HICREST

UNITRENTO

Introduction to Parallel Computing

a.y. 2025 – 2026

Dr. Brahmaiah Gandham

Lecture 7

Shared Memory Programming with Pthreads

Outline

- Producer-consumer synchronization
- semaphores.
- Barriers and condition variables.
- Read-write locks.

Deadlocks

UNITRENTO

- Deadlock: is any situation in which no member of some group of entities can proceed because each waits for another member, including itself, to take action, such as sending a message or, more commonly, releasing a lock.
- E.g., locking mutexes in reverse order

```
/* Thread A */
pthread_mutex_lock(&mutex1);
pthread_mutex_lock(&mutex2);

/* Thread B */
pthread_mutex_lock(&mutex2);
pthread_mutex_lock(&mutex1);
```

Global sum function using 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 */
```

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

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

In both cases, the critical section is outside the loop

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.

Example: message exchange in a ring (receive from left, send to right)

UNITRENTO

```
/* 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 */
```

Issue: Some threads might read messages[dest] before the other thread put something in there

How to fix it?

- We could fix it with busy waiting, but would have the same problems discussed before

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

- In principle it is possible to fix it with mutex (but in a complex way)
- POSIX provides a better way: **semaphores** (it is not part of pthread, could be not available on macOS)

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  */);
```

Semaphores can be shared also among processes (shared !=0)

```
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 */);
```

Semaphores

UNITRENTO

- sem_wait(sem_t *sem) blocks if the semaphore is 0. If the semaphore is > 0, it will decrement the semaphore and proceed.
- sem_post(sem_t *sem) if there is a a thread waiting in sem_wait(), that thread can proceed with the execution. Otherwise, the semaphore is incremented.
- sem_getvalue(sem_t *sem, int *sval) places the current value of the semaphore pointed to sem into the integer pointed to by sval.
- Notes:
- Mutexes are binary. Semaphores are unsigned int.
- Mutexes start unlocked. Semaphores start with initial value.
- Mutexes are usually locked/unlocked by the same thread. Semaphores are usually increased/decreased by different threads.

Using semaphores so that threads can send messages

UNITRENTO

```
1  /* messages is allocated and initialized to NULL in main   */
2  /* semaphores is allocated and initialized to 0 (locked) in
         main */
3  void* Send_msg(void* rank) {
4     long my_rank = (long) rank;
5     long dest = (my_rank + 1) % thread_count;
6     char* my_msg = malloc(MSG_MAX*sizeof(char));
7
8     sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
9     messages[dest] = my_msg;
10    sem_post(&semaphores[dest])
            /* ''Unlock'' the semaphore of dest */
11
12    /* Wait for our semaphore to be unlocked */
13    sem_wait(&semaphores[my_rank]);
14    printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
15
16    return NULL;
17 }  /* Send_msg */
```

Note: This problem does not have a critical section. It has a type of synchronization known as **producer-consumer**

Barriers

UNITRENTO

- 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

UNITRENTO

```
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

UNITRENTO

- 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);
    . . .
}
```

What if we want to use the barrier again?

Someone must reset counter to 0. Who and when?

If a thread reset it to 0, this might be reset before all the threads see it was equal to thread_count

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);
   }
   . . .
```

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);
   }
   . . .
```

Do we have the same problem here? Yes, there could be a race condition

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.
- A condition variable indicates an event; cannot store or retrieve a value from a condition variable

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;
```

Implementing a barrier with condition variables

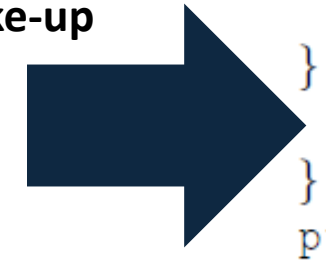
UNITRENTO

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

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);
    . . .
 }
```

Spurious wake-up



Condition variables

Condition variables in Pthreads have type *pthread_cond_t*. The function will unblock one of the blocked threads, and

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

will unblock all of the blocked threads. The functions

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

Create and destroy the condition variable.

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

int pthread_cond_destroy(pthread_cond_t*  cond_p  /* in/out */);
```

Condition variables

The function

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

1) unlock the mutex;

2) block the thread until it is unblocked by another thread's call to *pthread_cond_signal* or *pthread_cond_broadcast*

3) when the thread is unblocked, lock the mutex

It is like:

```
pthread_mutex_unlock(&mutex_p);
wait_on_signal(&cond_var_p);
pthread_mutex_lock(&mutex_p);
```

Condition variables

UNITRENTO

- pthread_cond_wait(), pthread_cond_signal() differ from sem_wait() e sem_post() because:

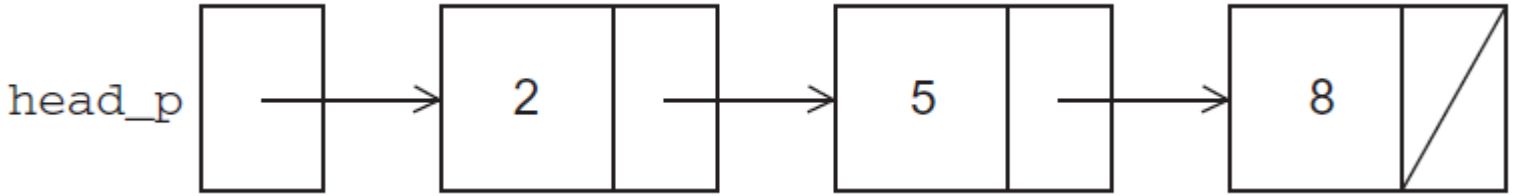
- pthread_cond_wait() ALWAYS blocks process execution
- pthread_cond_signal() can be ignored. If there are no threads in the cond_wait, the signal is lost
- sem_t can get values >=0.

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;
}
```

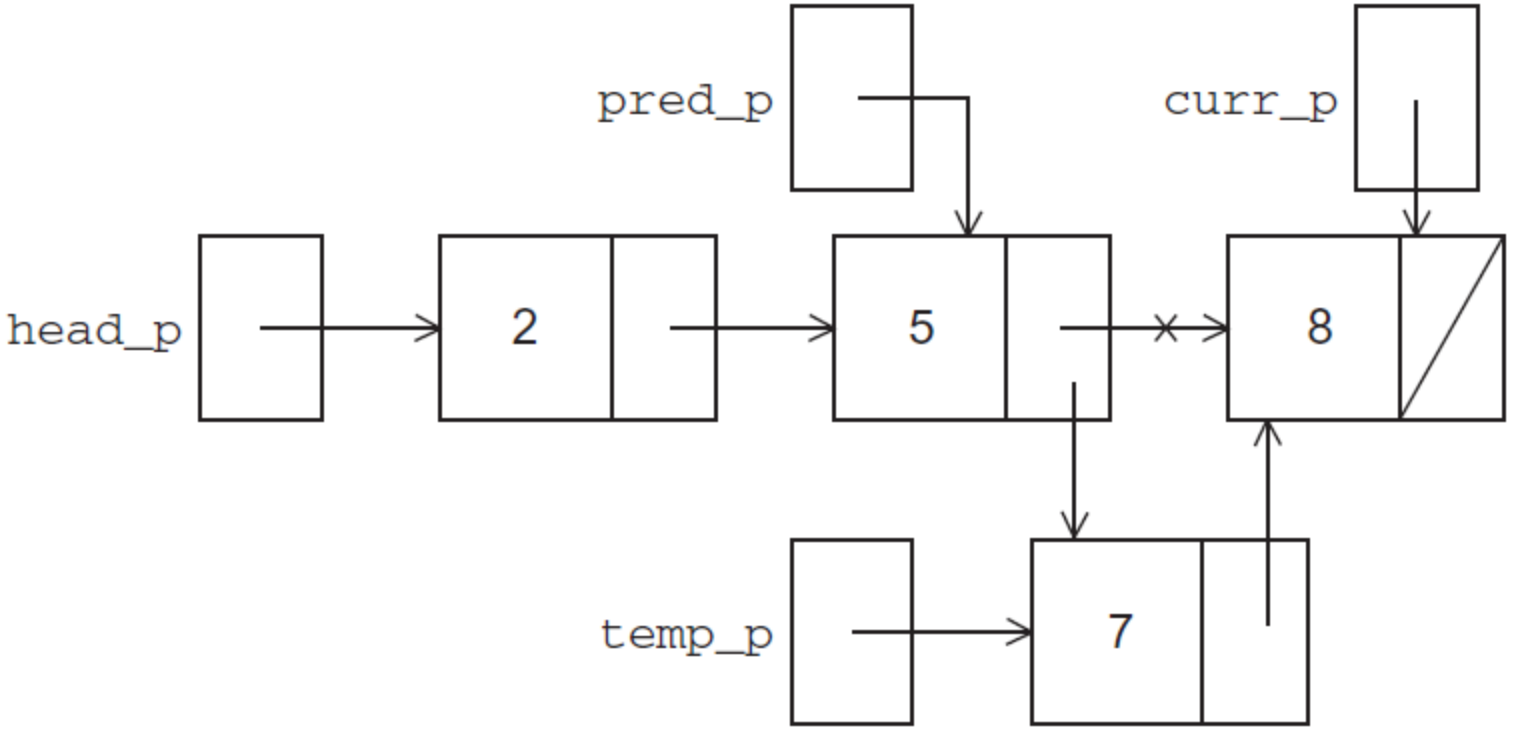
Linked List Membership

```
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



Inserting a new node into a list

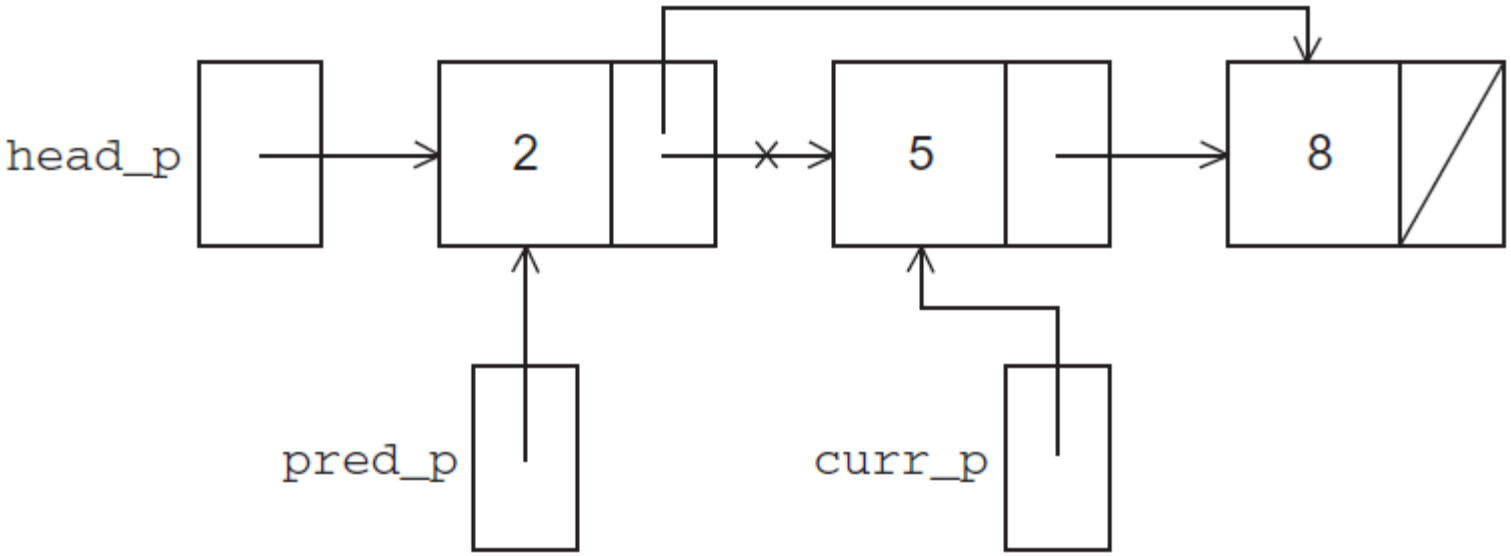
UNITRENTO

```
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



Deleting a node from a linked list

UNITRENTO

```
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 */
```

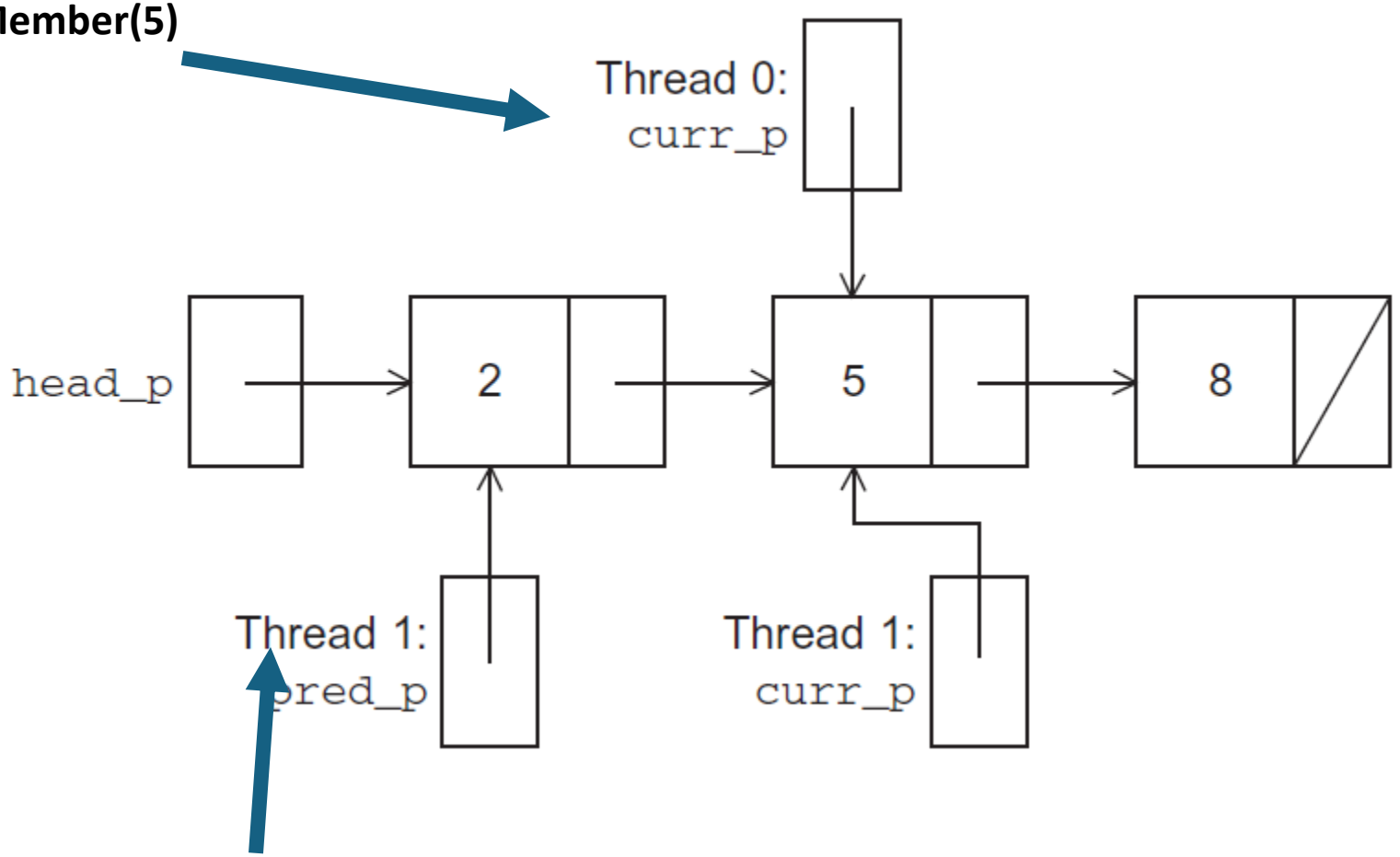
A Multi-Threaded Linked List

UNITRENTO

- 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.
- If multiple threads call Member at the same time, we are fine
- What if one thread calls Member while another thread is deleting an element?

Simultaneous access by two threads

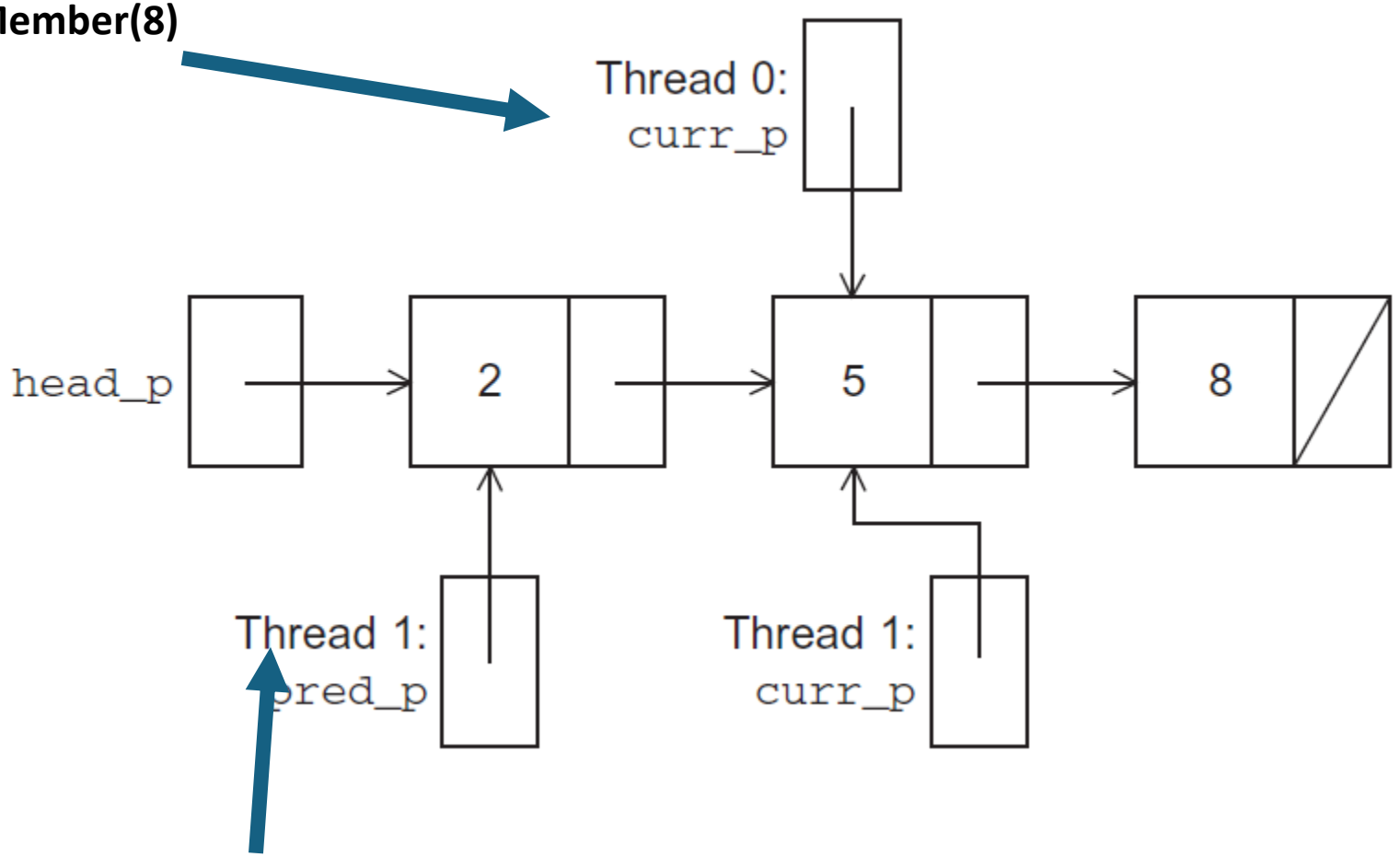
Executing **Member(5)**



Executing **Delete(5)**

Simultaneous access by two threads

Executing **Member(8)**



Executing **Delete(5)**

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 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.

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

UNITRENTO

- 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.

pthread_rwlock functions

UNITRENTO

- pthread_rwlock_init initializes the rwlock

```
int pthread_rwlock_init(pthread_rwlock_t* rwlock, pthread_rwlockattr_t* attr);
```

- pthread_rwlock_destroy frees the rwlock

```
int pthread_rwlock_destroy(pthread_rwlock_t* rwlock);
```

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

8 (physical) cores

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

8 (physical) cores

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

8 (physical) cores

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

8 (physical) cores

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

8 (physical) cores

Concluding Remarks

- 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.
- A condition variable is a data object that allows a thread to suspend execution until a certain event or condition occurs.
- 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.

References

- An Introduction to Parallel Programming, Peter Pacheco