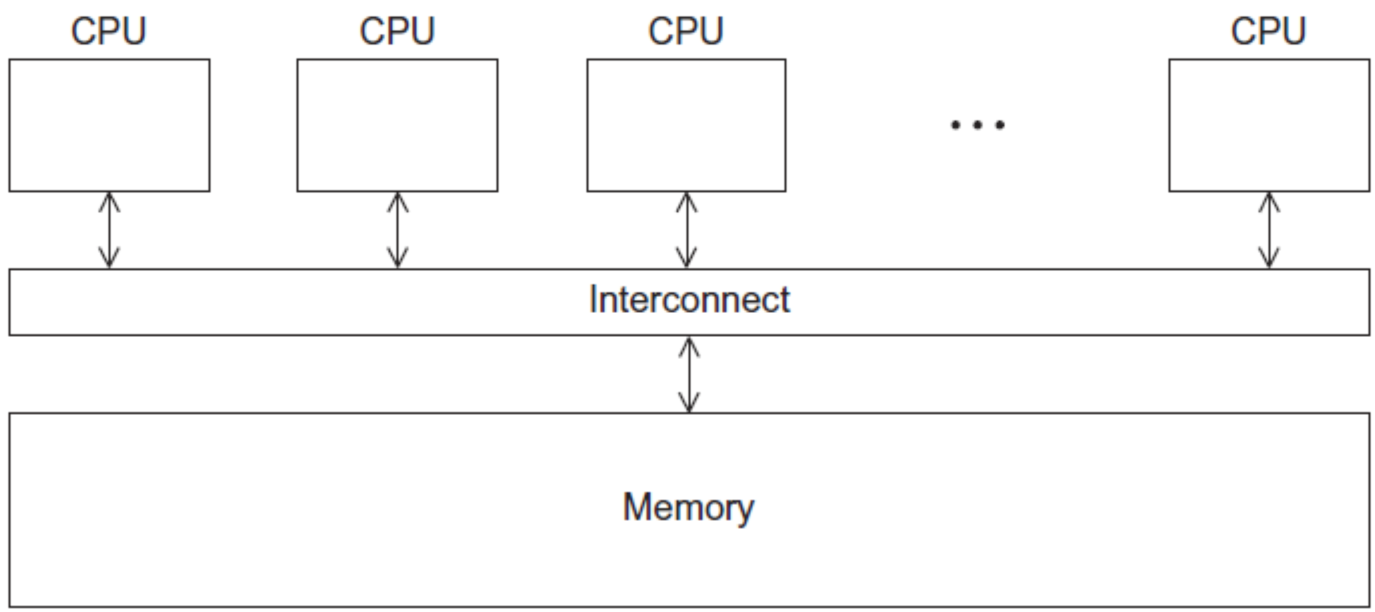
Pthreads: Busy Wait, Mutexes, Semaphores, Conditions, and Barriers

Bryan Mills

Chapter 4.2-4.8

A Shared Memory System



Pthreads

- Pthreads is a POSIX standard for describing a thread model, it specifies the API and the semantics of the calls.
- Model popular –practically all major thread libraries on Unix systems are Pthreads-compatible
- The Pthreads API is only available on POSIXR systems — Linux, MacOS X, Solaris, HPUX, …

Preliminaries

- Include `pthread.h` in the main file
- Compile program with `-lpthread`
  - `gcc -o test test.c -lpthread`
  - may not report compilation errors otherwise but calls will fail
- Check return values on common functions

Include

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

#define NUM_THREADS 10

void *hello (void *rank) {
  printf("Hello Thread”);
}
int main() {
  pthread_t ids[NUM_THREADS];
  for (int i=0; i < NUM_THREADS; i++) {
    pthread_create(&ids[i], NULL, hello, &i);
  }
  for (int i=0; i < NUM_THREADS; i++) {
    pthread_join(ids[i], NULL);
  }
}
```

This includes the pthreads library.

C Libraries

- Libraries are created from many source files, and are either built as:
  - Archive files (libmine.a) that are statically linked.
  - Shared object files (libmine.so) that are dynamically linked
- Link libraries using the gcc command line options -L for the path to the library files and -l to link in a library (a .so or a .a):

```
gcc -o myprog myprog.c  -L/home/bnm/lib -lmine
```

- You may also need to specify the library path:

```
-I /home/bnm/include
```

Runtime Linker

- Libraries that are dynamically linked need to be able to find the shared object (.so)
- **`LD_LIBRARY_PATH`** environment variable is used to tell the system where to find these files.

```
export LD_LIBRARY_PATH=/home/bnm/lib;$LD_LIBRARY_PATH
```

Header Files

- Header files (.h) contain function and macro definitions to be shared with other programs.
- To be used to specify the functions provided by the actual code files (.c, .cc)

Thread creation

- Types: `pthread_t` – type of a thread
- Functions:

```
int pthread_create(pthread_t *thread,
              const pthread_attr_t *attr,
              void * (*start_routine)(void *),
              void *arg);
int pthread_join(pthread_t thread, void **status);
int pthread_detach();
void pthread_exit();
```

  - No explicit parent/child model, except main thread holds process info
  - Call `pthread_exit` in main, don't just fall through;
  - Most likely you wouldn't need `pthread_join`
    - `status` = exit value returned by joinable thread
  - Detached threads are those which cannot be joined (can also set this at creation)

pthread_t objects

- Opaque
- Actual data that they store is system-specific.
- Data members aren't directly accessible to user code.
- However, the Pthreads standard guarantees that a pthread_t object does store enough information to uniquely identify the thread with which it’s associated.

pthread_t object

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

#define NUM_THREADS 10

void *hello (void *rank) {
  printf("Hello Thread");
}
int main() {
  pthread_t ids[NUM_THREADS];
  for (int i=0; i < NUM_THREADS; i++) {
    pthread_create(&ids[i], NULL, hello, &i);
  }
  for (int i=0; i < NUM_THREADS; i++) {
    pthread_join(ids[i], NULL);
  }
}
```

Creating an array of pthread_t objects to hold all the threads we create.

Create Thread

```
int pthread_create (
   pthread_t*  thread_p                      /* out */ ,
   const pthread_attr_t*  attr_p             /* in */ ,
   void*  (*start_func ) ( void )            /* in */ ,
   void*  arg_p                              /* in */ );
```

- thread_p – Will set/return an allocated pthread_t object with information about the created thread.
- attr_p – Used to specify thread properties, NULL = Defaults.
- start_func – Pointer to the function to execute.
- arg_p – Argument to the start_func.
- Returns 0 on success, non-zero on failure.

Attributes

- Type: `pthread_attr_t` (see `pthread_create`)
- Attributes define the state of the new thread
- Attributes: system scope, joinable, stack size, inheritane.
- Use default behaviors with `NULL`.

```
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
pthread_attr_{set/get}{attribute}
```

- Example:

```
pthread_attr_t attr;
pthread_attr_init(&attr); // Needed!!!
pthread_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
pthread_create(NULL, &attr, foo, NULL);
```

Function started by pthread_create

- Prototype:
  void* thread_function (void* args_p) ;

- Void* can be cast to any pointer type in C.

- So args_p can point to a list containing one or more values needed by thread_function.
- Similarly, the return value of thread_function can point to a list of one or more values.

pthread_create

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

#define NUM_THREADS 10

void *hello (void *rank) {
  printf("Hello Thread”);
}
int main() {
  pthread_t ids[NUM_THREADS];
  for (int i=0; i < NUM_THREADS; i++) {
    pthread_create(&ids[i], NULL, hello, &i);
  }
  for (int i=0; i < NUM_THREADS; i++) {
    pthread_join(ids[i], NULL);
  }
}
```

Creating new thread(s).

Keeping track of threads.

Default attributes.

Function to call.

Argument to pass to function to call.

Stopping the Threads

- We call the function pthread_join once for each thread.
- A single call to pthread_join will wait for the thread associated with the pthread_t object to complete.

pthread_join

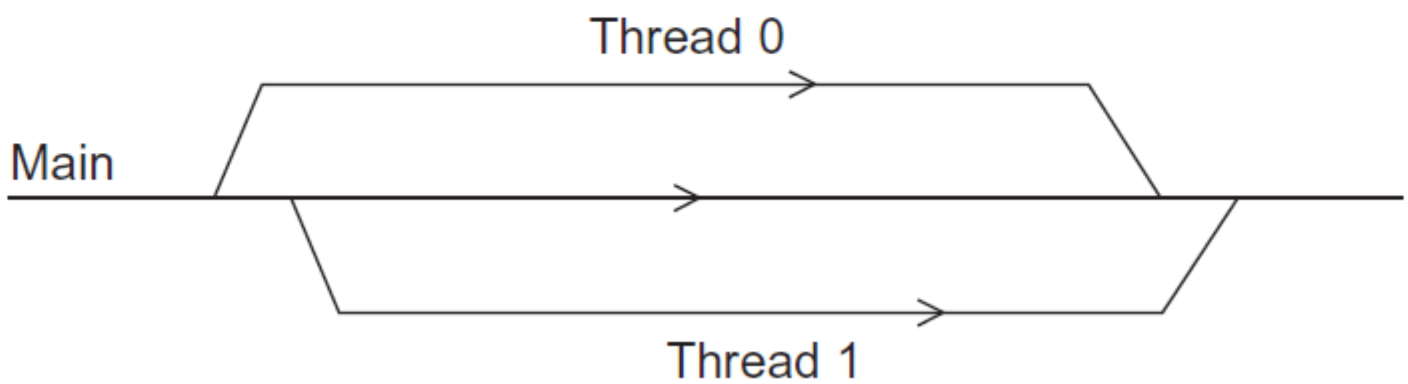
```
#include <stdio.h>
#include <pthread.h>

#define NUM_THREADS 10

void *hello (void *rank) {
  printf("Hello Thread");
}
int main() {
  pthread_t ids[NUM_THREADS];
  for (int i=0; i < NUM_THREADS; i++) {
    pthread_create(&ids[i], NULL, hello, &i);
  }
  for (int i=0; i < NUM_THREADS; i++) {
    pthread_join(ids[i], NULL);
  }
}
```

Called once for each thread. Will wait on thread to finish.

Running the Threads



- Main thread forks and joins two threads.
  - Not really a parent->child relationship
  - Only distinction is that main thread holds system process information.

Contention Scope

- *Contention scope* is the POSIX term for describing bound and unbound threads
- A bound thread is said to have *system contention scope*
  - i.e., it contends with all threads in the system
- An unbound thread has *process contention scope*
  - i.e., it contends with threads in the same process
- For most practical reasons use bound threads (system scope)
  - Solaris LWP switching cheap, Linux is one-to-one anyways…

What’s Wrong?

```
void *hello (void *rank) {
  int *rank_int_ptr = (int*) rank;
  printf("Hello Thread %d\n", *rank_int_ptr);
}
int main() {
  pthread_t ids[NUM_THREADS];
  for (int i=0; i < NUM_THREADS; i++) {
    pthread_create(&ids[i], NULL, hello, &i);
  }
}
```

What `i` will be passed in?

Global variables

- Can introduce subtle and confusing bugs!
- Limit use of global variables to situations in which they're really needed.
  - Shared variables.

$$
\begin{array}{|c|c|c|c|}
\hline
a_{00} & a_{01} & \cdots & a_{0,n-1} \\
\hline
a_{10} & a_{11} & \cdots & a_{1,n-1} \\
\hline
\vdots & \vdots & & \vdots \\
\hline
a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\
\hline
\vdots & \vdots & & \vdots \\
\hline
a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \\
\hline
\end{array}
\quad
\begin{array}{|c|}
\hline
x_0 \\
x_1 \\
\vdots \\
x_{n-1} \\
\hline
\end{array}
=
\begin{array}{|c|}
\hline
y_0 \\
\hline
y_1 \\
\hline
\vdots \\
\hline
y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} \\
\hline
\vdots \\
\hline
y_{m-1} \\
\hline
\end{array}
$$

MATRIX-VECTOR MULTIPLICATION IN PTHREADS

Serial pseudo-code

```
/* For each row of A */
for (i = 0; i < m; i++) {
   y[i] = 0.0;
   /* For each element of the row and each element of x */
   for (j = 0; j < n; j++)
      y[i] += A[i][j]* x[j];
}
```

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

Using Pthreads

Thread	Components of y
0	y[0], y[1]
1	y[2], y[3]
2	y[4], y[5]

thread 0

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

Thread i

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

Pthreads matrix-vector multiplication

```
void *nth_mat_vect(void *rank) {
  int *rank_int_ptr = (int*) rank;
  int my_rank = *rank_int_ptr;
  int local_m = M / NUM_THREADS;
  int my_first_row = (my_rank * local_m);
  int my_last_row = (my_rank+1)*local_m - 1;

  for (int i = my_first_row; i <= my_last_row; i++) {
    Y[i] = 0.0;
    for (int j = 0; j < N; j++) {
      Y[i] += A[i][j] * X[j];
    }
  }
  return NULL;
}
```

CRITICAL SECTIONS

Estimating π

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

```
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;
```

A thread function for computing π

```
void *calc(void *rank) {
  int *rank_int_ptr = (int*) rank;
  int my_rank = *rank_int_ptr;
  int my_n = N / NUM_THREADS;
  int my_first_i = (my_rank * my_n);
  int my_last_i = my_first_i + my_n;
  double factor = -1.0;
  if (my_first_i % 2 == 0) {
    factor = 1.0;
  }
  for (int i = my_first_i; i <= my_last_i; i++) {
    sum += factor / (2*i+1);
    factor = -1 * factor;
  }
  return NULL;
}
```

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.

Why?

Possible race condition

```
int x = 0;
void *compute(void *rank) {
  int *rank_int_ptr = (int*) rank;
  int y = *rank_int_ptr;
  x += y;
}
```

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

Busy-Waiting

- A thread repeatedly tests a condition, but, effectively, does no useful work until the condition has the appropriate value.

- Beware of optimizing compilers, though!

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

flag initialized to 0 by main thread

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 */
```

Busy-waiting sum

```
void *calc(void *rank) {
  int *rank_int_ptr = (int*) rank;
  int my_rank = *rank_int_ptr;
  int my_n = N / NUM_THREADS;
  int my_first_i = (my_rank * my_n);
  int my_last_i = my_first_i + my_n;
  double factor = -1.0;
  if (my_first_i % 2 == 0) {
    factor = 1.0;
  }
  for (int i = my_first_i; i <= my_last_i; i++) {
    while (flag != my_rank);
    sum += factor / (2*i+1);
    flag = (flag+1) % NUM_THREADS;
    factor = -1 * factor;
  }
  return NULL;
}
```

Time?

With no barriers – 2 threads
pi=3.134463

real 0m0.739s
user 0m1.445s
sys 0m0.012s

With no barriers – 3 threads
pi=3.146096

real 0m0.647s
user 0m1.792s
sys 0m0.006s

With busy-wait – 2 threads
pi=3.141593

real 0m4.509s
user 0m8.815s
sys 0m0.053s

With no barriers – 3 threads
pi=3.141593

real 0m6.872s
user 0m19.383s
sys 0m0.056s

Busy-waiting sum at end

```
void *calc(void *rank) {
  int *rank_int_ptr = (int*) rank;
  int my_rank = *rank_int_ptr;
  int my_n = N / NUM_THREADS;
  int my_first_i = (my_rank * my_n);
  int my_last_i = my_first_i + my_n;
  double factor = -1.0;
  if (my_first_i % 2 == 0) {
    factor = 1.0;
  }
  double my_sum = 0.0;
  for (int i = my_first_i; i <= my_last_i; i++) {
    my_sum += factor / (2*i+1);
    factor = -1 * factor;
  }
  while (flag != my_rank);
  sum += my_sum;
  flag = (flag+1) % NUM_THREADS;
  return NULL;
}
```

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.

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.

Pthread Mutexes

- Type: `pthread_mutex_t`

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

- Important: Mutex scope must be visible to all threads!

Busy-waiting sum at end

```
void *calc(void *rank) {
  int *rank_int_ptr = (int*) rank;
  int my_rank = *rank_int_ptr;
  int my_n = N / NUM_THREADS;
  int my_first_i = (my_rank * my_n);
  int my_last_i = my_first_i + my_n;
  double factor = -1.0;
  if (my_first_i % 2 == 0) {
    factor = 1.0;
  }
  double my_sum = 0.0;
  for (int i = my_first_i; i <= my_last_i; i++) {
    my_sum += factor / (2*i+1);
    factor = -1 * factor;
  }
  pthread_mutex_lock(&mutex);
  sum += my_sum;
  pthread_mutex_unlock(&mutex);
  return NULL;
}
```

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

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

Run-times (in seconds) of π programs using n = 108 terms on a system with two four-core processors.

Time	`flag`	Thread				
		0	1	2	3	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.
  - Producer/Consumer (message sending)

Semaphores

- Similar to mutexes but there isn't necessarily ownership of "data", but coordination.
- Think of it as a counter:
  - When counter = 0 its locked (wait)
  - When counter > 0 its unlocked (proceed)
- Two main functions:
  - Wait – decrements the counter if > 0 returns; if <=0 then waits
  - Post – increments the counter
- Not provided natively using pthreads, although fairly straight forward to implement.

Message Sending Example

```
Void * send_msg(void* rank) {
  int *rank_int_ptr = (int*) rank;
  int my_rank = *rank_int_ ptr;
  int dest = my_rank + 1 % NUM_THREADS;
  char* my_msg = malloc(MSG_MAX*sizeof(char));
  sprintf(my_msg, "Hello from %d", my_rank);

  messages[dest] = my_msg;
  sem_post(&semaphores[dest]);

  sem_wait(&semaphores[my_rank])
  printf("Thread %d> %s\n", my_rank, messages[my_rank]);

  return NULL;
}
```

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.

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.

Why Barriers

- Synchronization at algorithm steps.
  - Iterations in linear solver, check to see CI is good.
- Write state to disk
  - Simulated wind tunnel, write image.
  - Checkpointing
- Debugging
  - Print state of matrixes at different steps
- Error Checking
  - Verify algorithm is converging

Barriers with Mutex

```
void* Thread_work( . . . ) {
   . . .

   /* START Barrier */
   pthread_mutex_lock(&barrier_mutex);
   counter++;
   pthread_mutex_unlock(&barrier_mutex);
   while (counter < thread_count);
   /* END Barrier */

   . . .
}
```

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

- Type `pthread_cond_t`

```
int pthread_cond_init(pthread_cond_t *cond,
                      const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond,
             pthread_mutex_t *mutex);
int pthread_cond_singal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Barrier With Condition

```
void* Thread_work( . . . ) {
  . . .

  /* START 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);
  /* END Barrier */

  . . .
}
```

Some rules…

- Shared data should always be accessed through a single mutex
- Think of a boolean condition (expressed in terms of program variables) for each condition variable. Every time the value of the boolean condition may have changed, call `Broadcast` for the condition variable
  - Only call `Signal` when you are absolutely certain that *any and only one* waiting thread can enter the critical section
  - if noone is waiting, signal is lost
- Signaling/Broadcasting thread need not have the mutex
  - may be more efficient to release it first…
- Globally order locks, acquire in order in all threads

READ-WRITE LOCKS

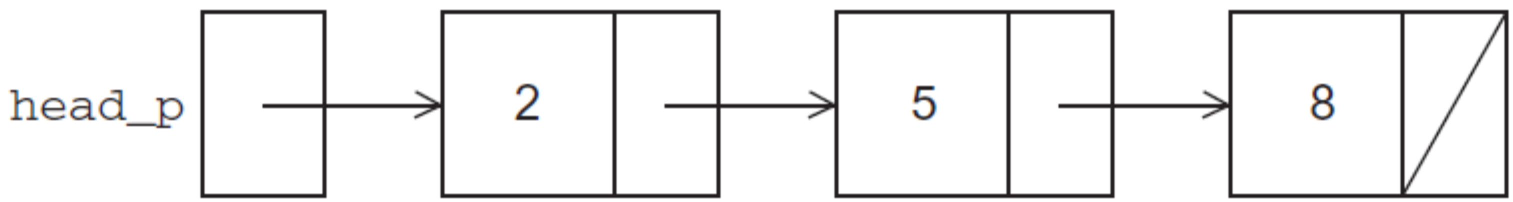
SECTION 4.9

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




```c
/* Struct for list nodes */
struct list_node_s {
   int    data;
   struct list_node_s* next;
};
```

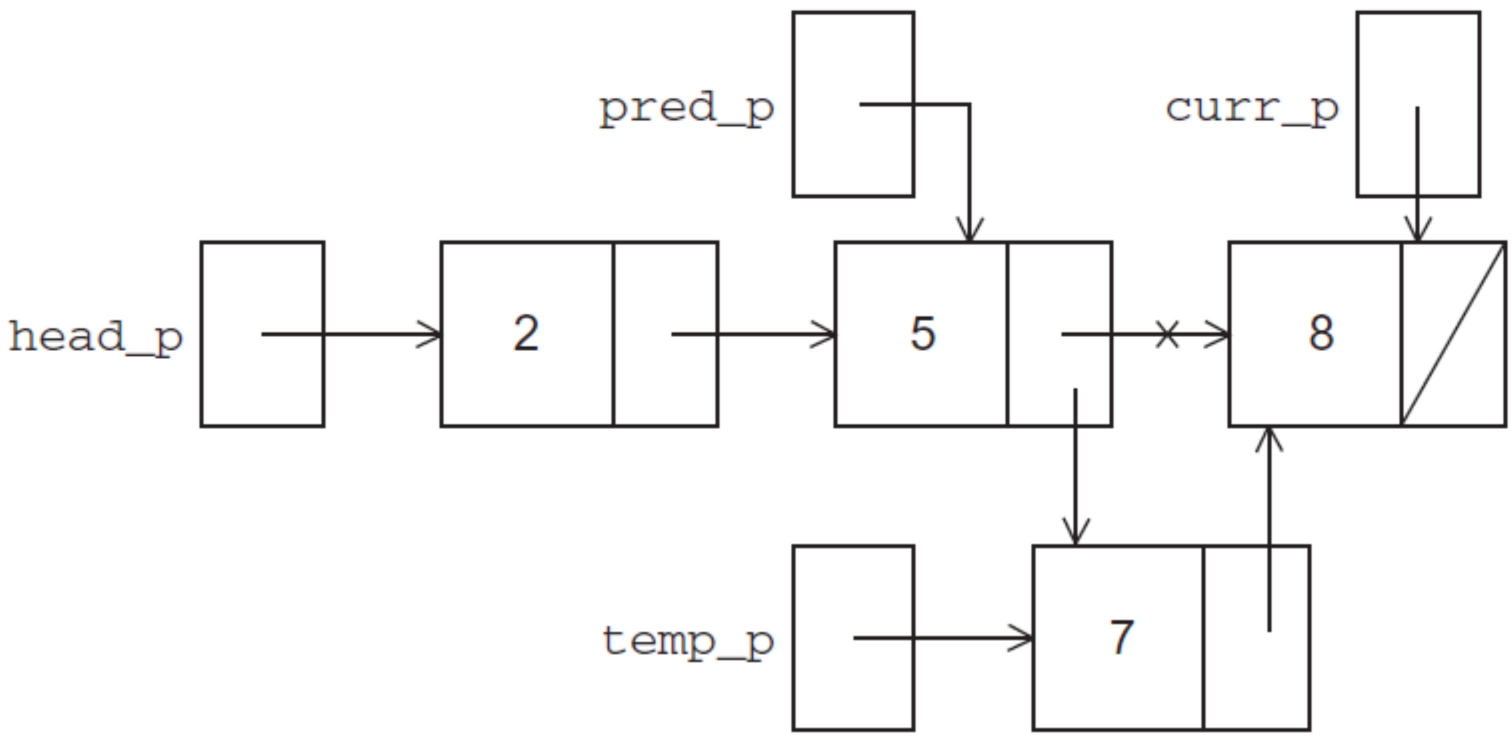
Linked List Membership

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

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

   if (temp == NULL || temp->data > value) {
     return 0;
   } else {
     return 1;
   }
}
```

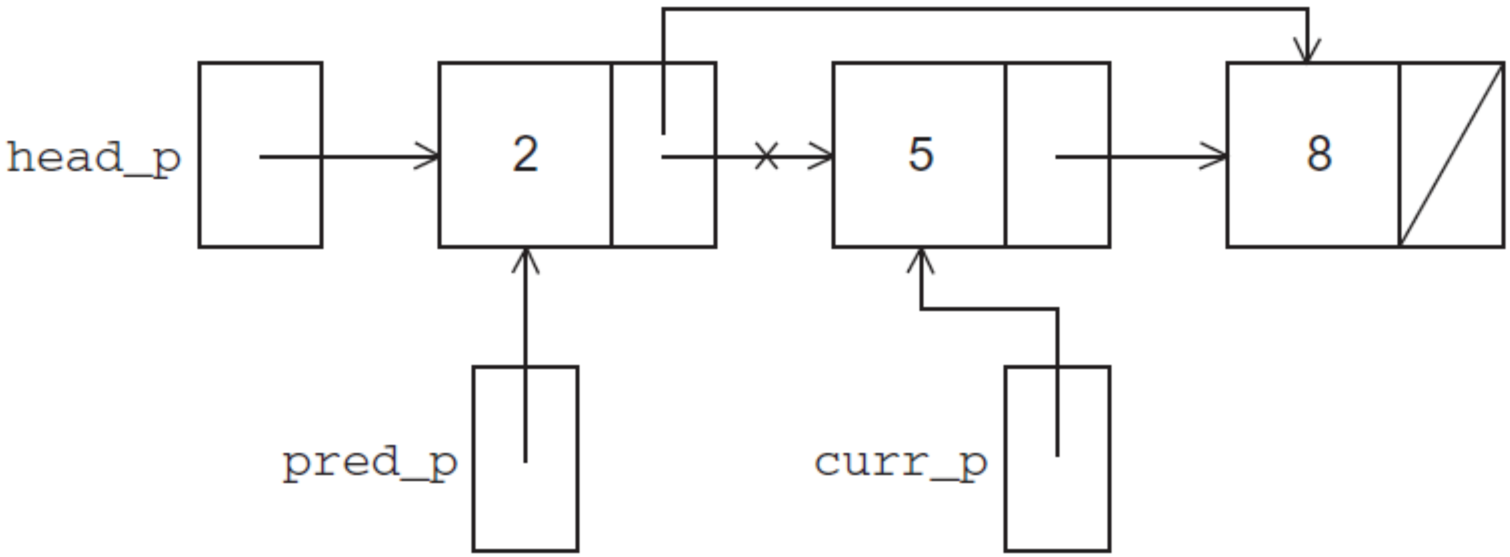
Inserting a new node into a list



Inserting a new node into a list

```
int Insert(int value) {
   struct list_node_s* curr = head;
   struct list_node_s* pred = NULL;
   struct list_node_s* temp;
   while (curr != NULL && curr->data < value) {
      pred = curr;
      curr = curr->next;
   }
   if (curr == NULL || curr->data > value) {
      temp = malloc(sizeof(struct list_node_s));
      temp->data = value;
      temp->next = curr;
      if (pred == NULL)
         head = temp;
      else
         pred->next = temp;
   } else { /* value in list */
      return 0;
   }
}
```

Deleting a node from a linked list



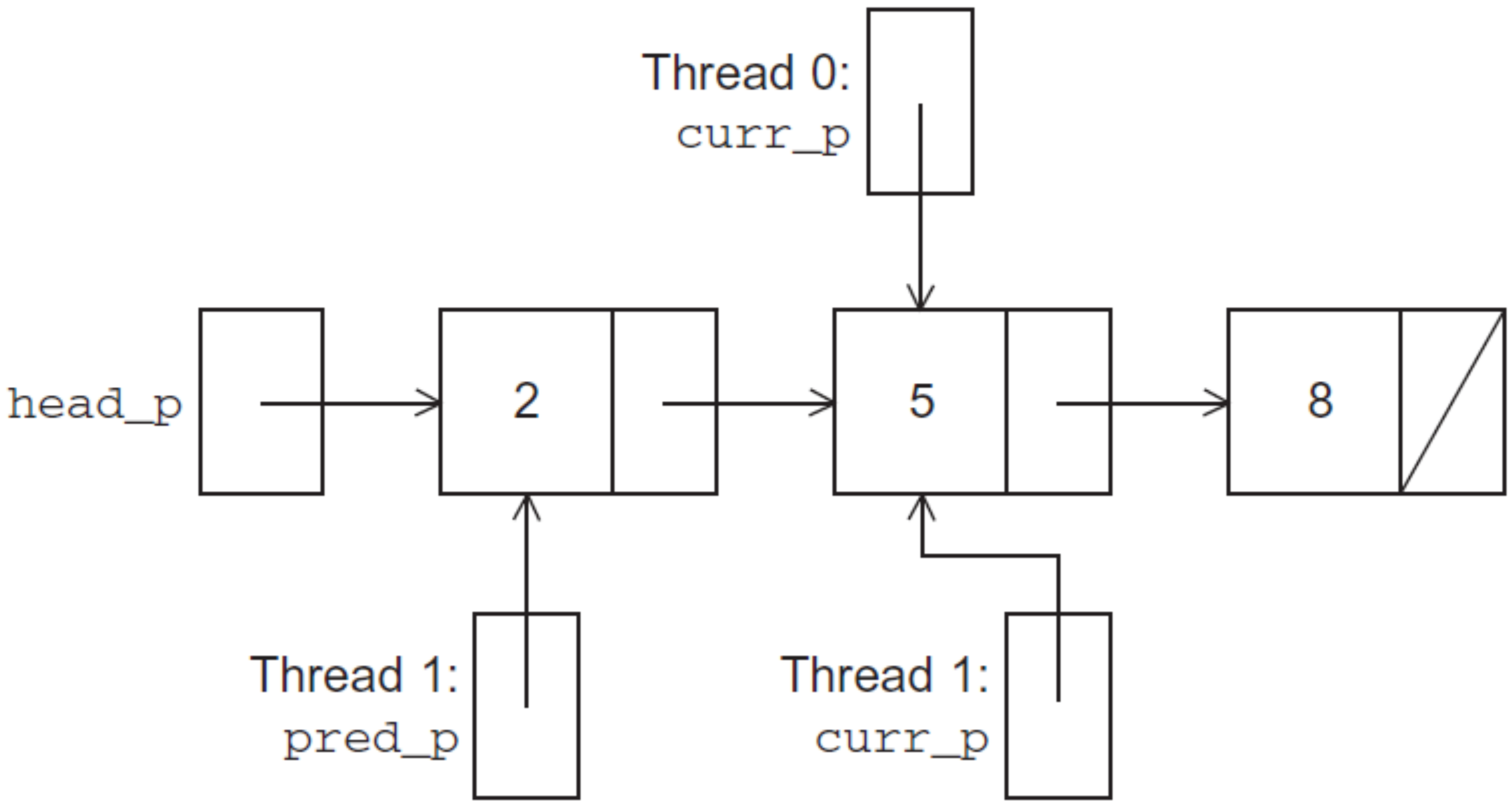
Deleting a node from a linked list

```
int Delete(int value) {
   struct list_node_s* curr = head;
   struct list_node_s* pred = NULL;
   int rv = 1;
   /* Find value */
   while (curr != NULL && curr->data < value) {
      pred = curr;
      curr = curr->next;
   }
   if (curr != NULL && curr->data == value) {
      if (pred == NULL) { /* first element in list */
         head = curr->next;
         free(curr);
      } else {
         pred->next = curr->next;
         free(curr);
      }
      return 1;
   } else { /* Not in list */
      return 0;
   }
}
```

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);
```

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 for list nodes */
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 (1)

```
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;
   }
```

Implementation of Member with one mutex per list node (2)

```
   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 */
```

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.
- 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.
- 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

```
pthread_rwlock_rdlock(&rwlock);
Member(val);
pthread_rwlock_unlock(&rwlock);

pthread_rwlock_wrlock(&rwlock);
Insert(val);
pthread_rwlock_unlock(&rwlock);

pthread_rwlock_wrlock(&rwlock);
Delete(val);
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

THREAD-SAFETY

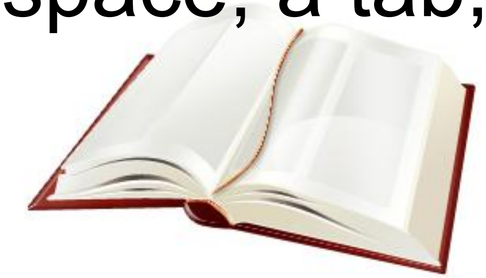
SECTION 4.11

Thread-Safety

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

Example

- Suppose we want to use multiple threads to “tokenize” a file that consists of ordinary English text.
- The tokens are just contiguous sequences of characters separated from the rest of the text by white-space — a space, a tab, or a newline.



Simple approach

- Divide the input file into lines of text and assign the lines to the threads in a round-robin fashion.

- The first line goes to thread 0, the second goes to thread 1, . . . , the tth goes to thread t, the t +1st goes to thread 0, etc.

Simple approach

- We can serialize access to the lines of input using mutexes.

- After a thread has read a single line of input, it can tokenize the line using the strtok function.

The strtok function

- The first time it's called the string argument should be the text to be tokenized.
  - Our line of input.
- For subsequent calls, the first argument should be NULL.

```
char * strtok (char* string,
               const char* seperators)
```

The strtok function

- The idea is that in the first call, strtok caches a pointer to string, and for subsequent calls it returns successive tokens <u>taken from the cached copy</u>.

Multi-threaded tokenizer

```
void *Tokenize(void* rank) {
   . . .
   /* Force sequential reading of the input */
   pthread_mutex_lock(&mutex);
   fg_rv = fgets(my_line, MAX, fp);
   pthread_mutex_unlock(&mutex);
   while (fg_rv != NULL) {
      printf("Thread %d > my line = %s", my_rank, my_line);
      count = 0;
      my_string = strtok(my_line, " \t\n");
      while ( my_string != NULL ) {
         count++;
         printf("Thread %d > string %d = %s\n", my_rank, count, my
         my_string = strtok(NULL, " \t\n");
      }
      printf("Thread %d > After tokenizing, my_line = %s\n", my_ra
      pthread_mutex_lock(&mutex);
      fg_rv = fgets(my_line, MAX, fp);
      pthread_mutex_unlock(&mutex);
   }
. . .
}
```

What happened?

- strtok caches the input line by declaring a variable to have static storage class.
- This causes the value stored in this variable to persist from one call to the next.
- Unfortunately for us, this cached string is shared, not private.

What happened?

- Thus, thread 0's call to strtok with the third line of the input has apparently overwritten the contents of thread 1's call with the second line.
- So the strtok function is not thread-safe. If multiple threads call it simultaneously, the output may not be correct.

Other unsafe C library functions

- Regrettably, it's not uncommon for C library functions to fail to be thread-safe.
- The random number generator random in stdlib.h.
- The time conversion function localtime in time.h.

“re-entrant” functions

- In some cases, the C standard specifies an alternate, thread-safe, version of a function.

```
char * strtok (char* string,
               const char* seperators,
               char** saveptr_p)
```

- Not all re-entrant functions are threadsafe!

Python and GIL

- Python using a **global interpreter lock**
- This means that any python code can't truly be executed in parallel
  - Although you can use threads in the c code and explicitly unlock other processes
  - Basic I/O functions do this by default, therefore mutliple python threads doing I/O do not block one another (ie webserver, file reads)
- Other implementations of python do not have this constraint (JPython)
- Ruby has some similar constraints

Concluding Remarks (1)

- A thread in shared-memory programming is analogous to a process in distributed memory programming.
- However, a thread is often lighter-weight than a full-fledged process.
- In Pthreads programs, all the threads have access to global variables, while local variables usually are private to the thread running the function.

Concluding Remarks (2)

- When indeterminacy results from multiple threads attempting to access a shared resource such as a shared variable or a shared file, at least one of the accesses is an update, and the accesses can result in an error, we have a race condition.

Concluding Remarks (3)

- A critical section is a block of code that updates a shared resource that can only be updated by one thread at a time.
- So the execution of code in a critical section should, effectively, be executed as serial code.

Concluding Remarks (4)

- Busy-waiting can be used to avoid conflicting access to critical sections with a flag variable and a while-loop with an empty body.
- It can be very wasteful of CPU cycles.
- It can also be unreliable if compiler optimization is turned on.

Concluding Remarks (5)

- A mutex can be used to avoid conflicting access to critical sections as well.
- Think of it as a lock on a critical section, since mutexes arrange for mutually exclusive access to a critical section.

Concluding Remarks (6)

- A semaphore is the third way to avoid conflicting access to critical sections.
- It is an unsigned int together with two operations: sem_wait and sem_post.
- Semaphores are more powerful than mutexes since they can be initialized to any nonnegative value.

Concluding Remarks (7)

- A barrier is a point in a program at which the threads block until all of the threads have reached it.
- A read-write lock is used when it's safe for multiple threads to simultaneously read a data structure, but if a thread needs to modify or write to the data structure, then only that thread can access the data structure during the modification.

Concluding Remarks (8)

- Some C functions cache data between calls by declaring variables to be static, causing errors when multiple threads call the function.
- This type of function is not thread-safe.