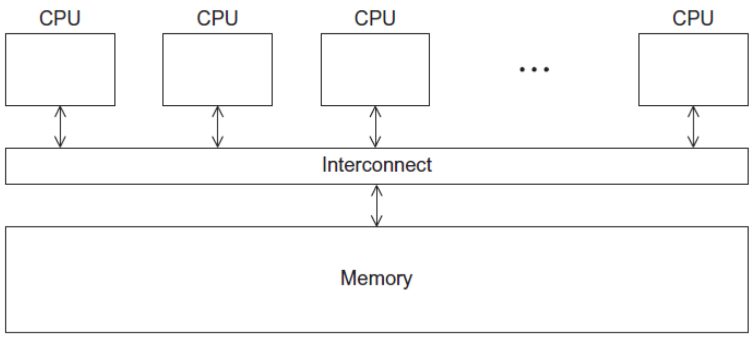
INTRODUCTION TO PTHREADS

From Xiaoyao Liang

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.

A SHARED MEMORY SYSTEM



POSIX THREADS

- Also known as Pthreads.
- A standard for Unix-like operating systems.
- A library that can be linked with C programs.
- Specifies an application programming interface (API) for multi-threaded programming.
- The Pthreads API is only available on POSIX systems — Linux, MacOS X, Solaris, HPUX, …

"HELLO WORLD !"

```
#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.

allocate memory space for thread handlers

5

"HELLO WORLD !"

```
   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 */

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 */
```

6

RUNNING THE PROGRAM

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

link in the Pthreads library

```
. / pth_hello 4
```

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

CREATING THE THREADS

pthread.h

pthread_t

One object for each thread.

```
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" OBJECT

- 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.
- Allocate object space before using.

PTHREADS FUNCTION

- 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.

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.

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;
```

ESTIMATING 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 */
```

13

BUSYING-WAITING

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

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

flag initialized to 0 by main thread

14

BUSYING-WAITING

- Beware of optimizing compilers, though!

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

Compiler can change the program order

- Disable compiler optimization
- Protect variable from optimization

int volatile flag

int volatile x

15

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 */
```

Using % so that last thread reset flag back to 0

16

REMOVE CRITICAL SECTION IN A LOOP

19.8 Seconds two threads Vs. 2.5 Seconds one threads
What is the problem??

```
   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 */
```

1.5 after moving the critical section out of the loop

17

PROBLEMS IN BUSY-WAITING

- A thread that is busy-waiting may continually use the CPU accomplishing nothing.
- Critical section is executed in thread order, large wait time if thread number exceed core number.

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.

18

MUTEXES

- 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.

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

MUTEXES

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

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

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

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

ESTIMATING PI MUTEXES

```
   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 */
```

21

PERFORMANCE COMPARISON

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.

22

SOME 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.

23

MESSAGE PASSING EXAMPLE

```
/* 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 */
```

24

MESSAGE PASSING EXAMPLE

```
…
pthread_mutex_lock(mutex[dest]);
…
messages[dest]=my_msg;
pthread_mutex_unlock(mutex[dest]);
…
pthread_mutex_lock(mutex[my_rank]);
printf("Thread  %ld>%s\n", my_rank, messages[my_rank]);
pthread_mutex_unlock(mutex[my_rank]);
```

Problem: If one thread goes too far ahead, it might access to the uninitialized location and crash the program.

Reason: mutex is always initialized as "unlock"

25

SEMAPHORE

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 */);
```

26

USING SEMAPHORE

all semaphores initialized to 0 **(locked)**
…
messages[dest]=my_msg;
sem_post(&semaphores[dest]); **/*unlock the destination semaphore*/**
…
sem_wait(&semaphores[my_rank]); **/*wait for its own semaphore to be unlocked*/**
printf("Thread %ld>%s\n", my_rank, messages[my_rank]);

Semaphore is more powerful than mutex because you can initialize semaphore to any value

How to use mutex for the message passing?

27

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.

28

BARRIERS FOR EXECUTION TIME

```
/* 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;
```

29

BARRIERS FOR DEBUG

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

30

IMPLEMENTING BARRIER

```
/* 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.

31

CAVEAT

- Busy-waiting wastes CPU cycles.
- What about we want to implement a second barrier and reuse counter?
  - ➢ *If counter is not reset, thread won't block at the second barrier*
  - ➢ *If counter is reset by the last thread in the barrier, other threads cannot see it.*
  - ➢ *If counter is reset by the last thread after the barrier, some thread might have already entered the second barrier, and the incremented counter might get lost.*
- Need to use different counters for different barriers.

32

IMPLEMENTING 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);
   }
   . . .
```

33

CAVEAT

- No wasting CPU cycles since no busy-waiting
- What happens if we need a second barrier?
  - *"counter" can be reused.*
  - *"counter_sem" can also be reused.*
  - *"barrier_sem" need to be unique, there is potential that a thread proceeds through two barriers but another thread traps at the first barriers if the OS put the thread at idle for a long time.*

34

PTHREADS BARRIER

- Open Group provides Pthreads barrier

  pthread_barrier_init();
  pthread_barrier_wait();
  pthread_barrier_destroy();

- Not universally available

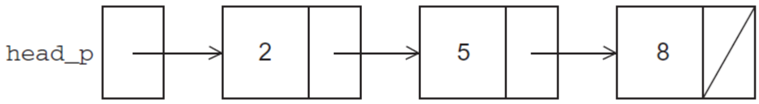
35

LINKED LIST

- 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.

36

LINKED LIST




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

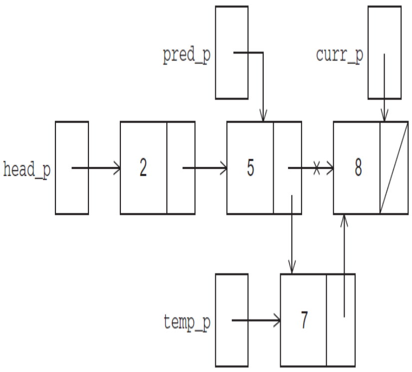
MEMBERSHIP

```
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 */
```

INSERT

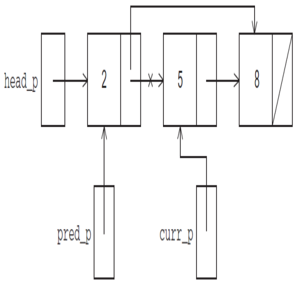



```
int Insert(int value, struct list_node_s** head_pp) {
   struct list_node_s* curr_p = *head_pp;
   struct list_node_s* pred_p = NULL;
   struct list_node_s* temp_p;

   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 */
```

DELETE

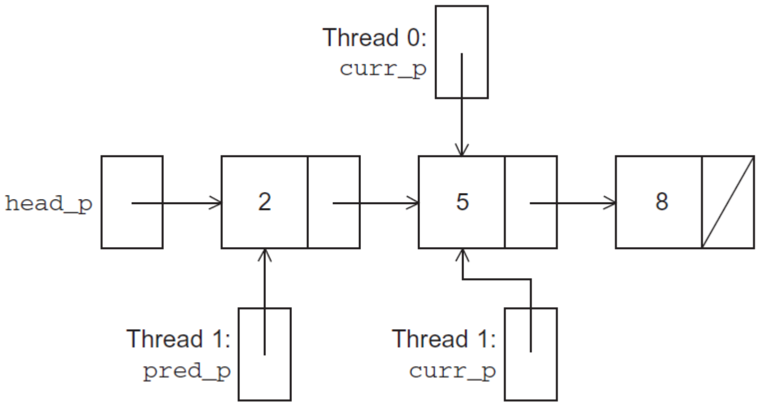



```
int Delete(int value, struct list_node_s** head_pp) {
   struct list_node_s* curr_p = *head_pp;
   struct list_node_s* pred_p = NULL;

   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 */
```

LINKED LIST WITH MULTI-THREAD



41

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).

42

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.

43

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;
}
```

44

SOLUTION #2

```
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 */
```

45

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.

46

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.

47

READ-WRITE LOCKS

- A read-write lock is somewhat like a mutex except that it provides two lock functions.
- The first lock function is a read lock for reading, while the second locks it for writing.
- If any threads own the lock for reading, any threads that want to obtain the lock for **writing will block**. **But reading will not be blocked.**
- 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

48

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

49

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

50