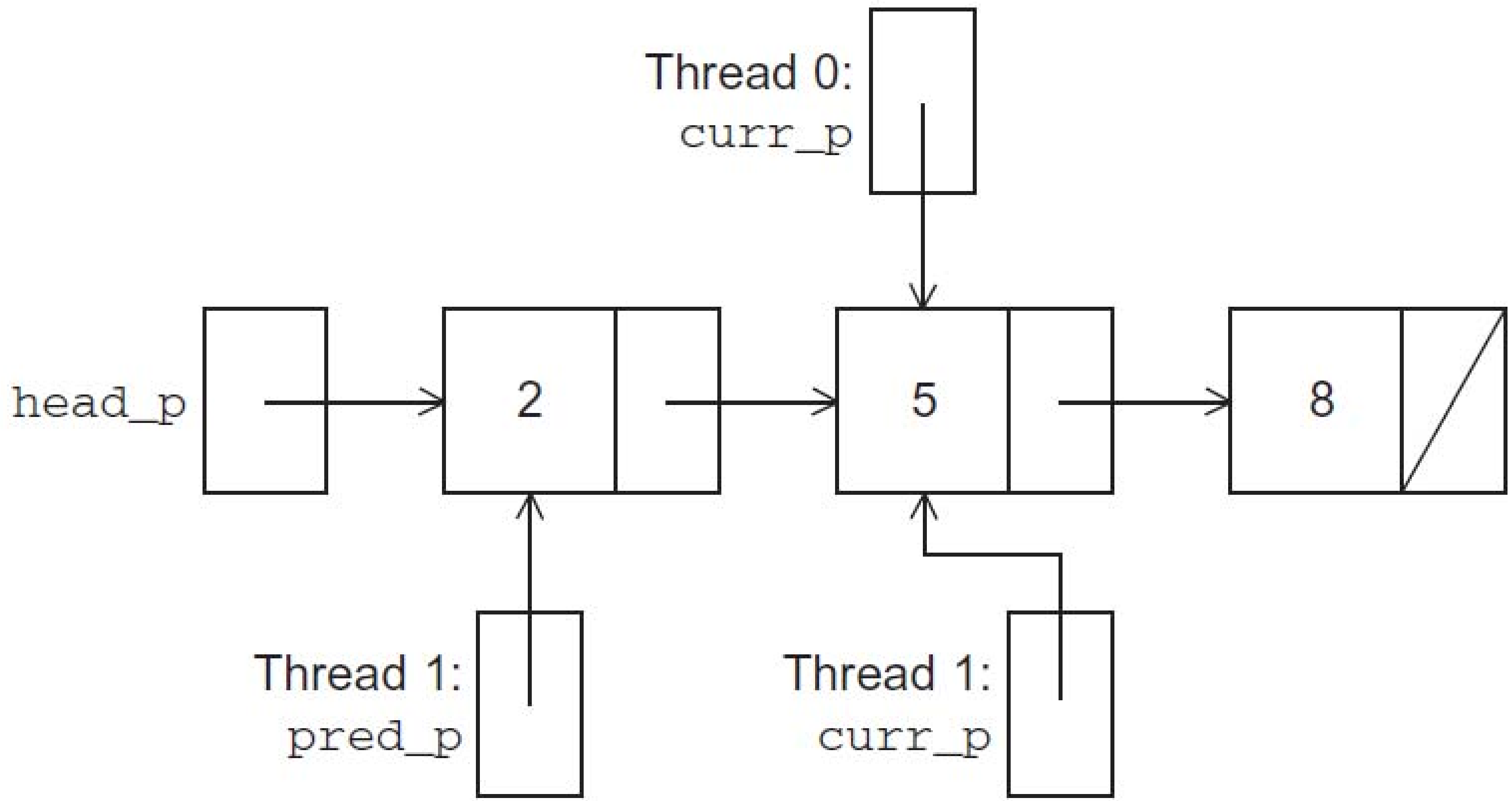
```
int Delete(int value, struct list_node_s** head_pp) {
   struct list_node_s* curr_p = *head_pp;
   struct list_node_s* pred_p = NULL;

   while (curr_p != NULL && curr_p->data < value) {
      pred_p = curr_p;
      curr_p = curr_p->next;
   }

   if (curr_p != NULL && curr_p->data == value) {
      if (pred_p == NULL) { /* Deleting first node in list */
         *head_pp = curr_p->next;
         free(curr_p);
      } else {
         pred_p->next = curr_p->next;
         free(curr_p);
      }
      return 1;
   } else { /* Value isn't in list */
      return 0;
   }
} /* Delete */
```



A Multi-Threaded Linked List

- Let's try to use these functions in a Pthreads program.
- In order to share access to the list, we can define head_p to be a global variable.
- This will simplify the function headers for Member, Insert, and Delete, since we won't need to pass in either head_p or a pointer to head_p: we'll only need to pass in the value of interest.

Simultaneous access by two threads



Solution #1

- An obvious solution is to simply lock the list any time that a thread attempts to access it.
- A call to each of the three functions can be protected by a mutex.

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

In place of calling Member(value).

Issues

- We're serializing access to the list.
- If the vast majority of our operations are calls to Member, we'll fail to exploit this opportunity for parallelism.
- On the other hand, if most of our operations are calls to Insert and Delete, then this may be the best solution since we'll need to serialize access to the list for most of the operations, and this solution will certainly be easy to implement.

Solution #2

- Instead of locking the entire list, we could try to lock individual nodes.
- A “finer-grained” approach.

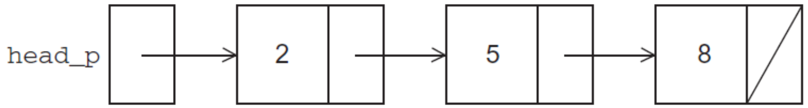
```
struct list_node_s {
    int data;
    struct list_node_s* next;
    pthread_mutex_t mutex;
}
```

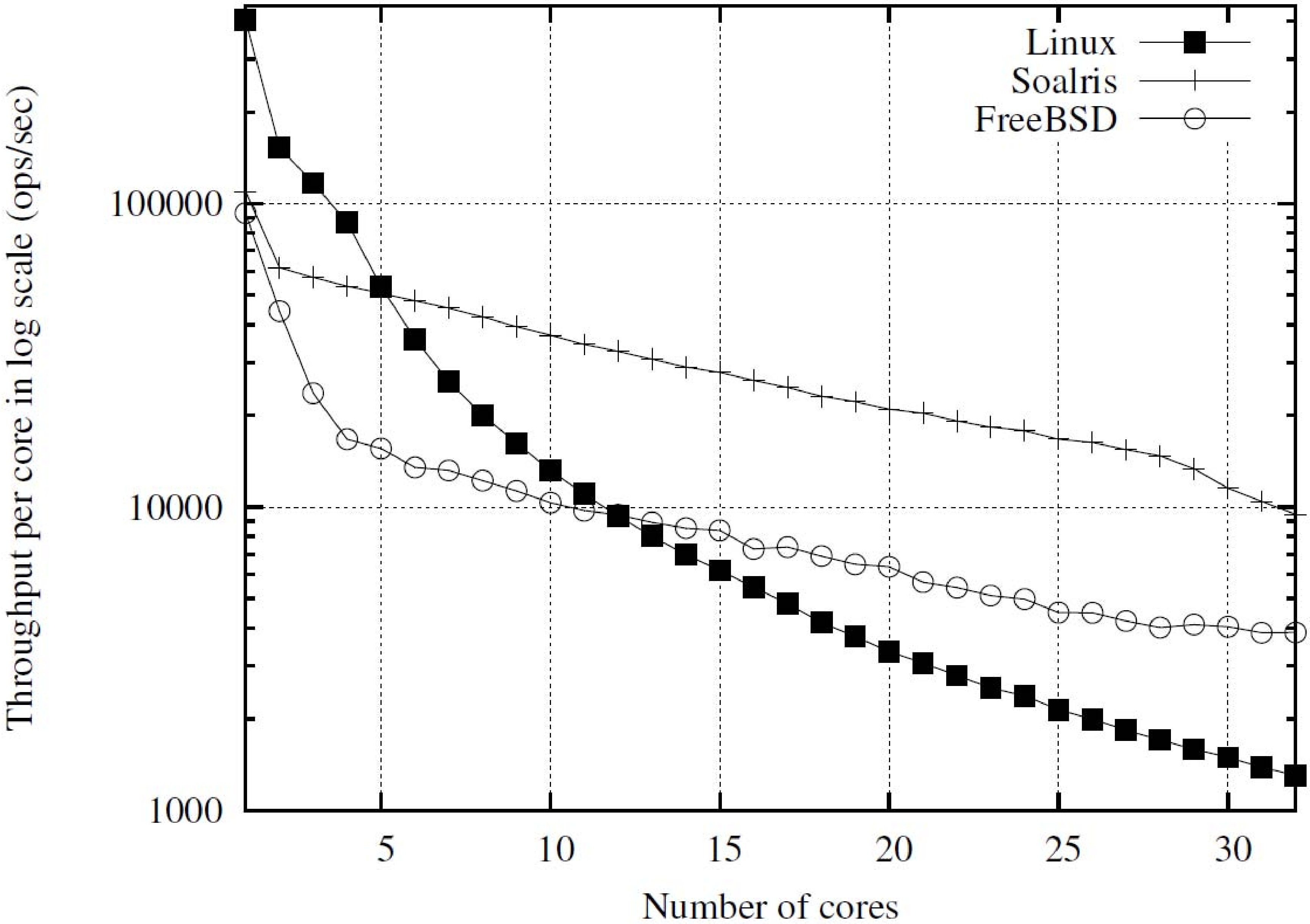
Issues

- This is much more complex than the original Member function.
- It is also much slower, since, in general, each time a node is accessed, a mutex must be locked and unlocked.
- The addition of a mutex field to each node will substantially increase the amount of storage needed for the list.

Implementation of Member with one mutex per list node

```
int  Member(int value) {
   struct list_node_s* temp_p;

   pthread_mutex_lock(&head_p_mutex);
   temp_p = head_p;
   while (temp_p != NULL && temp_p->data < value) {
      if (temp_p->next != NULL)
         pthread_mutex_lock(&(temp_p->next->mutex));
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      pthread_mutex_unlock(&(temp_p->mutex));
      temp_p = temp_p->next;
   }

   if (temp_p == NULL || temp_p->data > value) {
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      if (temp_p != NULL)
         pthread_mutex_unlock(&(temp_p->mutex));
      return 0;
   } else {
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      pthread_mutex_unlock(&(temp_p->mutex));
      return 1;
   }
}  /* Member */
```




Linux
Soalris
FreeBSD
Throughput per core in log scale (ops/sec)
100000
10000
1000
5
10
15
20
25
30
Number of cores

Pthreads Read-Write Locks

- Neither of our multi-threaded linked lists exploits the potential for simultaneous access to any node by threads that are executing Member.
- The first solution only allows one thread to access the entire list at any instant.
- The second only allows one thread to access any given node at any instant.

Pthreads Read-Write Locks

- A read-write lock is somewhat like a mutex except that it provides two lock functions.
- The first lock function locks the read-write lock for reading, while the second locks it for writing.

Pthreads Read-Write Locks

- So multiple threads can simultaneously obtain the lock by calling the read-lock function, while only one thread can obtain the lock by calling the write-lock function.

- Thus, 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.

Pthreads Read-Write Locks

- 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.

Protecting our linked list functions

```
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);
```

Linked List Performance

Implementation	Number of Threads 1	2	4	8
Read-Write Locks	0.213	0.123	0.098	0.115
One Mutex for Entire List	0.211	0.450	0.385	0.457
One Mutex per Node	1.680	5.700	3.450	2.700

100,000 ops/thread

99.9% Member

0.05% Insert

0.05% Delete

Linked List Performance

Implementation	Number of Threads 1	2	4	8
Read-Write Locks	2.48	4.97	4.69	4.71
One Mutex for Entire List	2.50	5.13	5.04	5.11
One Mutex per Node	12.00	29.60	17.00	12.00

100,000 ops/thread

80% Member

10% Insert

10% Delete