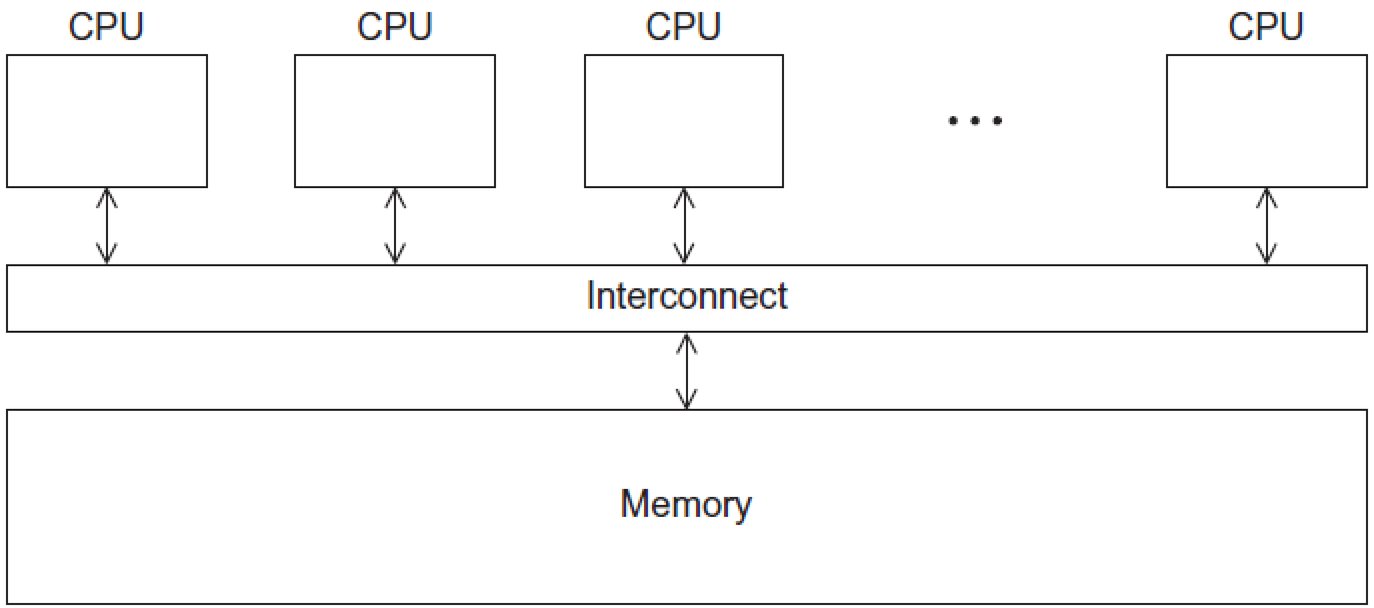
Shared Memory Programming with Pthreads

Jinkyu Jeong

Roadmap

- Problems programming shared memory systems.
- Controlling access to a critical section.
- Thread synchronization.
- Programming with POSIX threads.
- Mutexes.
- Producer-consumer synchronization and semaphores.
- Barriers and condition variables.
- Read-write locks.
- Thread safety.

A Shared Memory System



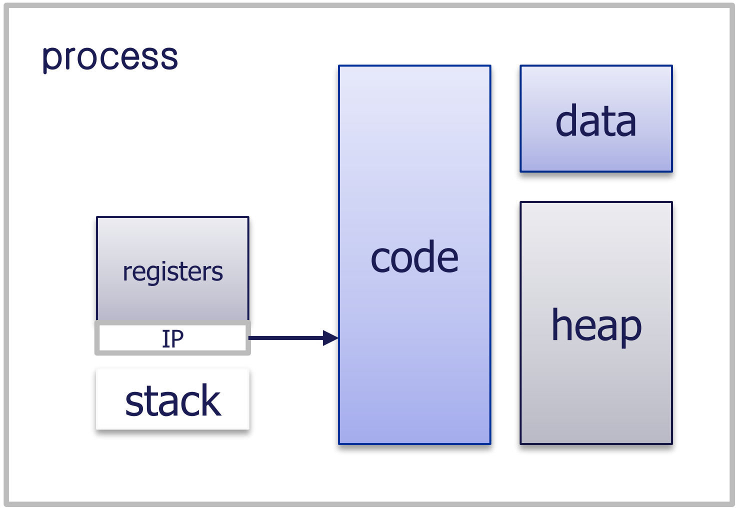
Shared Memory Programming

- **Multiple threads (processes) on shared address space**
  - More convenient programming model
  - Careful control required when shared data are accessed

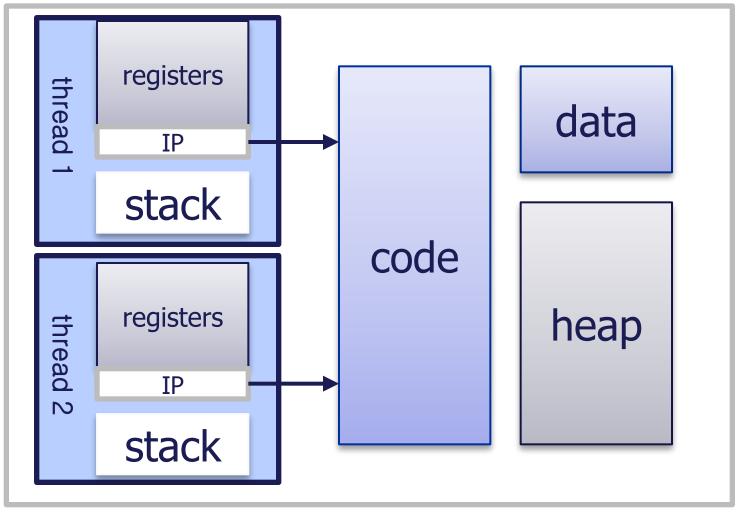
- **Programming models**
  - Threads libraries (classes): Pthreads, Java threads
  - New programming languages: Ada
  - Modifying syntax of existing languages: UPC (Berkeley Unified Parallel C), Cilk, C++ 11
  - Compiler directives: OpenMP

Threads vs. Processes

- **Process**
  - One address space per process
  - Each process has its own data (global variables), stack, heap



- **Thread**
  - Multiple threads share on address space
    - But its own stack and register context
  - Threads within the same address space share data (global variables), heap



POSIX® Threads

- **Also known as Pthreads.**
- **A standard for Unix-like operating systems.**
  - Created by IEEE
  - Called POSIX 1003.1c in 1995
- **A library that can be linked with C programs.**
- **Specifies an application programming interface (API) for multi-threaded programming.**

Caveat

- The Pthreads API is available on many Unix-like POSIX-conformant operating systems — Linux, MacOS X, Solaris, HPUX, …
- SFU/SUA subsystem on Micrsoft Windows implements many POSIX APIs

Hello World! (1)

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

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

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;

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

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

declares the various Pthreads functions, constants, types, etc.

Hello World! (2)

```
    for (thread = 0; thread < thread_count; thread++)
        pthread_create(&thread_handles[thread], NULL,
            Hello, (void*) 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;
}  /* main */
```

Hello World! (3)

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

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

   return NULL;
}  /* Hello */
```

Compiling a Pthread program

gcc −g −Wall −o pth_hello pth_hello . c −lpthread

link in the Pthreads library

Running a Pthreads program

. / pth_hello <number of threads>

. / pth_hello 1

Hello from the main thread
Hello from thread 0 of 1

. / pth_hello 4

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

Starting the Threads

pthread.h

One object for each thread.

pthread_t

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

pthread_t objects

- Opaque
- The actual data that they store is system-specific.
- Their 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.

A closer look (1)

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

We won’t be using, so we just pass NULL.

Allocate <u>before</u> calling.

A closer look (2)

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

Pointer to the argument that should be passed to the function *start_routine*.

The function that the thread is to run.

Function started by thread_create

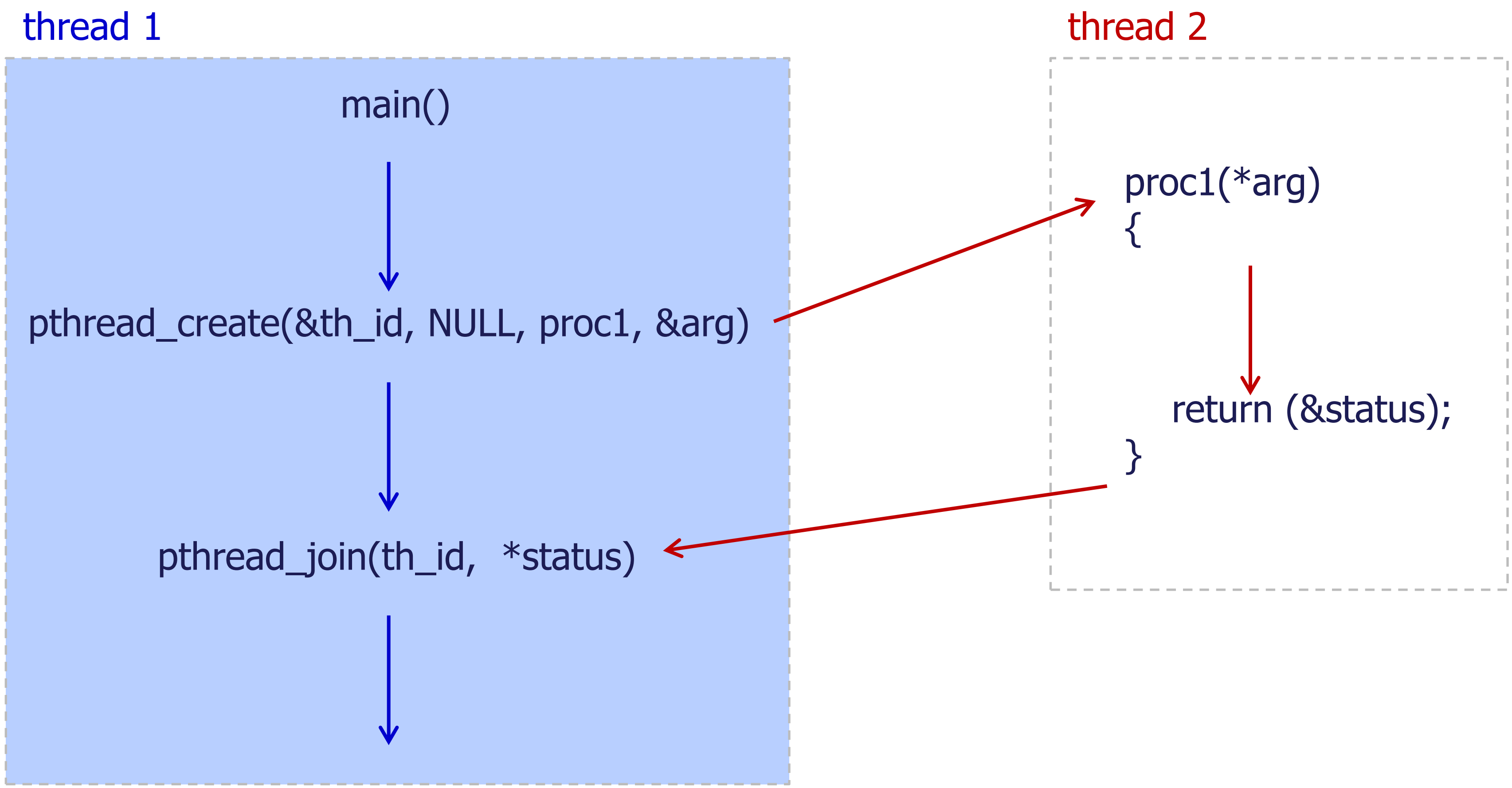
- **Prototype:**
  void* thread_function (void* args_p) ;

- **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.**

Pthreads – creation & join

- **pthread_create, pthread_join**



Pthreads – detached thread

- **pthread_attr_setdetachstate**

main()

pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED)

pthread_create(&th_id1, &attr, proc1, NULL)

proc1 {

}

terminates!

pthread_create(&th_id2, &attr, proc2, NULL)

proc2 {

}

terminates!

Pthreads –Exiting a Thread

- **4 ways to exit threads**
  - Thread will naturally exit after starting thread function returns
  - Thread itself can exit by calling pthread_exit()
  - Other threads can terminate a thread by calling pthread_cancel()
  - Thread exits if the process that owns the thread exits

- **APIs**
  - void pthread_exit (void *retval);
  - int pthread_cancel (pthread_t thread)

CRITICAL SECTIONS

Estimating pi

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

```
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 pi

```
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 */
```

Access to a shared variable → race condition → nondeterminism

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 more correct

❖ Results on my system (Intel i7 920, 4 cores, 2 SMTs per core)

# of threads	10^7	10^8
1	3.141593	3.141593
2	2.663542	-0.00811
4	3.166491	0.058198
8	-1.25512	3.360151

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`

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 */
```

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 */
```

Pthread Spinlock

- **The example busywaiting lock enforces the sequence of threads entering the critical section**
  - Thread 0 → Thread 1 → Thread 2 → …
  - If a lock-holder is de-scheduled, successive lock waiters wastes a lot of CPU cycles

- **Pthread library provides spinlock-based mutual exclusion**
  - pthread_spinlock_t
  - pthread_spin_init(pthread_spinlock_t* spinlock, int nr_shared)
  - pthread_spin_lock(pthread_spinlock_t* spinlock)
  - pthread_spin_unlock(pthread_spinlock_t* spinlock)

Global sum function that uses a spinlock

```
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_spin_lock(&spinlock);
   sum += my_sum;
   pthread_spin_unlock(&spinlock);

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

Mutexes

- A thread that is busy-waiting may continually use the CPU accomplishing nothing.
  - pthread_spinlock could still waste CPU cycles when a lock holder is de-scheduled
- 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.
- Waiters to enter the critical sleeps until the only lock holder releases exits the critical section
  - Avoids wasting of CPU 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.

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

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

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

Mutexes

- 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 */);
```

- Use of a mutex

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
/* or pthread_mutex_init(&lock, NULL)

pthread_mutex_lock( &lock );
   // critical section
pthread_mutext_unlock( &lock );
```

Mutexes

- **Specifying attribute of a mutex**

```
pthread_mutex_t lock;
Pthread_mutexattr_t attr;

pthread_mutexattr_init(&attr);
pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_FAST_NP)
pthread_mutex_init(&lock, &attr)

pthread_mutex_lock( &lock );
   // critical section
pthread_mutext_unlock( &lock );
```

- **Attributes**
  - PTHREAD_MUTEX_FAST_NP
  - PTHREAD_MUTEX_RECURSIVE_NP
  - PTHREAD_MUTEX_ERRORCHECK_NP

Non-portable to other systems

Global sum function that uses a mutex

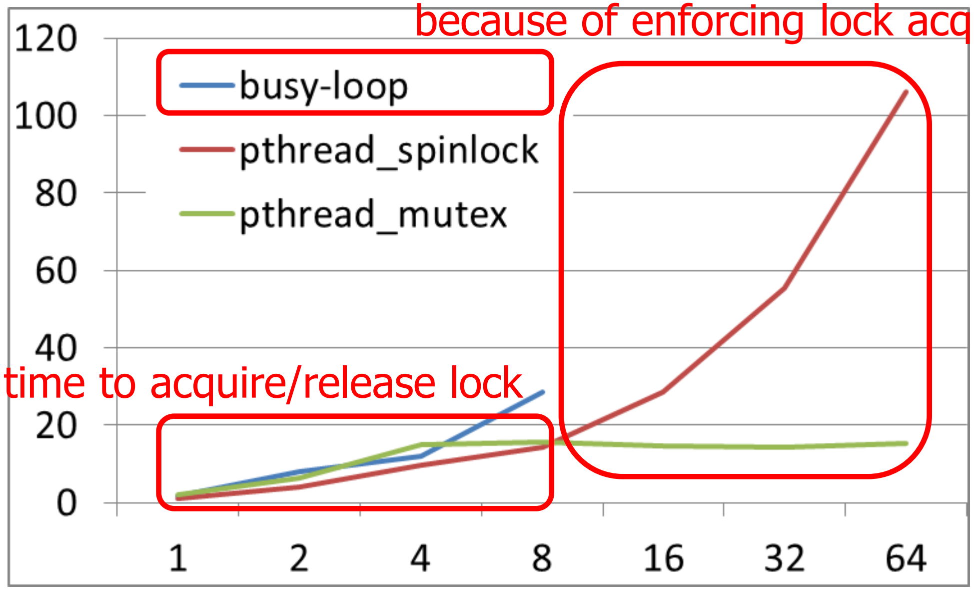
```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;
   double my_sum = 0.0;

   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 evaluation

- Busy-loop vs. pthread_spinlock vs. pthread_mutex
- Lock/unlock for each global sum variable update
- Environment
  - Intel i7 4 cores + 2 SMT per core (8 logicla cores)

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

Busy-loop is worse than spinlock because of enforcing lock acquisition sequence



Spinlock wastes CPU cycles when a lock holder is preempted

Mutex takes longer time to acquire/release lock

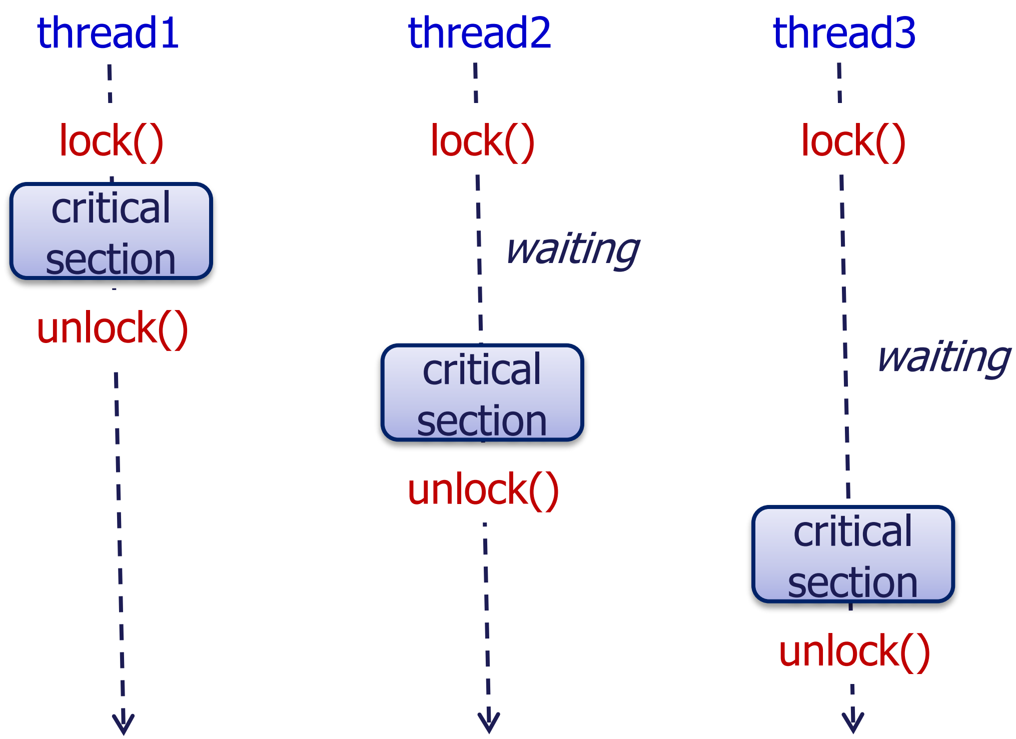
Performance evaluation

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.
- Because the busy-loop implementation enforces the sequence of lock acquisition
  - Thread 0 → Thread 1 → Thread 2 → …

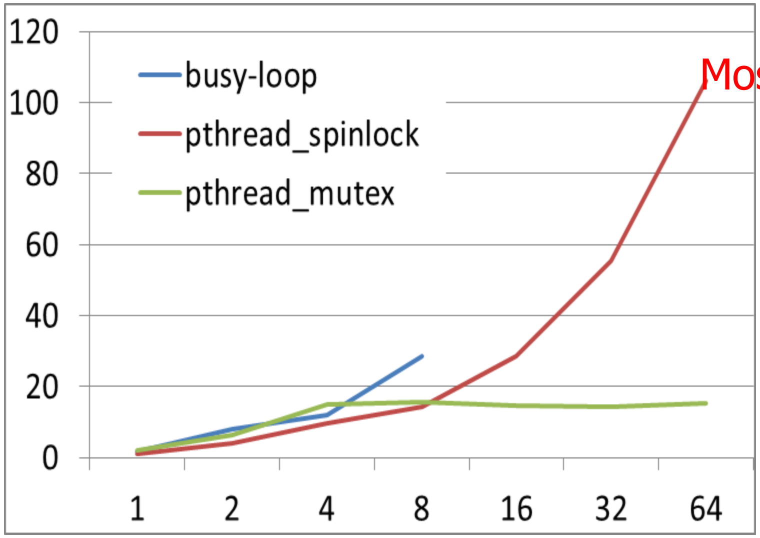
Serialization

- **Critical sections serialize the code execution**
  - Too many or large critical sections can slow down the performance – sequential code may run faster

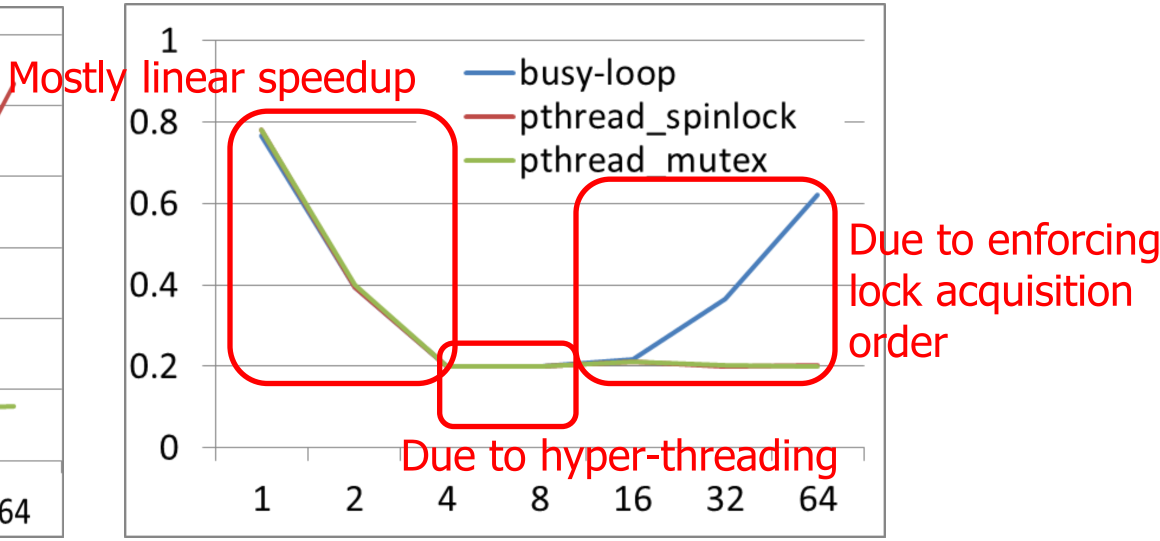


Performance evaluation of critical section after loop

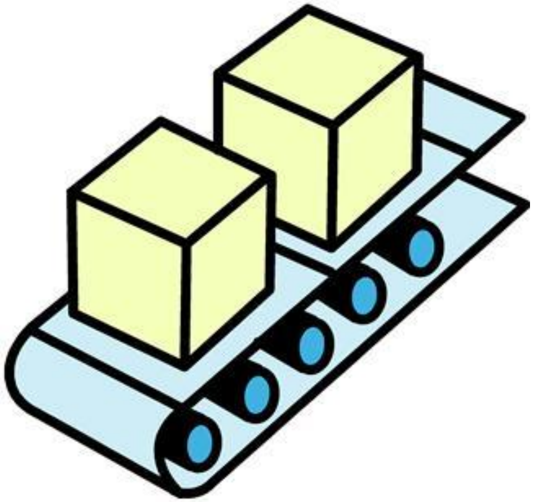
- Busy-loop vs. pthread_spinlock vs. pthread_mutex
- Environment
  - Intel i7 4 cores + 2 SMT per core (8 logicla cores)



Lock/unlock per global sum update



Single critical section after loop



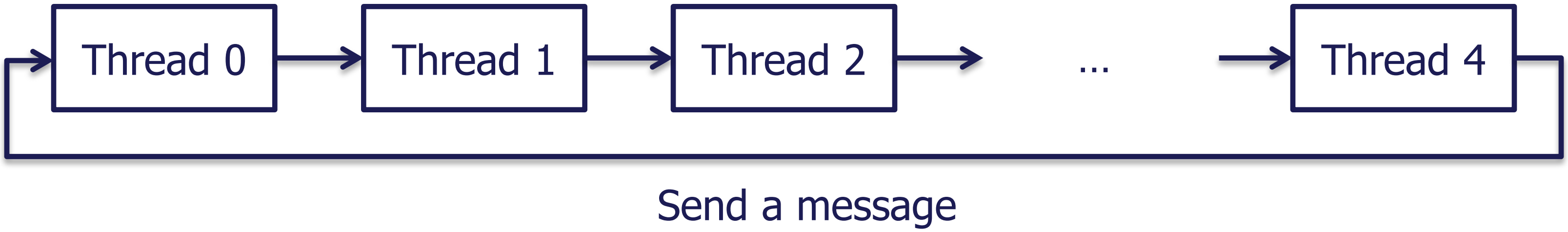
PRODUCER-CONSUMER SYN CHRONIZATION AND SEMAP HORES

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 problem

Producer Consumer Problem

- **Example**
  - Sending a message to its successor thread



A first attempt at sending messages using pthreads

```
/* messages has type char**. It's 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_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 */
```

Some threads can print this message because it's not given a message from its predecessor

Mutex-based Solution?

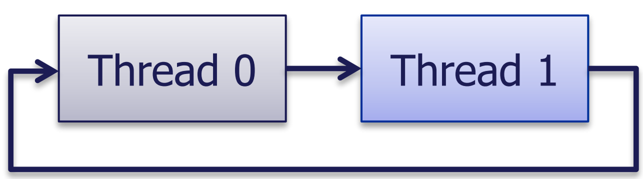
- **Using a mutex array**
  - mutex[t] where t is the number of threads
  - Each mutex protects each thread's message buffer

Thread 1

```
. . .
pthread_mutex_lock(mutex[dest]);
. . .
messages[dest] = my_msg;
pthread_mutex_unlock(mutex[dest]);
. . .
pthread_mutex_lock(mutex[my_rank]);
```

Thread 0

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



→ Thread 0 could print NULL

Semaphore

- **A control knob**
  - Whether one or multiple threads (processes) can proceed or not
  - Using semaphore in C

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   */);
```

Shared between processes or threads

Initial value

```
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 */);
```

Increase an int value so that a thread can enter a critical section

If the int value is 0, waits until other threads increases the int value.
If the int value is positive, decrease it and enter a critical section

Semaphore-based Solution

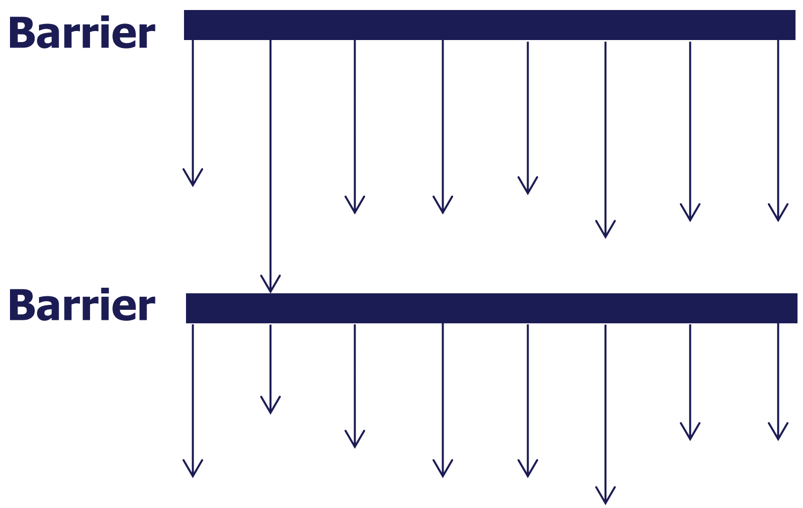
```
/* messages is allocated and initialized to NULL in main    */
/* semaphores is allocated and initialized to 0 (locked) in
      main */
void* Send_msg(void* rank) {
   long my_rank = (long) rank;
   long dest = (my_rank + 1) % thread_count;
   char* my_msg = malloc(MSG_MAX*sizeof(char));

   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 AND CONDITION VARIABLES

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;
```

Barrier

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);
}
```

Barrier

Implementing a Barrier

- **Using 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+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.

Protects the counter variable

Busy loop until all threads reach here

Problem of Busy-waiting+Mutex barrier

- **If we want to use multiple barrier**
  - Reuse one barrier
    - We need to reset the counter variable
    - If some threads did not exit the while loop, because the counter variable becomes zero, the threads cannot proceed
  - Build multiple barrier
    - Waste of memory
    - # of counter + mutex is linear to the number of barrier we want to use

Implementing a Barrier

- **Using 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.

- **Using semaphore**

Semaphore-based Barrier

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

Protects the counter variable

The last thread opens the gate to make other threads to be able to proceed

Block threads until the gate is opened

Semaphore-based Barrier – con't

- **Advantage**
  - Do not waste CPU cycles compared to the busy-wait+mutex barrier

- **Reusable?**
  - No
  - For some reason if a thread is de-scheduled for a long time so that it does not pass the sem_wait(&barrier_sem) in the first barrier, other thread can pass through the second barrier
    - The gate of the first barrier is opened until every thread passes it
    - Some threads reached the second barrier can think that the second barrier is opened

Condition Variables

- **Another way for thread synchronization**
  - While 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, the programmer would need to have threads continually polling to check if the condition is met.
    - This can be very resource consuming since the thread would be continuously busy in this activity.
  - A condition variable is **always** associated with a mutex lock.

Condition Variables – con't

- **How condition variables work**
  - A thread locks a mutex associated with a condition variable.
  - The thread tests the condition to see if it can proceed.
  - If it can
    - Your thread does its work.
    - Your thread unlocks the mutex.
  - If it cannot
    - The thread sleeps. The mutex is automatically released.
    - Some other threads signals the condition variable.
    - Your thread wakes up from waiting with the mutex automatically locked, and it does its work.
    - Your thread releases the mutex when it's done.

Condition Variables – con't

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

Pthread APIs for Condition Variable

- **Static initialization**
  - pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
- **Dynamic initialization**
  - pthread_cond_t cond;
    pthread_cond_init (&cond, (pthread_condattr_t*)NULL);
- **Destroying a condition variable**
  - pthread_cond_destroy (&cond);
  - Destroys a condition variable, freeing the resources it might hold.

Pthread APIs for Condition Variable -con't

- **int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex)**
  - Blocks the calling thread until the specified condition is signalled.
  - This should be called while mutex is locked, and it will automatically release the mutex while it waits.
- **int pthread_cond_signal (pthread_cond_t *cond)**
  - Signals another thread which is waiting on the condition variable.
  - Calling thread should have a lock.
- **int pthread_cond_broadcast(pthread_cond_t *cond)**
  - Used if more than one thread is in a blocking wait state.

Barrier using Condition Variable

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

Lock a mutex which is associated with the condition variable.
This lock also protects the counter variable.

Wakeup other threads when every thread arrives at this barrier

If every thread doesn't reach this barrier, a thread sleeps here

Spin-then-Block Barrier

- **Busy-loop-based barrier**

  \- Waste of CPU cycles

  \+ Good when multiple threads will reach a barrier in a short time

- **Mutex-based barrier**

  \+ No waste of CPU cycles

  \- Blocking and waking up threads wastes scheduling and wakeup costs in OS

- **Spin-then block barrier**
  - Takes the advantages of both approaches
  - Spins for a while to wait for other threads
  - If spinning gets long, a thread sleeps

Spin-then-Block Barrier –con't

```
int counter = 0
pthread_mutex_t mutex;
pthread_cond_t cond;

Thread_work() {
    ...
    /* barrier */
    pthread_mutex_lock(&mutex);
    counter++;
    if (counter == thread_count ) {
        counter = 0;
        pthread_cond_broadcast(&cond);
    } else {
        int spin_count = 0;
        pthread_mutex_unlock(&mutex);
        while ( counter != 0 && ++spin_count < spin_threshold );
        pthread_mutex_lock(&mutex);
        if ( counter != 0 )
            while ( pthread_cond_wait(&cond, &mutex) != 0 );
    }
    pthread_mutex_unlock(&mutex);
    ...
}
```

Spin for a while

Then block

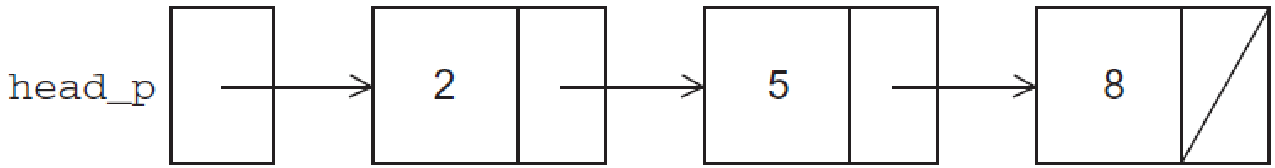
Barrier APIs in Pthread

- **Not all systems implement barrier API**
- **But, some systems provide barrier API in their Pthread libraries**
  - E.g., Linux
- **APIs**
  - pthread_barrier_init(pthread_barrier_t* barrier, pthread_barrierattr_t* attr, int value)
    - Initialize a barrier
    - The integer value specifies the number of threads to synchronize
    - Attr is usually NULL
  - pthread_barrier_wait(pthread_barrier_t* barrier)
    - Waits until the specified number of threads arrives at the barrier

READ-WRITE LOCKS

Controlling a Large, Shared data Structure

- **A linked list**
  - Each node stores an int value
  - All nodes are linked in sorted order
  - Methods
    - Member() tests whether a value is in the list
    - Insert() inserts a new value
    - Delete() removes a specified value




```c
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 */
```

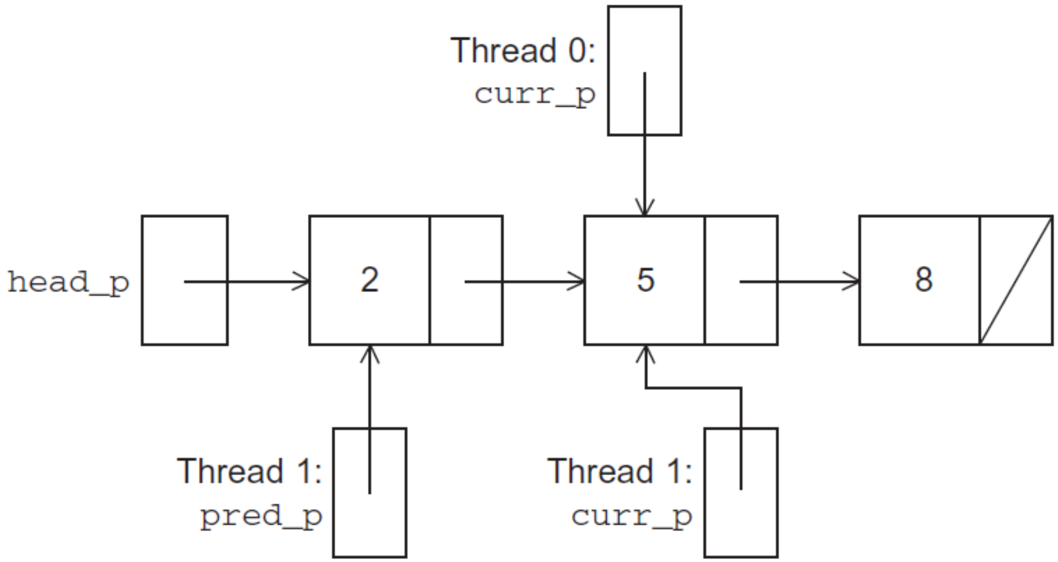
Deleting a node from a linked list

```
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

- **Multiple threads concurrently access a shared linked list**
  - head_p is a global variable for the entry of the linked list
  - Multiple threads invoke Member(), Insert() and Delete() methods

→ Race condition → Non-determinism



Solution #1

- **Use a mutex to protect entire linked list (Coarse-grined locking)**

```
Pthread_mutex_lock(&list_mutex);
Member(value), Insert(value) or Delete(value)
Pthread_mutex_unlock(&list_mutex);
```

- **Issues**
  - Serialization of threads
  - Member() actually does not modify the linked list

  → Serialization looses the opportunity for parallelism
  - Insert() and Delete() are majority of uses

  → Serialization can be a good solution

  → But, multiple threads can update different locations in the linked list

Solution #2

- **A fine-grained locking**
  - Use multiple mutex to protect each node

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

- **Issues**
  - More complex implementation of Member(), Insert() and Delete() functions.
  - Slower than using one mutex for whole linked list.
    - Accessing every node invokes mutex lock/unlock functions
  - Storage overhead

Member() using Fine-grained Locking

Coarse-grained locking

```
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 */
```

Fine-grained locking

```
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 */
```

Read-Write Lock

- **Neither of the solutions exploits the potential for simultaneous access to any node by threads that are executing Member().**
  - The coarse-grained locking only allows one thread to access the entire list at any instant.
  - The fine-grained locking only allows one thread to access any given node at any instant.

- **How about to use read-write lock?**
  - Multiple Member() functions can run in parallel
  - Read-write lock still provides mutual exclusion to modifications (Insert() and Delete())

Read-Write Lock in Pthread

- **Static initialization**
  - Pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER
- **Dynamic initialization**
  - Pthread_rwlock_init(pthread_rwlock_t *rwlock,
    pthread_rwlockattr_t *attr)
- **Destroying a rwlock**
  - Pthread_rwlock_destroy(pthread_rwlock_t* rwlock)
- **Read locking**
  - Pthread_rwlock_rdlock(pthread_rwlock_t* rwlock)
- **Write locking**
  - Pthread_rwlock_wrlock(pthread_rwlock_t* rwlock)
- **Unlocking**
  - Pthread_rwlock_unlock(pthread_rwlock_t* rwlock)

Solution #3

- **Read-Write lock-based**

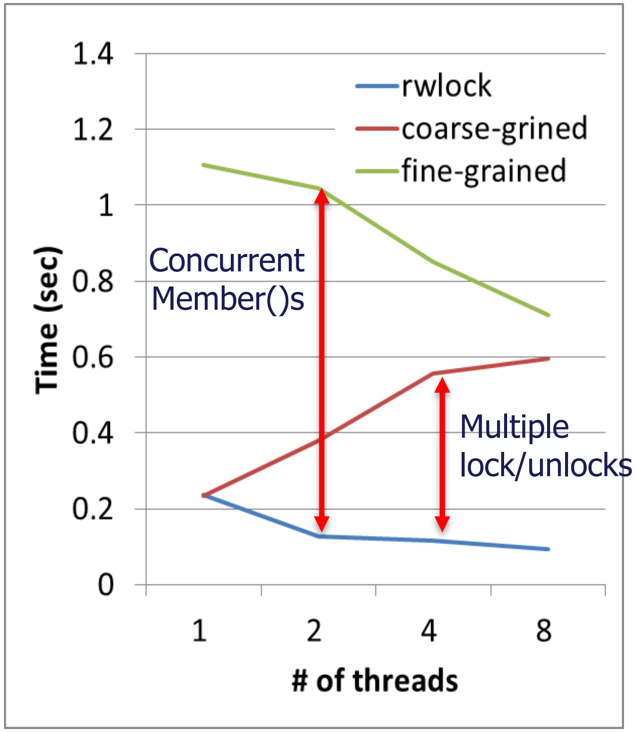
```
pthread_rwlock_rdlock(&rwlock);
Member(value);
pthread_rwlock_unlock(&rwlock);
. . .
pthread_rwlock_wrlock(&rwlock);
Insert(value);
pthread_rwlock_unlock(&rwlock);
. . .
pthread_rwlock_wrlock(&rwlock);
Delete(value);
pthread_rwlock_unlock(&rwlock);
```

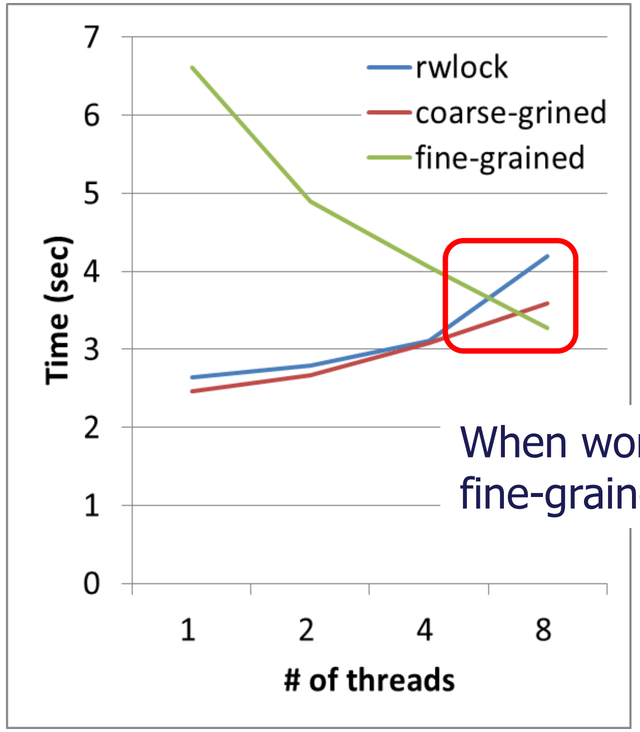
Linked List Performance

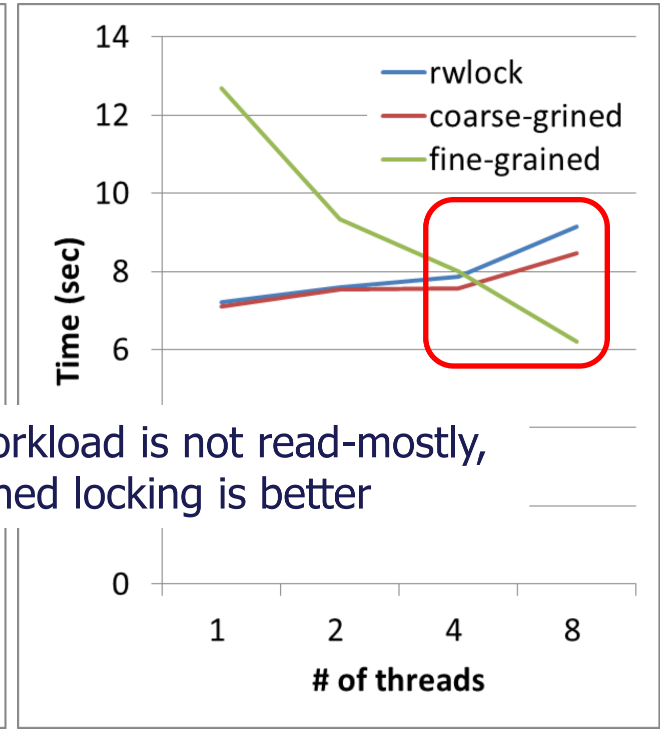
- **Environment**
  - Intel i7 920 4 cores + 2 SMT per core (8 logical cores)

- **1000 initial nodes + 100,000 operations**
- **Three cases**
  - 90% of Member(), 5% of Insert(), 5% of Delete()
  - 80% of Member(), 10% of Insert(), 10% of Delete()
  - 60% of Member(), 20% of Insert(), 20% of Delete()

Linked List Performance –con't







When workload is not read-mostly, fine-grained locking is better

Member 99%
Insert 0.5%
Delete 0.5%

Member 80%
Insert 10%
Delete 10%

Member 60%
Insert 20%
Delete 20%

- **Best use of lock depends on the access patterns to a shared data**

THREAD-SAFETY

Thread-Safety

- **A block of code is thread-safe if it can be simultaneously executed by multiple threads without causing problems.**
  - Functions called from a thread must be thread-safe.
  - We identify four (non-disjoint) classes of thread-unsafe functions:
    - Class 1: Failing to protect shared variables
    - Class 2: Relying on persistent state across invocations
    - Class 3: Returning a pointer to a static variable
    - Class 4: Calling thread-unsafe functions

Thread Safety – con't

- **Class 1: Failing to protect shared variables.**
  - Fix: Use mutex operations.
  - Issue: Synchronization operations will slow down code.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
int cnt = 0;
/* Thread routine */
void *count(void *arg) {
    int i;

    for (i=0; i<NITERS; i++) {
        pthread_mutex_lock (&lock);
        cnt++;
        pthread_mutex_unlock (&lock);
    }
    return NULL;
}
```

Thread Safety – con't

- **Class 2: Relying on persistent state across multiple function invocations.**
  - Random number generator relies on static state
  - Fix: Rewrite function so that caller passes in all necessary state.

```
/* rand - return pseudo-random integer on 0..32767 */
int rand(void) {
    static unsigned int next = 1;
    next = next*1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}
/* srand - set seed for rand() */
void srand(unsigned int seed) {
    next = seed;
}
```

Thread Safety – con't

- **Class 3: Returning a ptr to a static variable.**
- **Fixes:**
  1. Rewrite code so caller passes pointer to struct.
     - Issue: Requires changes in caller and callee.
  2. Lock-and-copy
     - Issue: Requires only simple changes in caller (and none in callee)
       - However, caller must free memory.

```
struct hostent *gethostbyname(char
*name){
    static struct hostent h;
    <contact DNS and fill in h>
    return &h;
}
```

```
hostp = malloc(...));
gethostbyname_r(name, hostp);
```

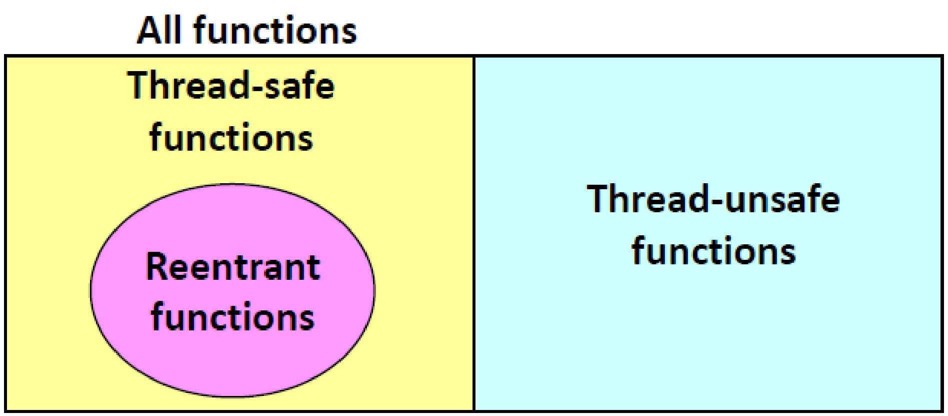
```
struct hostent
*gethostbyname_ts(char *name)
{
    struct hostent *unshared
                        = malloc(...);
    pthread_mutex_lock(&lock); /* lock */
    shared = gethostbyname(name);
    *unshared = *shared; /* copy */
    pthread_mutex_unlock(&lock);
    return q;
}
```

Thread Safety – con't

- **Class 4: Calling thread-unsafe functions.**
  - Calling one thread-unsafe function makes an entire function thread-unsafe.
  - Fix: Modify the function so it calls only thread-safe functions

Reentrant Functions

- **A function is reentrant iff it accesses NO shared variables when called from multiple threads.**
  - Reentrant functions are a proper subset of the set of thread-safe functions.



–NOTE: The fixes to Class 2 and 3 thread-unsafe functions require modifying the function to make it reentrant.

Thread-Safe Library

- **Many standard C library functions are thread safe**
  - See “man 7 pthreads”

- Some functions are not thread-safe
  - These usually have reentrant version as well

Thread-unsafe function	Class	Reentrant version
asctime	3	asctime_r
ctime	3	ctime_r
gethostbyaddr	3	gethostbyaddr_r
gethostbyname	3	gethostbyname_r
inet_ntoa	3	(none)
localtime	3	localtime_r
rand	2	rand_r

THREAD SUPPORT IN C++11

Thread class

- **Header file**
  - `#include <thread>`
- **Creation**
  - `std::Thread t1(thread_func, id)`
- **Destroy**
  - `std::terminate()` inside a thread ≈ `pthread_exit()`
  - `~t1()` ≈ `pthread_cancel(pthread_t)`
- **Methods**
  - `get_id()` ≈ `pthread_self()`
  - `detach()` ≈ `pthread_detach(pthread_t)`
  - `join()` ≈ `pthread_join(pthread_t)`
  - `native_handle()`
    - Returns `pthread_t` on a POSIX system

Mutex Class

- **Header file**
  - `#include <mutex>`
- **Construction**
  - `std::mutex mutex;`
- **Methods**
  - `lock() ≈ pthread_mutext_lock(&mutex)`
  - `try_lock() ≈ pthread_mutex_trylock(&mutex)`
  - `unlock() ≈ pthread_mutex_unlock(&mutex)`

- **Variant of mutex**
  - `recursive_mutex` class
  - `timed_mutex` class

Condition Variable Class

- **Headerfile**
  - `#include <condition_variable>`
- **Methods**
  - `notify_one()` ≈ `pthread_cond_signal(&cond)`
  - `notify_all()` ≈ `pthread_cond_broadcast(&cond)`
  - `wait(std:unique_lock<std::mutex>& lock, Predicate pred)` ≈ `pthread_mutex_wait(&cond, &mutex)`
- **Other classes and APIs**
  - Refer to http://en.cppreference.com/w/cpp/thread

Conclusion

- **Programming in a shared memory system**
  - Pthread is a standard thread library on POSIX systems
- **Synchronization**
  - Busy-waiting
  - Semaphore
  - Mutex, spinlock, and read/write locks
  - Barrier
  - Condition variable
- **Thread safety**
- **C++11 supports thread**