Advanced Topics on Shared Memory Programming with Pthreads

Pacheco. Chapter 4

T. Yang. UCSB CS140. Spring 2014

Outline

- **More on thread synchronization.**
  - Read-write locks.
  - Applications in a shared link list
- **False sharing**
- **Deadlocks and thread safety.**

READ-WRITE LOCKS

Synchronization Example for Readers-Writers Problem

- **A data set is shared among a number of concurrent threads.**
  - Readers – only read the data set; they do **not** perform any updates
  - Writers – can both read and write
- **Requirement:**
  - allow multiple readers to read at the same time.
  - Only one writer can access the shared data at the same time.
- **Reader/writer access permission table:**

	Reader	Writer
Reader	OK	No
Writer	NO	No

Readers-Writers (First try with 1 mutex lock)

- **writer**

```
  do {
          mutex_lock(w);
          //    writing is performed
          mutex_unlock(w);
 } while (TRUE);
```

- **Reader**

```
          do {
          mutex_lock(w);
          //    reading is performed
          mutex_unlock(w);
 } while (TRUE);
```

	Reader	Writer
Reader	?	?
Writer	?	?

Readers-Writers (First try with 1 mutex lock)

- **writer**

```
        do {
                mutex_lock(w);
                //    writing is performed
                mutex_unlock(w);
        } while (TRUE);
```

- **Reader**

```
                do {
                mutex_lock(w);
                //    reading is performed
                mutex_unlock(w);
        } while (TRUE);
```

	Reader	Writer
Reader	no	no
Writer	no	no

2nd try using a lock + readcount

- **writer**

```
 do {
            mutex_lock(w);// Use writer mutex lock
            //    writing is performed
            mutex_unlock(w);
} while (TRUE);
```

- **Reader**

```
do {
            readcount++; // add a reader counter.
            if(readcount==1) mutex_lock(w);
            //    reading is performed
           readcount--;
              if(readcount==0)  mutex_unlock(w);
} while (TRUE);
```

Readers-Writers Problem with semaphone

- **Shared Data**
  - Data set
  - Lock mutex (to protect readcount)
  - Semaphore wrt initialized to 1 (to synchronize between readers/writers)
  - Integer readcount initialized to 0

Readers-Writers Problem

- **A writer**

```
do {
        sem_wait(wrt) ;  //semaphore wrt

       // writing is performed

        sem_post(wrt) ;  //
} while (TRUE);
```

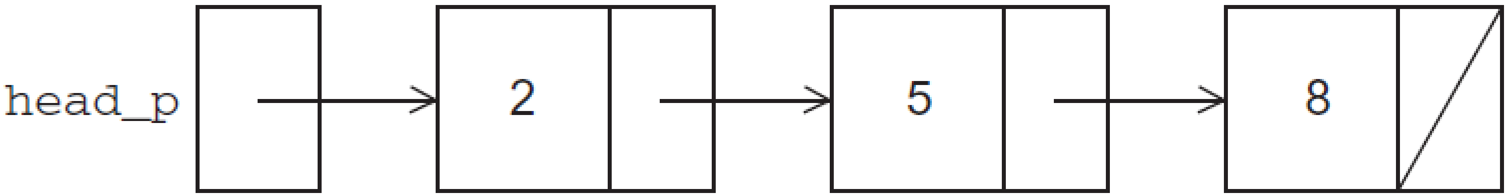
Readers-Writers Problem (Cont.)

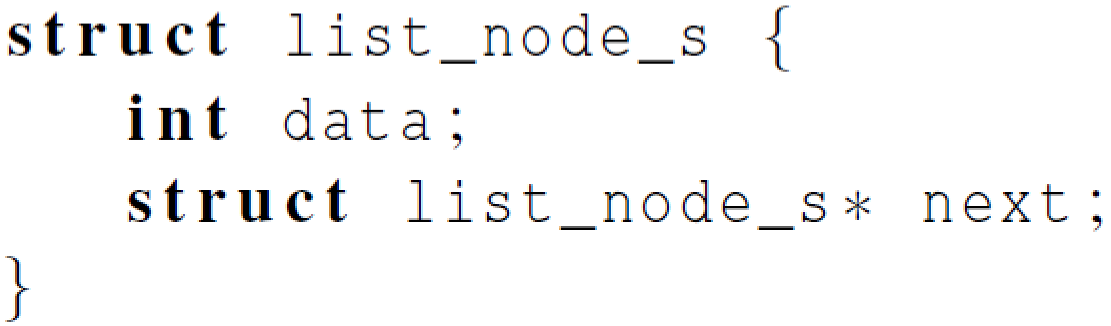
- **Reader**

```
do {
                mutex_lock(mutex);
                readcount ++ ;
                if (readcount == 1)
                            sem_wait(wrt);  //check if anybody is writing
                mutex_unlock(mutex)

                 // reading is performed

                 mutex_lock(mutex);
                 readcount  - - ;
                 if (readcount  == 0)
                           sem_post(wrt) ;  //writing is allowed now
                 nlock(mutex) ;
        } while (TRUE);
```

Application case: Sharing a sorted linked list of integers

- **Demonstrate controlling of access to a large, shared data structure**
- **Operations supported**
  - Member, Insert, and Delete.

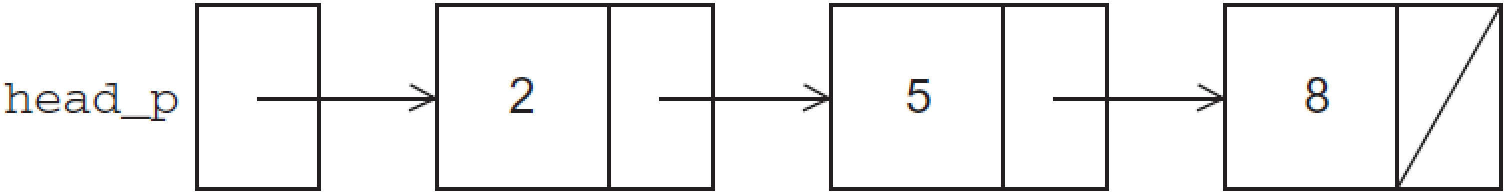




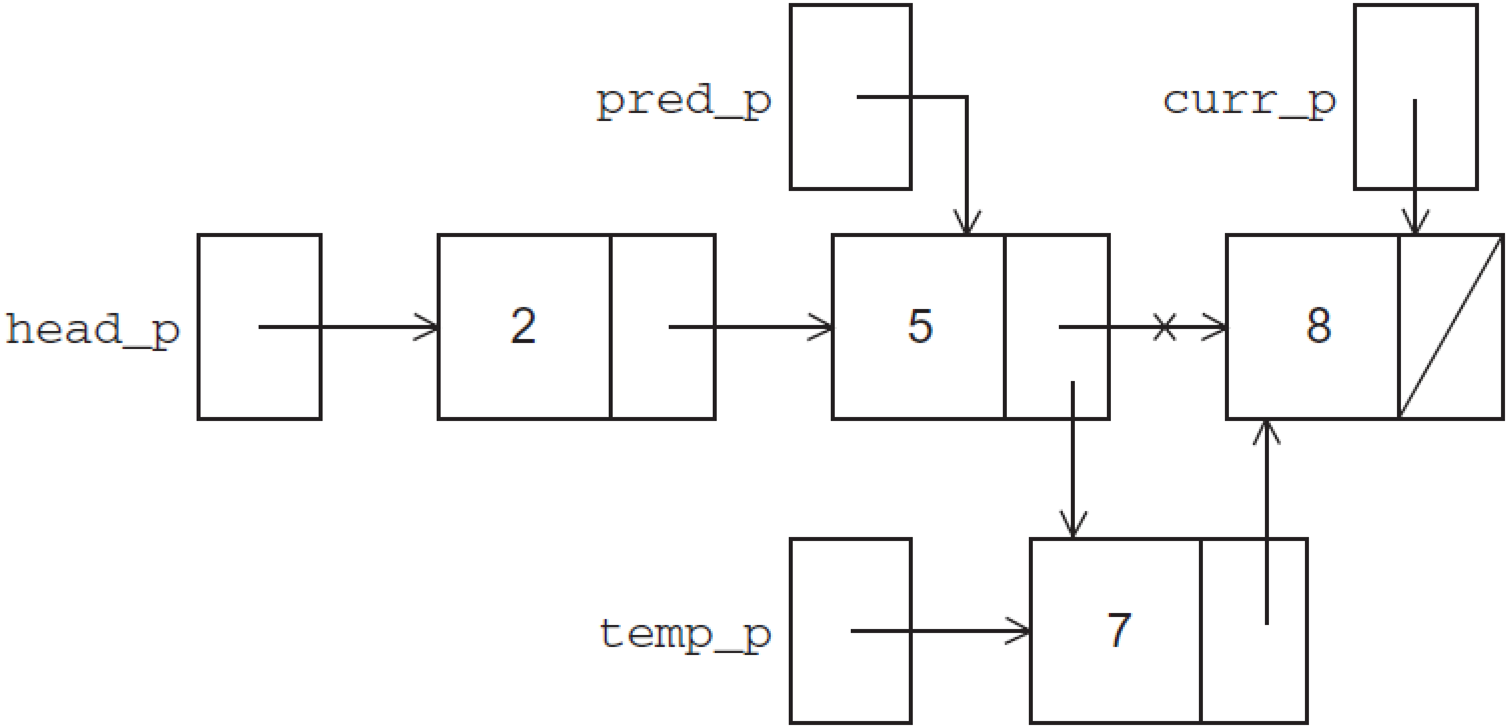
```
struct list_node_s {
   int data;
   struct list_node_s* next;
}
```

Membership operation for a linked list

```
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 operation: Inserting a new node

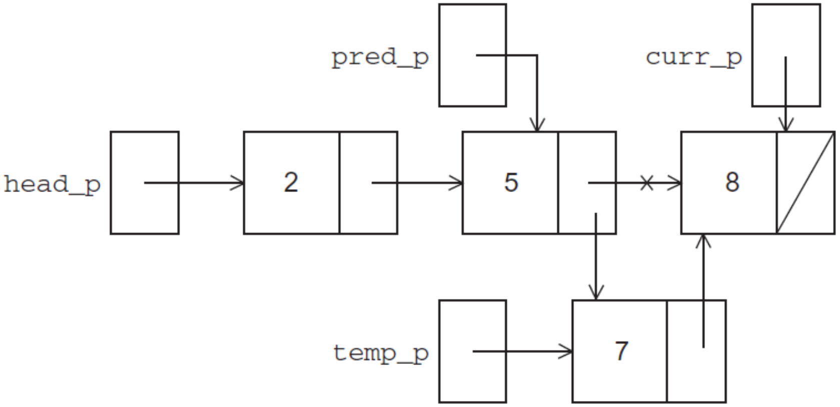


Inserting a new node into a list

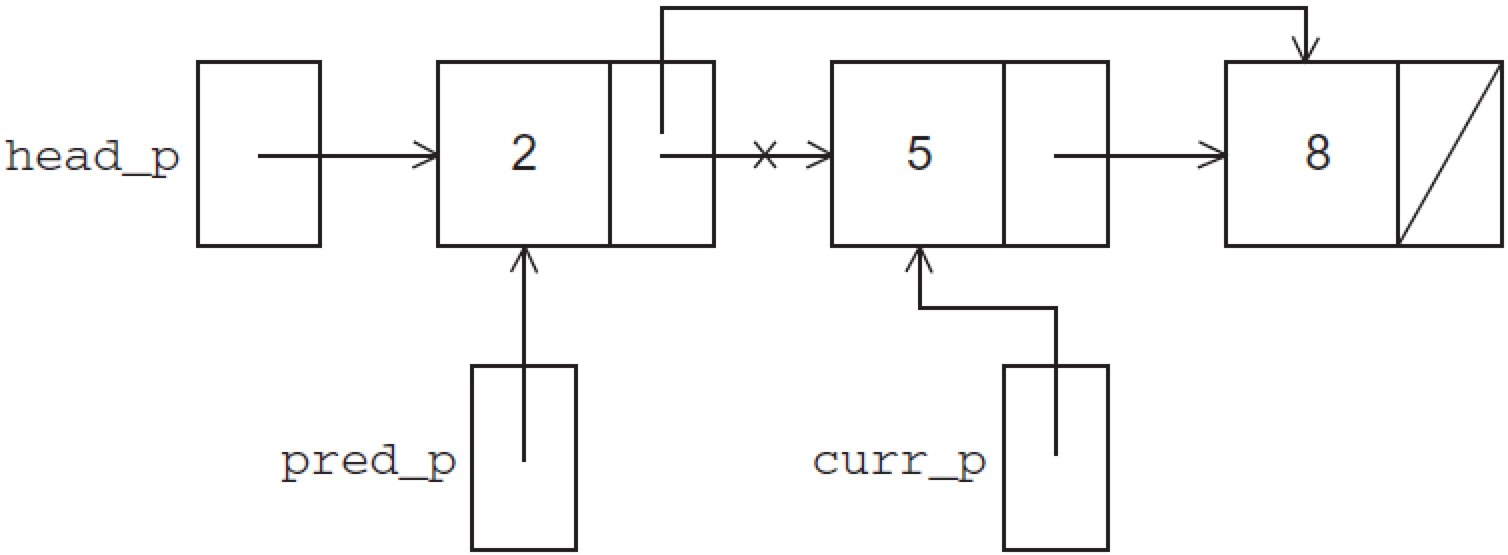
```
int Insert(int value, struct list_node_s** head_pp) {
   struct list_node_s* curr_p = *head_pp;
   struct list_node_s* pred_p = NULL;
   struct list_node_s* temp_p;

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

   if (curr_p == NULL || curr_p->data > value) {
      temp_p = malloc(sizeof(struct list_node
      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 */
```

Find the right position in the sorted list

Insert to this position



Delete operation: remove a node from a linked list

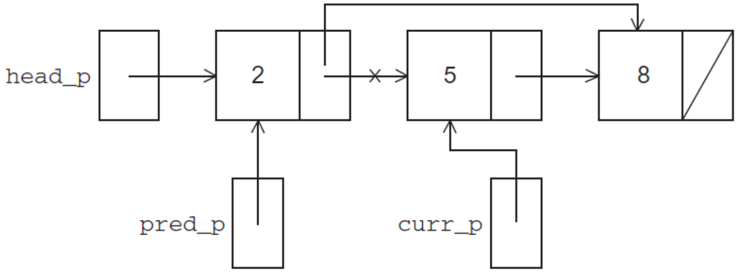


Deleting a node from a linked list

```
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
      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 */
```

Find a node with the given value

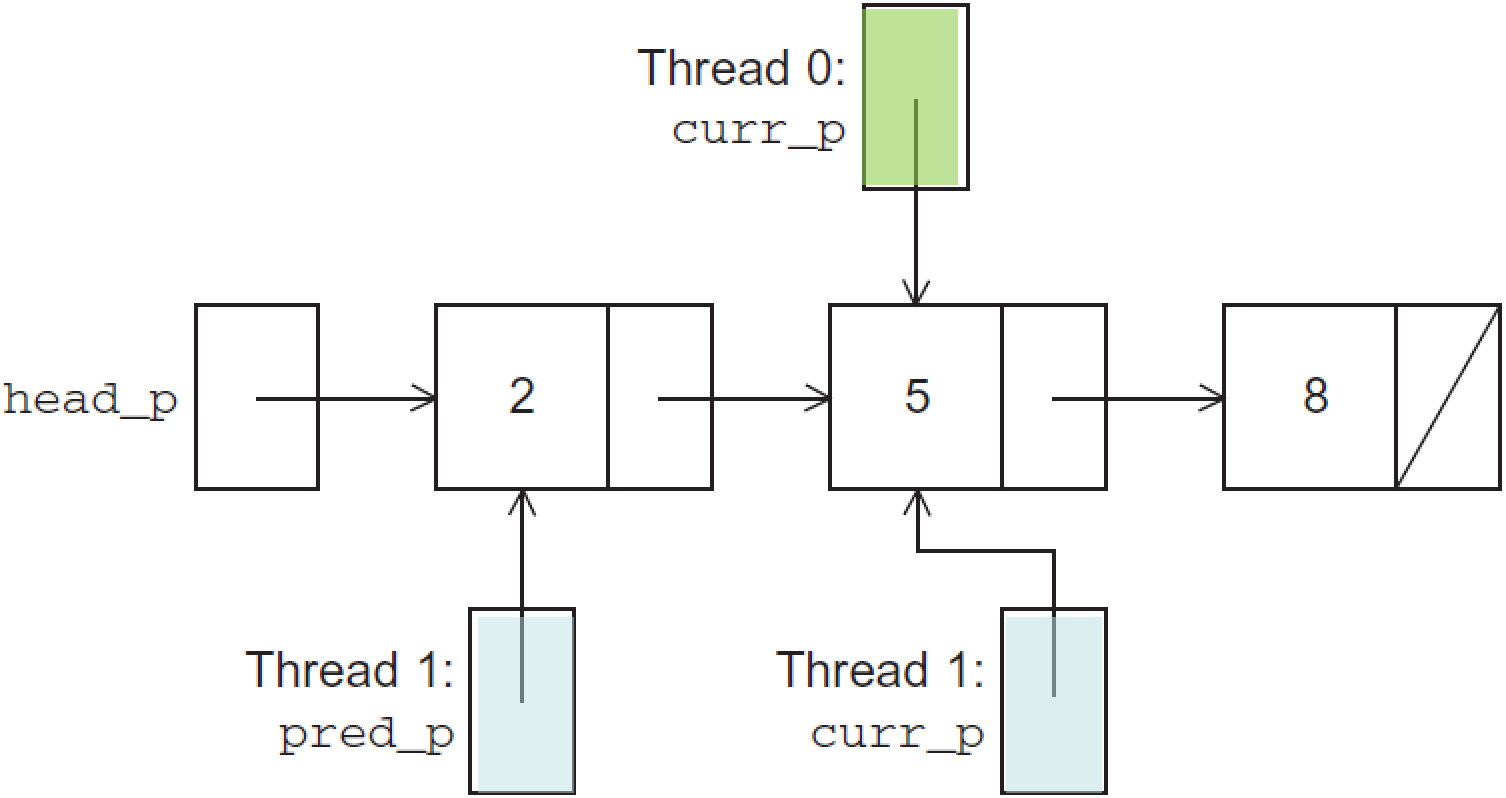
Remove this node



A Multi-Threaded Linked List

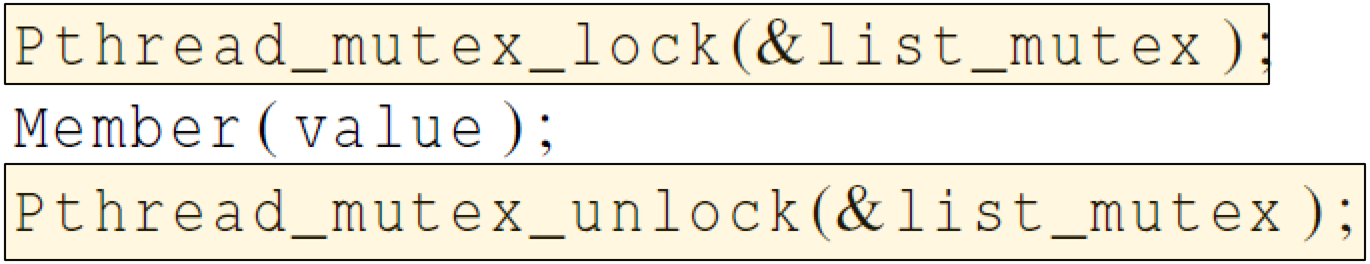
- Allow a sorted linked list to be accessed by multiple threads
- In order to share access to the list, define head_p to be a global variable.
  - This will simplify the function headers for Member, Insert, and Delete,
  - since we won’t need to pass in either head_p or a pointer to head_p: we’ll only need to pass in the value of interest.

Simultaneous access by two threads



Solution #1

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



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

In place of calling Member(value).

Issues

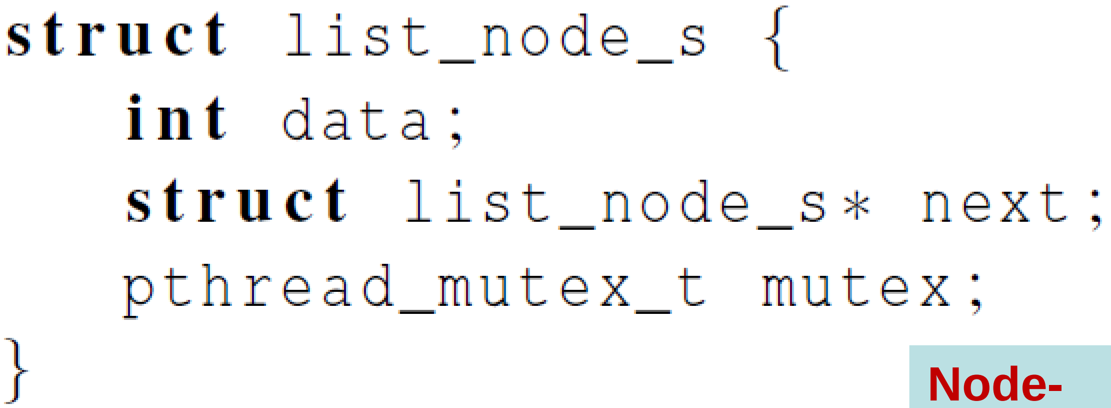
- We’re serializing access to the list.
- If the vast majority of our operations are calls to Member, we’ll fail to exploit this opportunity for parallelism.
- On the other hand, if most of our operations are calls to Insert and Delete,
  - This may be the bes
    - since serializati of performance im
    - Easy to implem

List-level	Member	Insert	Delete
Member	no	no	no
Insert	no	no	no
Delete	no	no	no

Mem			
Inser	?	?	?
Delete	?	?	?

Solution #2

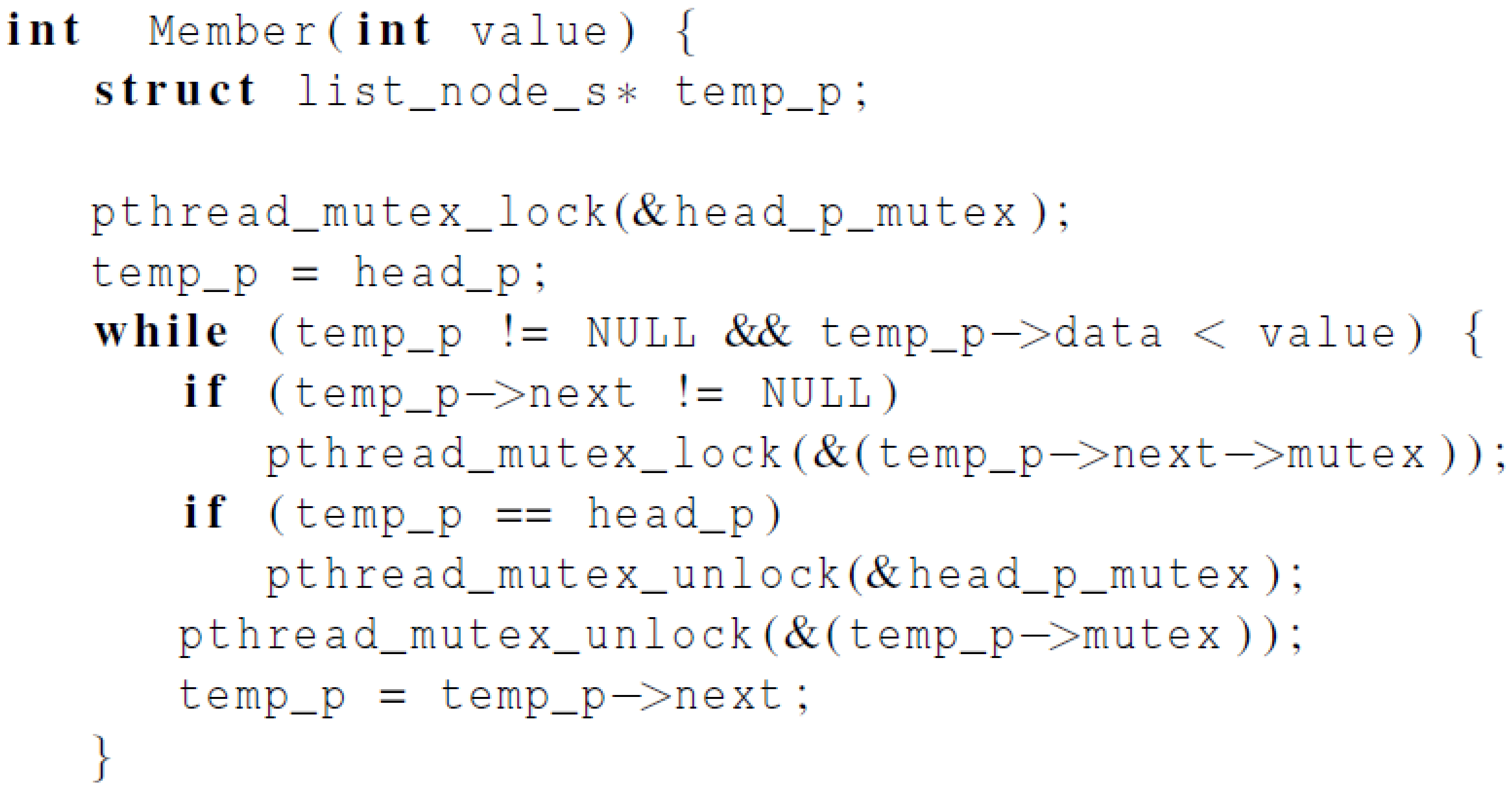
- **Instead of locking the entire list, lock individual nodes.**
  - A “finer-grained” approach: One mutex lock per node



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

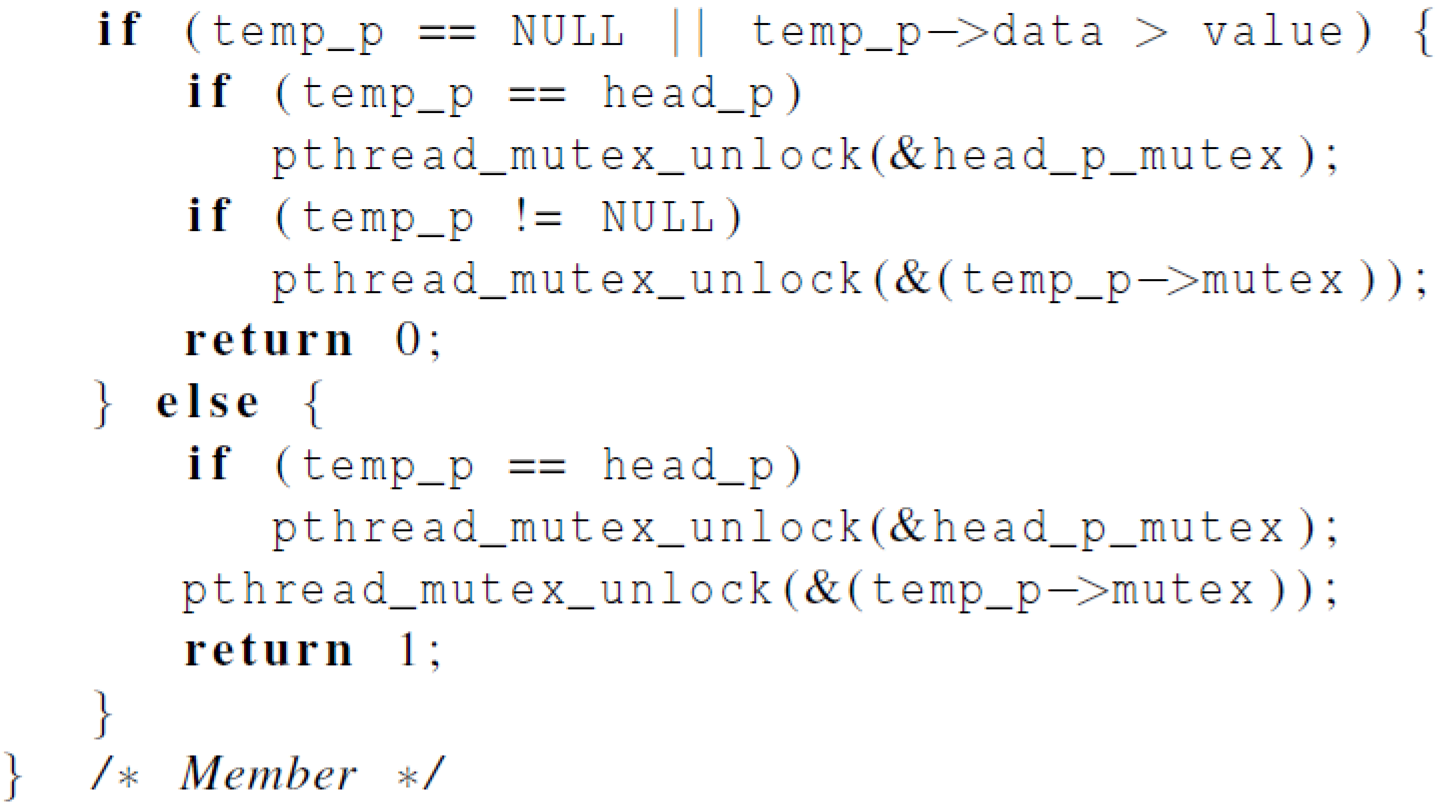
Node-level	Member	Insert	Delete
Member	no	no	no
Insert	no	no	no
Delete	no	no	no

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 */
```

Issues

- Much more complex than the original Member function.
- Much slower,
  - each time a node is accessed, a mutex must be locked and unlocked.
- Significant space cost
  - Adding a mutex field to each node

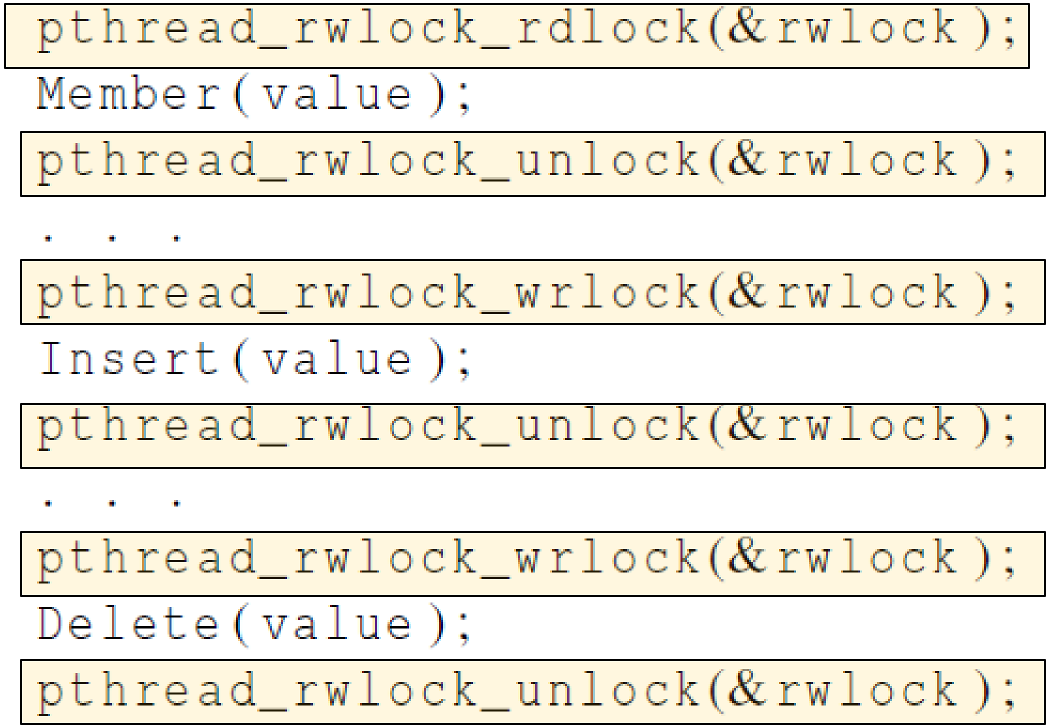
Motivation for using Pthreads Read-Write Locks

- 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.
- Example for

a linked list



```
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);
```

	Member	Insert	Delete
Member	?	?	?
Insert	?	?	?
Delete	?	?	?

Pthreads Read-Write Locks

- 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.
- 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 loc
threads that want to obtai
or writing will block in thei
functions.

List-level	Member	Insert	Delete
Member	yes	no	no
Insert	no	no	no
Delete	no	no	no

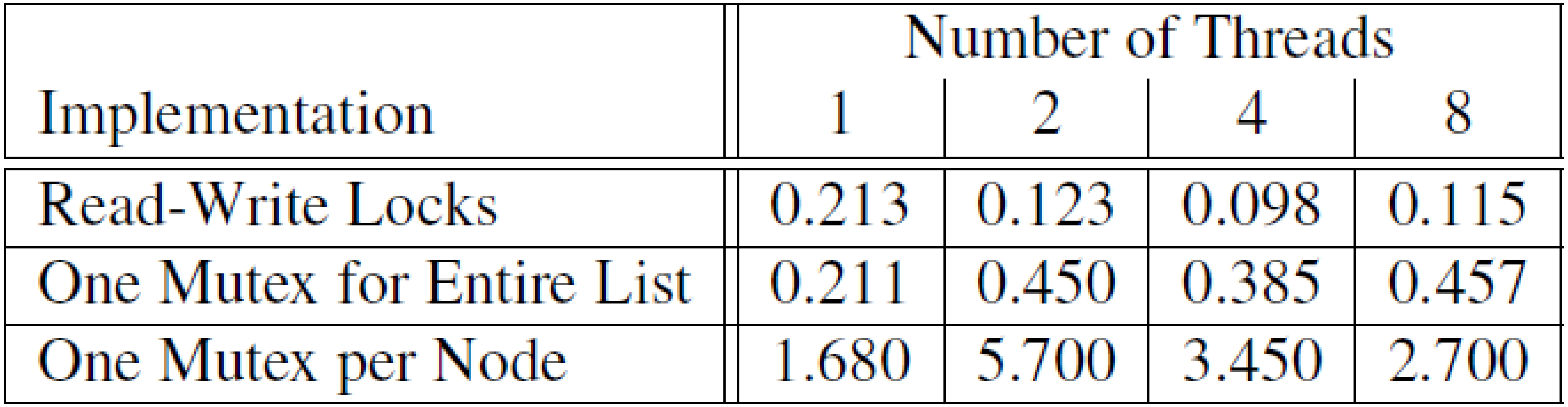
A performance comparison of 3 implementations for a linked list

Total time in second for executing 100,000 operations.

99.9% Member

0.05% Insert

0.05% Delete



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

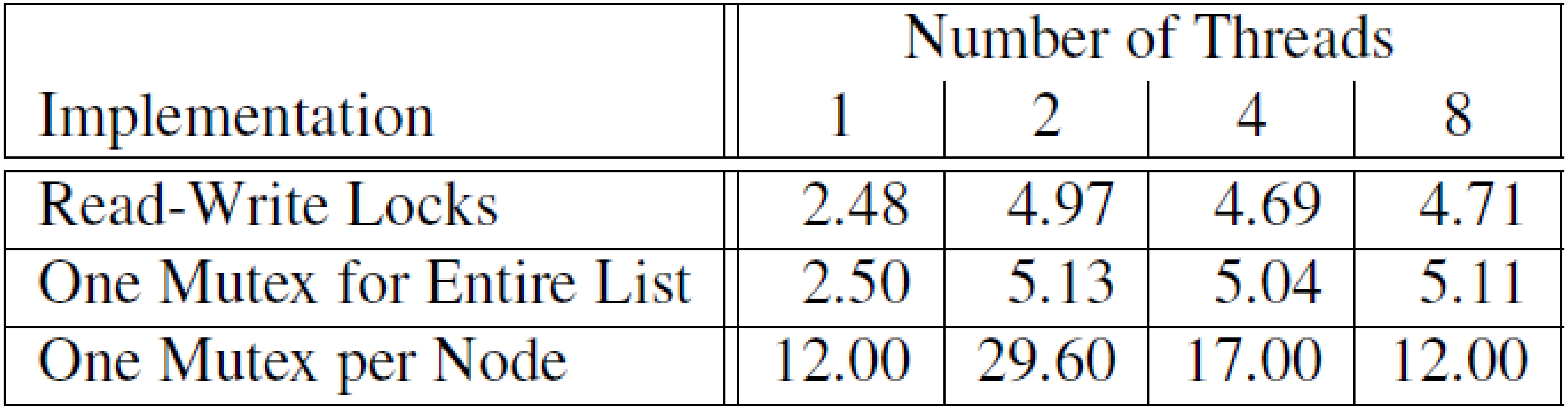
Linked List Performance: Comparison

Total time in seconds for executing 100,000 operations

80% Member

10% Insert

10% Delete



	Number of Threads			
Implementation	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

Issues with Threads: False Sharing, Deadlocks, Thread-safety

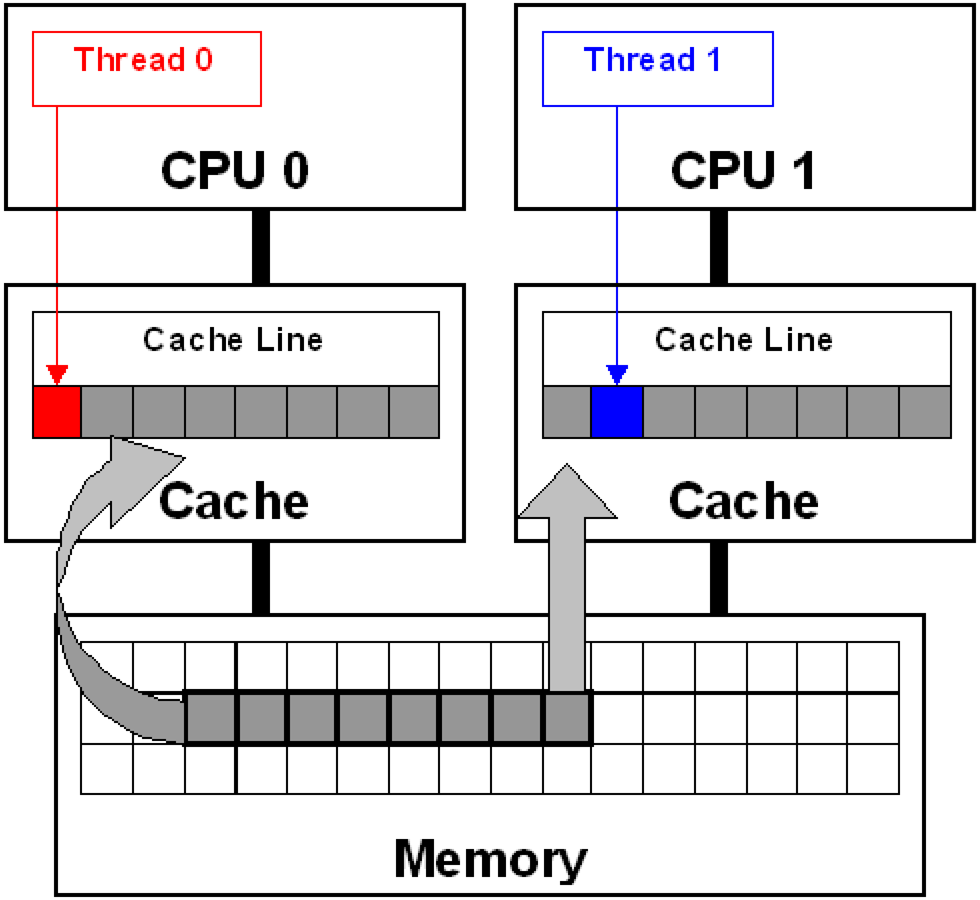
Caches, Cache-Coherence, and False Sharing

- Underlying cache-memory interaction can have a significant impact on shared-memory program performance in some cases.
- Cache fetches data

with a *cacheline* as a unit.

Cachline=128 bytes in

Intel Xeon.



Problem: False Sharing

- **Occurs when two or more processors/cores access different data in same cache line, and at least one of them writes.**
  - Leads to ping-pong effect.
- **Let's assume we parallelize code with p=2:**

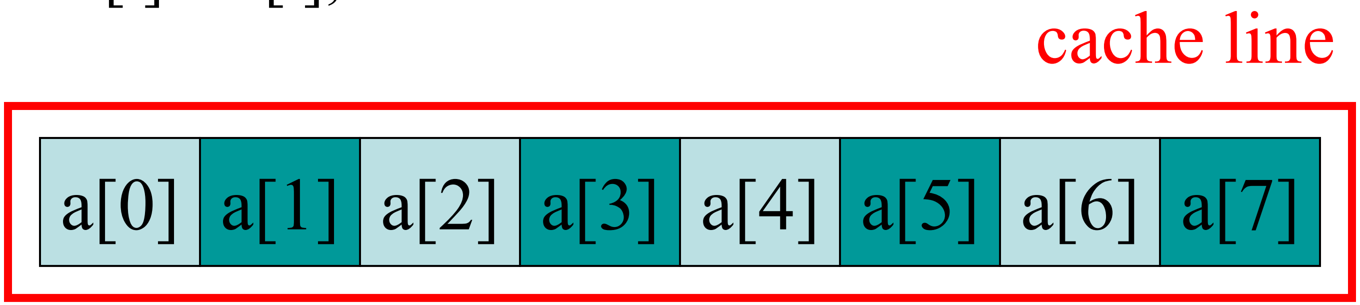
```
 for( i=0; i<n; i++ )
    a[i] = b[i];
```

  - Each array element takes 8 bytes
  - Cache line has 64 bytes (8 numbers)

False Sharing: Example (2 of 3)

Execute this program in two processors

```
for( i=0; i<n; i++ )
     a[i] = b[i];
```



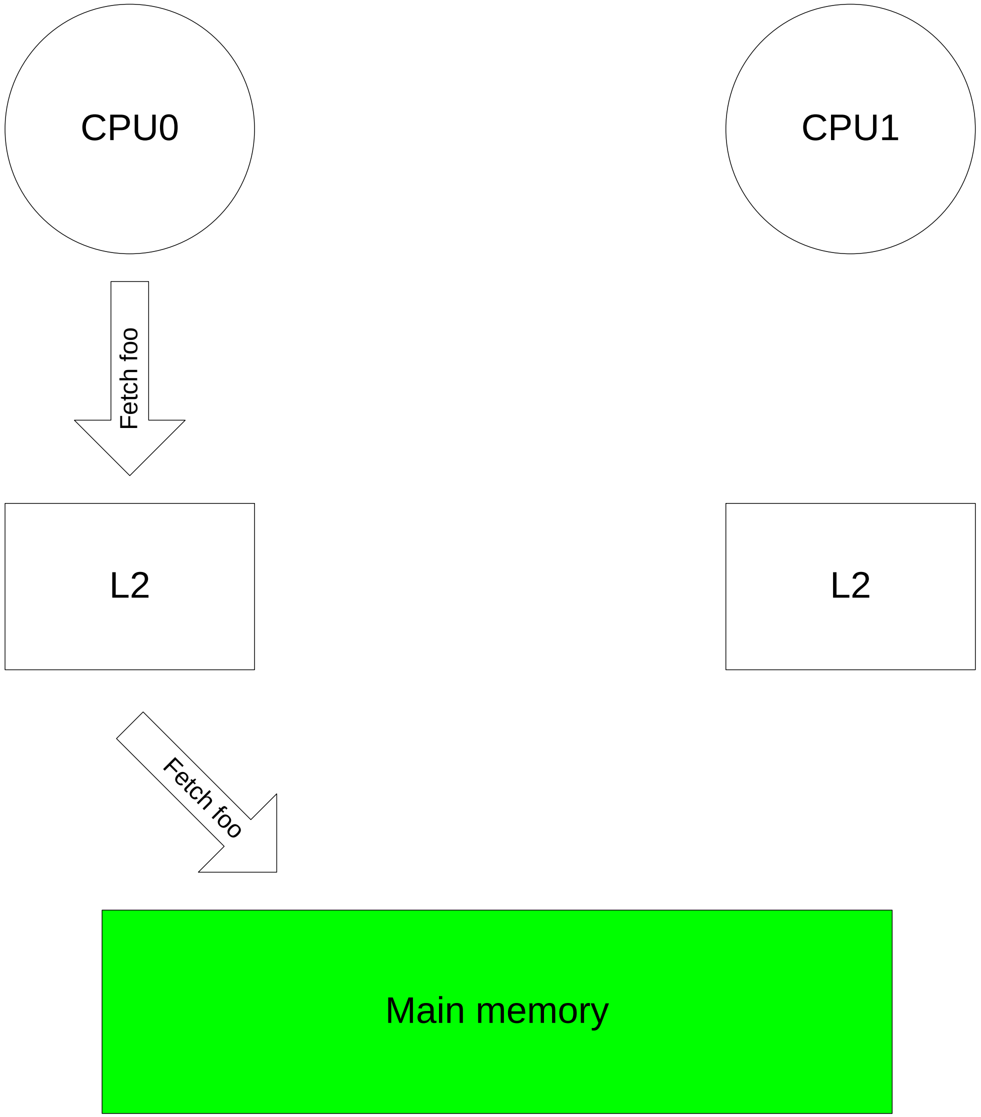
cache line

a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7]

Written by CPU 0

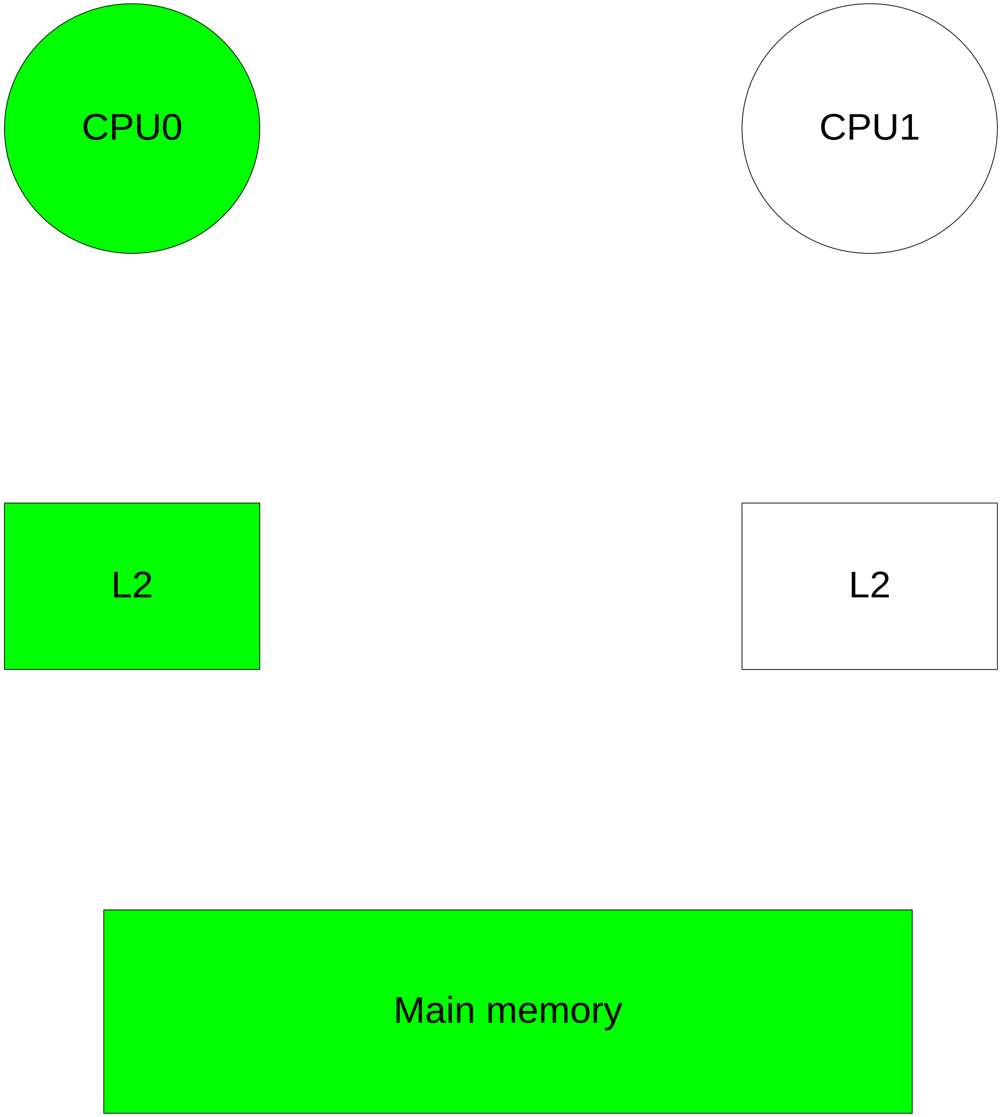
Written by CPU 1

False Sharing: Ping-Pong Effort of Cacheline Access

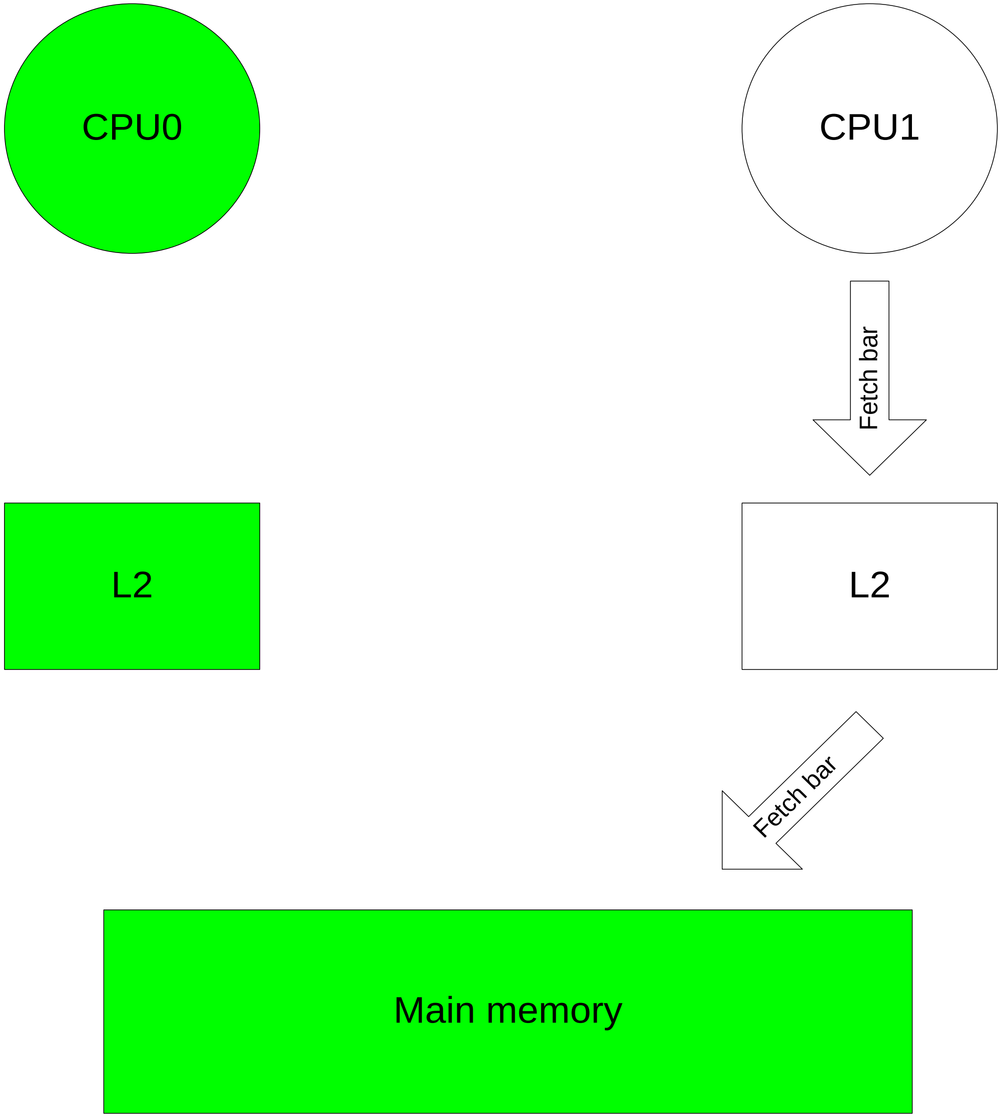


Source: Jeff Odom

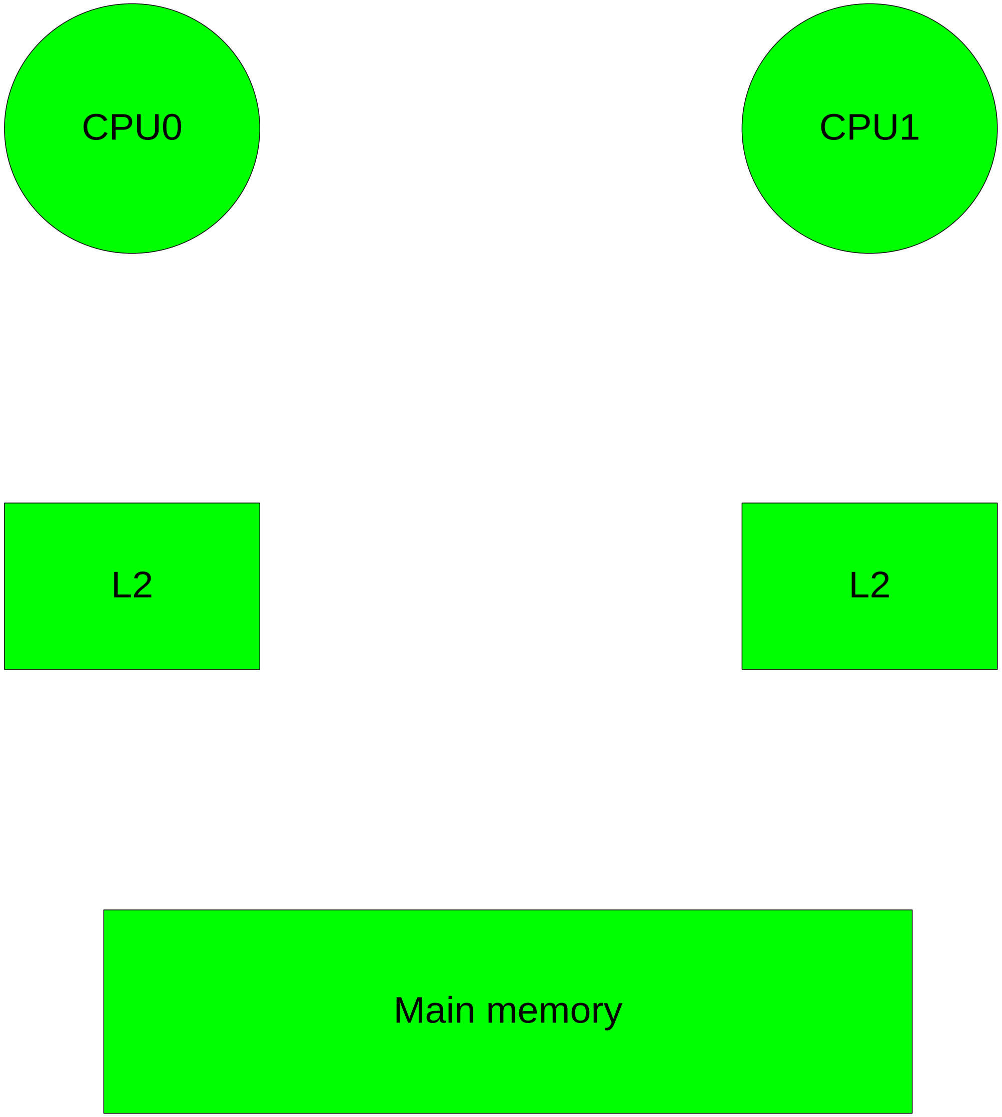
False Sharing : Ping-Pong Effort of Cacheline Access



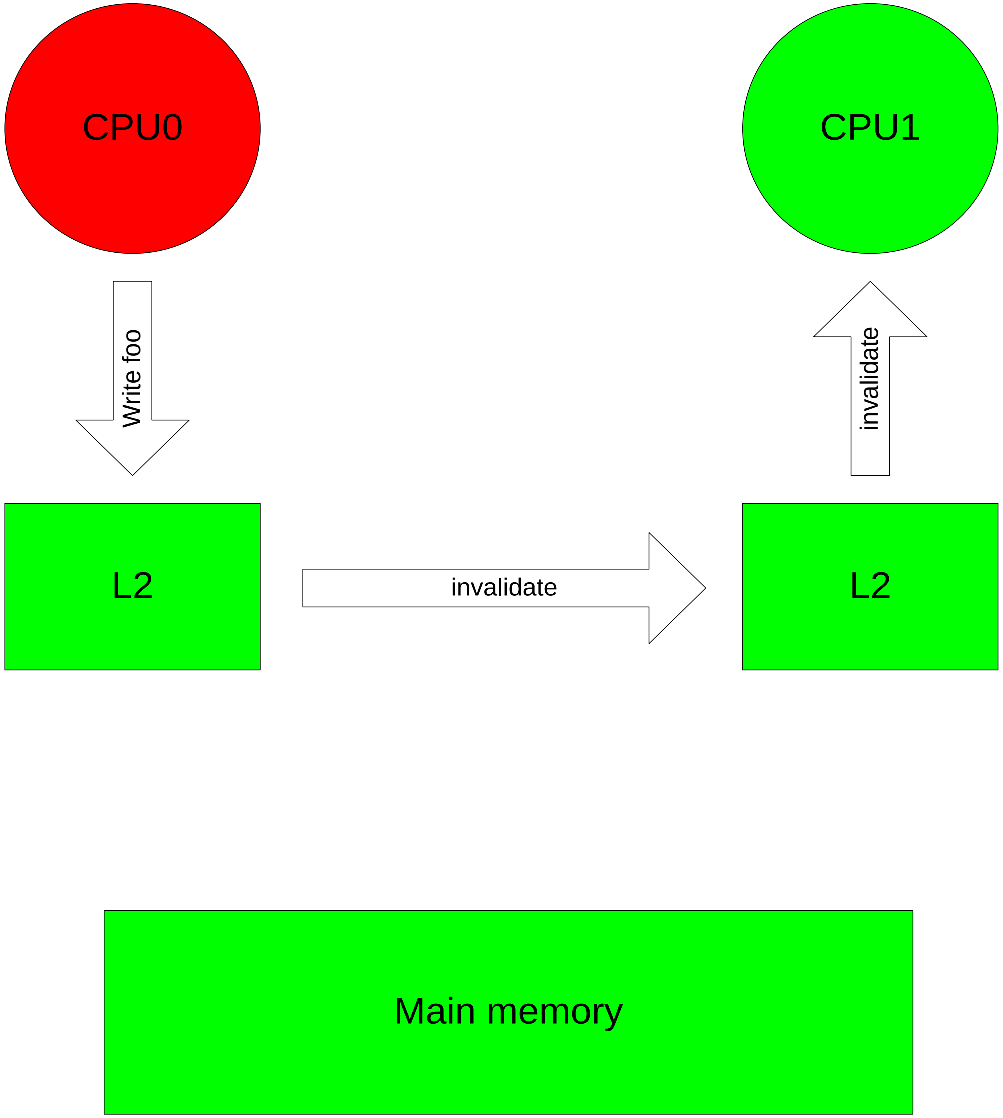
False Sharing : Ping-Pong Effort of Cacheline Access



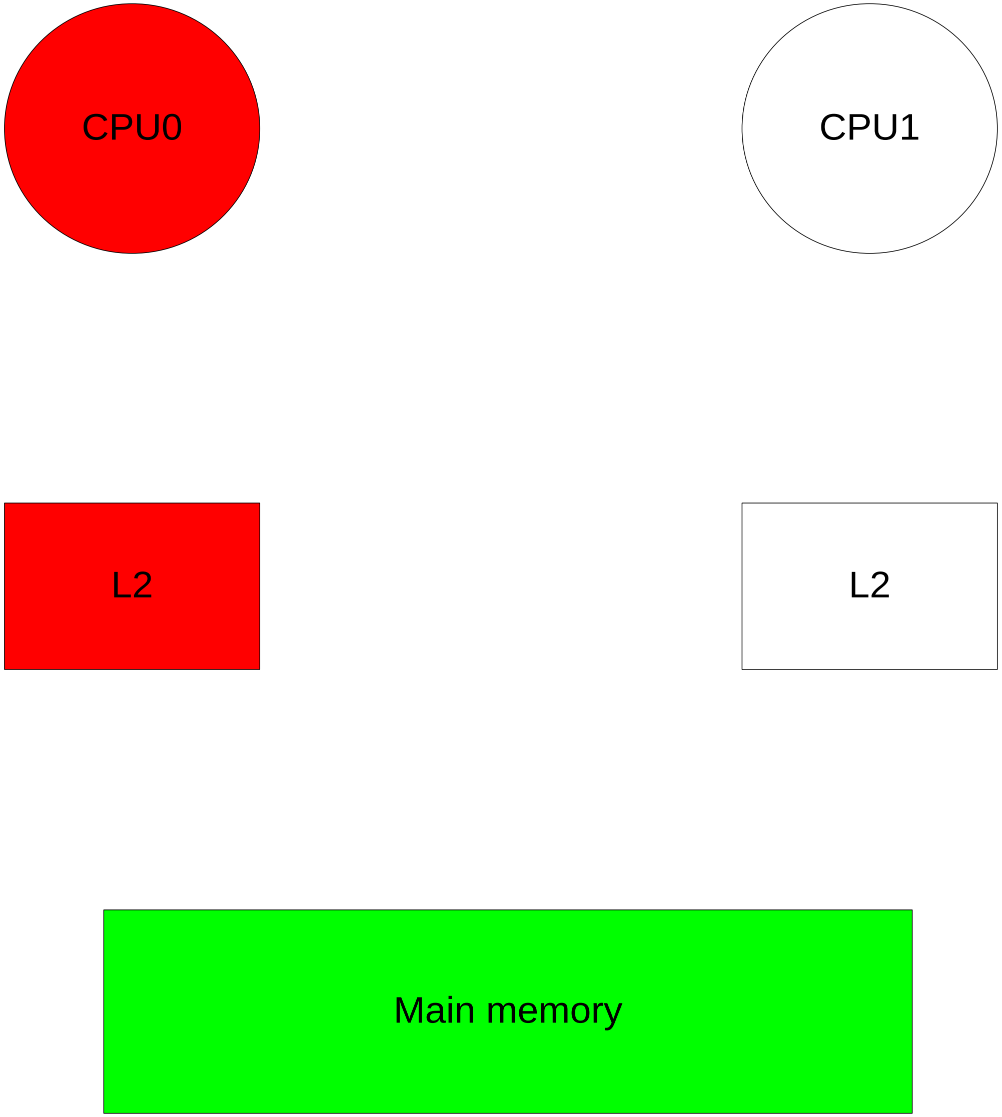
False Sharing : Ping-Pong Effort of Cacheline Access



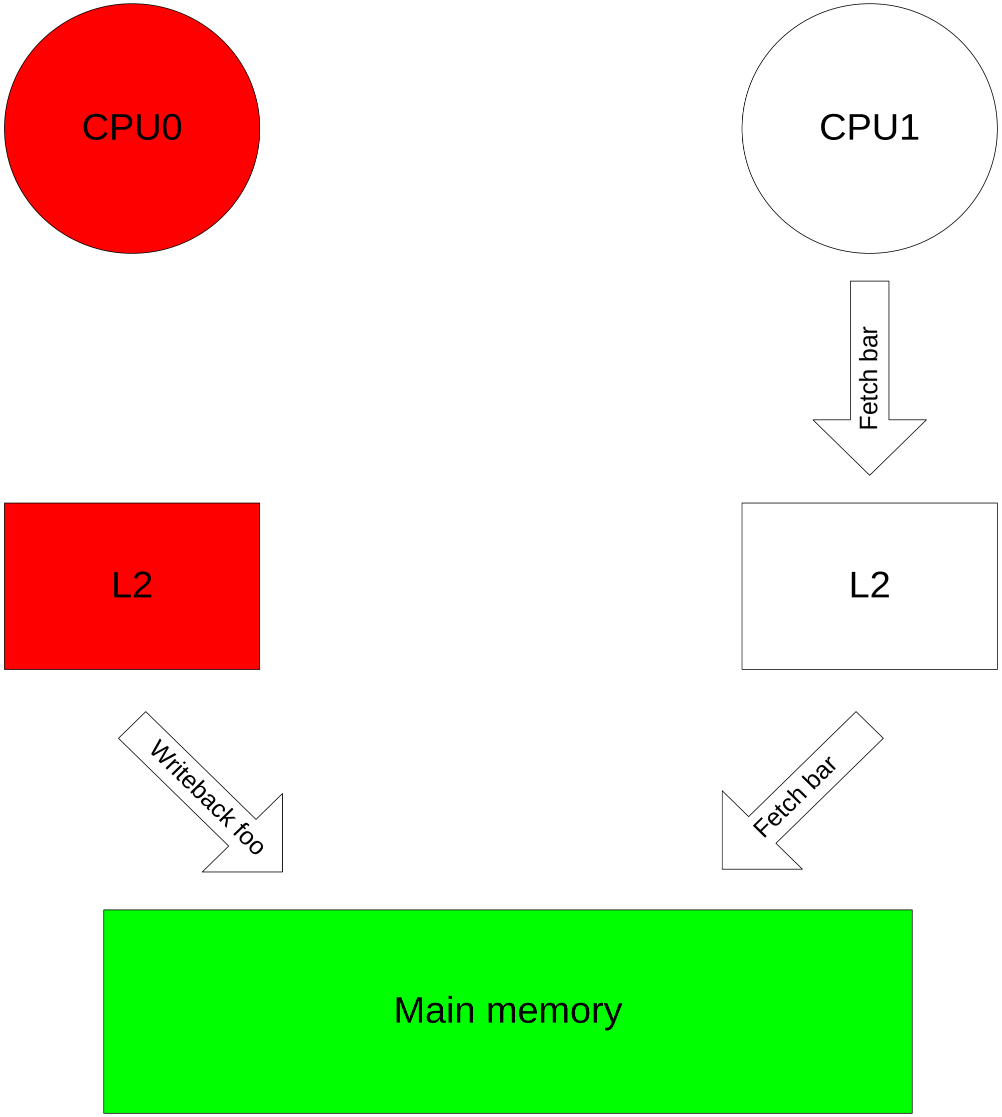
False Sharing : Ping-Pong Effort of Cacheline Access



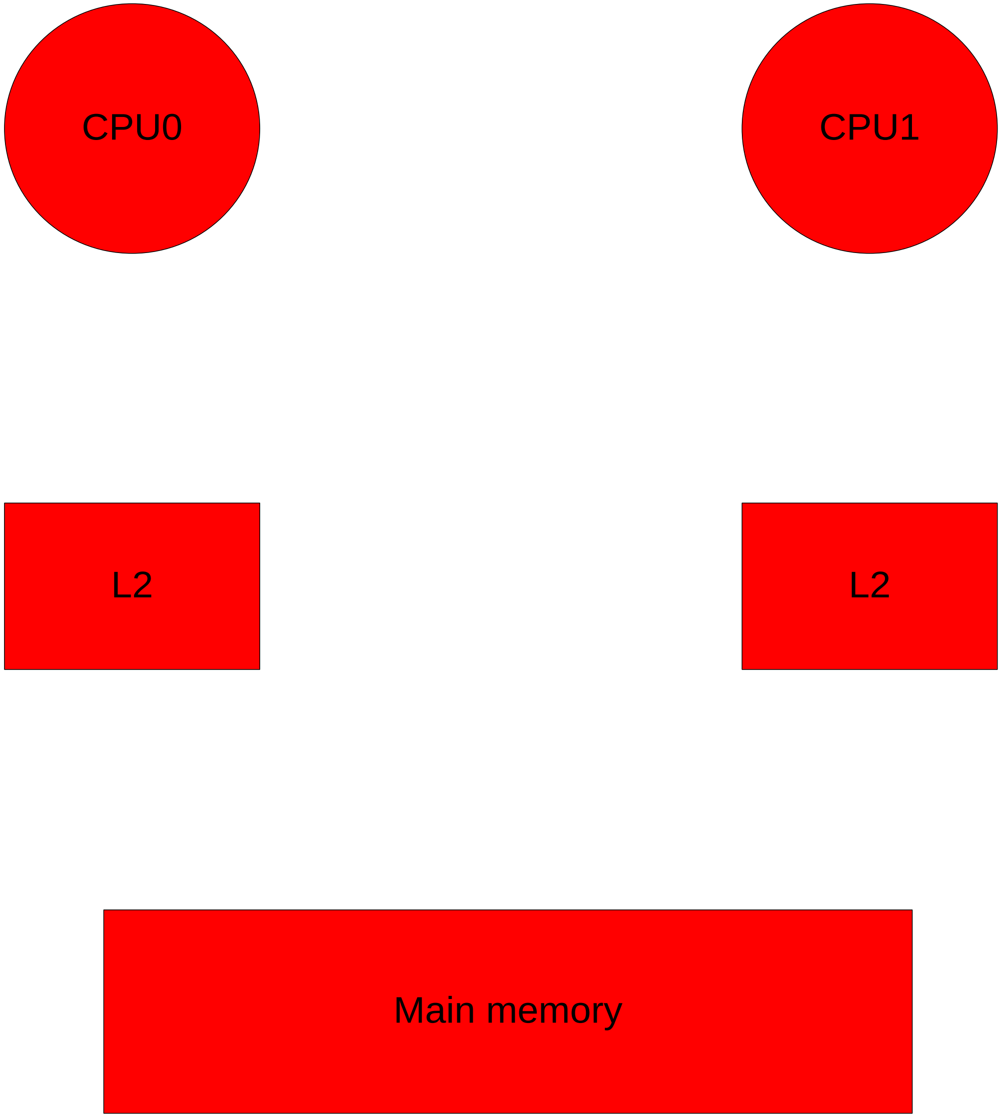
False Sharing : Ping-Pong Effort of Cacheline Access



False Sharing : Ping-Pong Effort of Cacheline Access



False Sharing : Ping-Pong Effort of Cacheline Access



False Sharing: Example

Two CPUs execute:

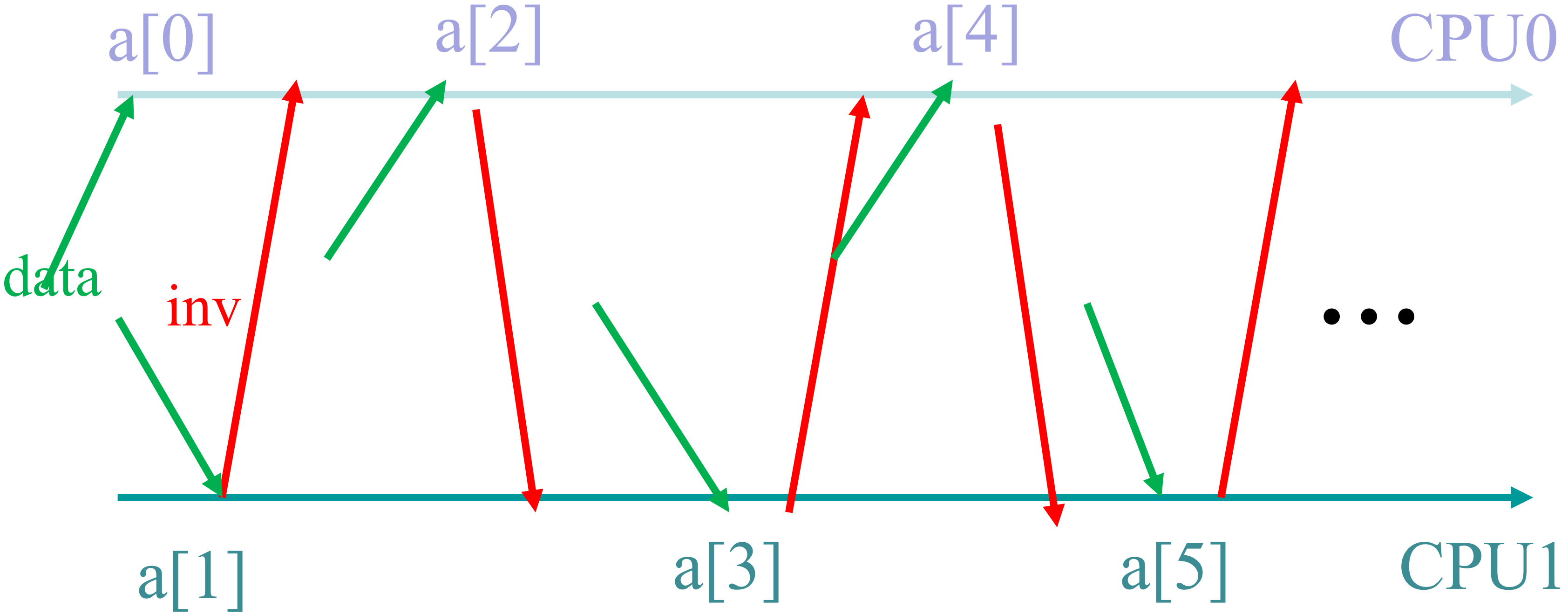
```
for( i=0; i<n; i++ )
    a[i] = b[i];
```

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]

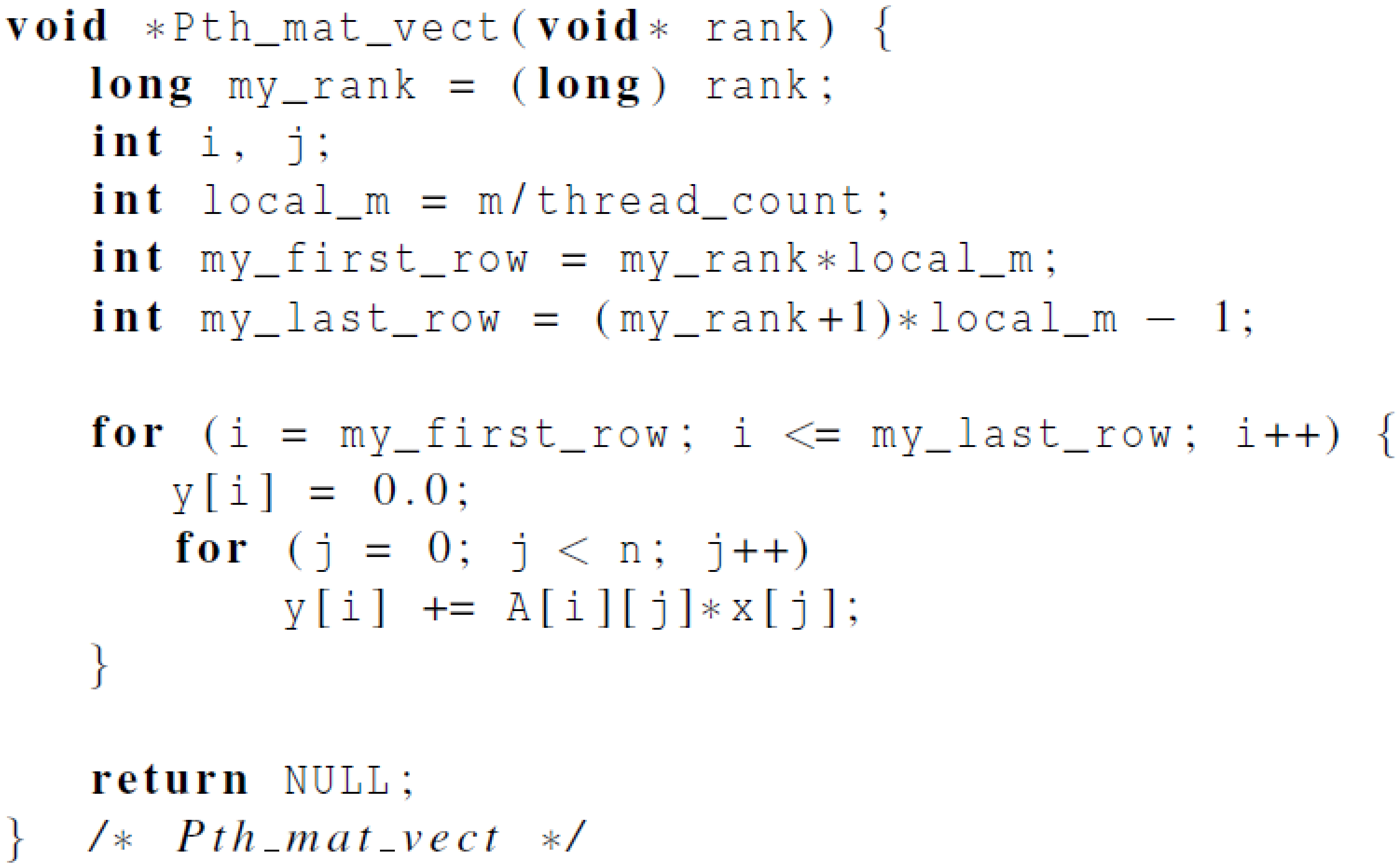
cache line

Written by CPU 0

Written by CPU 1

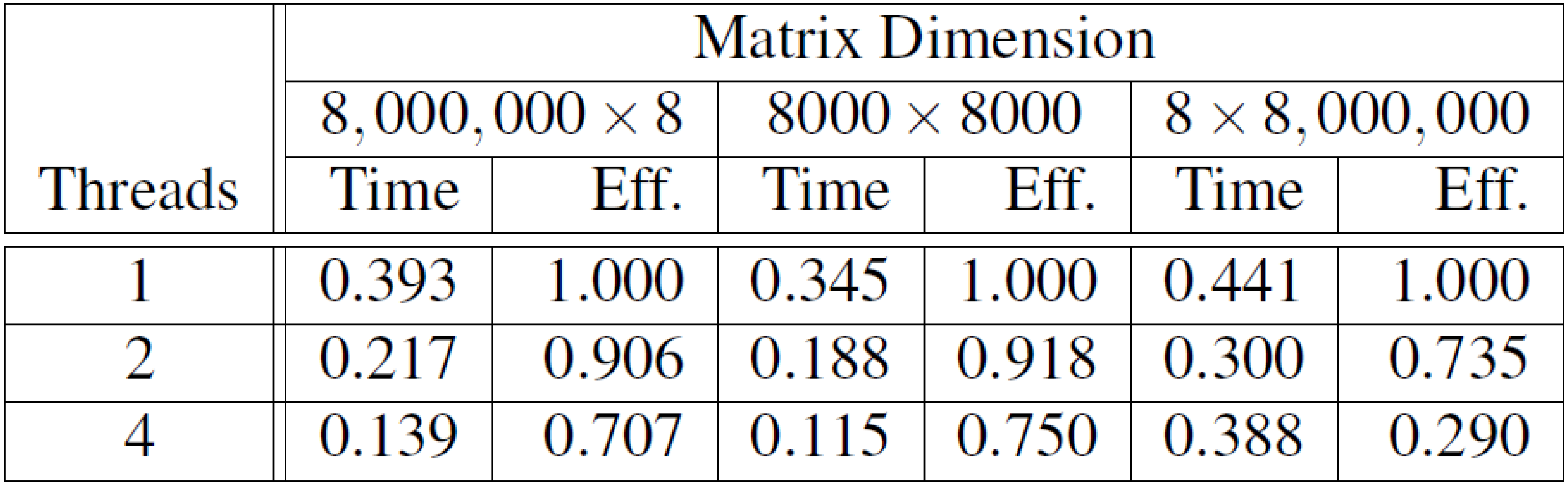


Block-based pthreads matrix-vector multiplication



```
void *Pth_mat_vect(void* rank) {
   long my_rank = (long) rank;
   int i, j;
   int local_m = m/thread_count;
   int my_first_row = my_rank*local_m;
   int my_last_row = (my_rank+1)*local_m - 1;

   for (i = my_first_row; i <= my_last_row; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++)
          y[i] += A[i][j]*x[j];
   }

   return NULL;
}  /* Pth_mat_vect */
```

Impact of false sharing on performance of matrix-vector multiplication



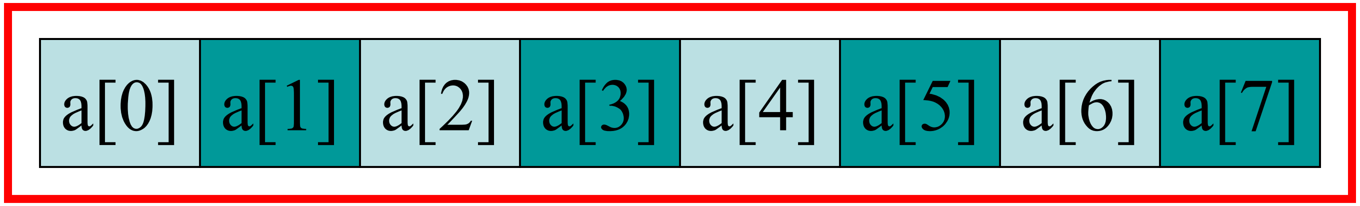
Threads	Matrix Dimension					
	8,000,000 × 8		8000 × 8000		8 × 8,000,000	
	Time	Eff.	Time	Eff.	Time	Eff.
1	0.393	1.000	0.345	1.000	0.441	1.000
2	0.217	0.906	0.188	0.918	0.300	0.735
4	0.139	0.707	0.115	0.750	0.388	0.290

(times are in seconds)

How to avoid false sharing?

Two CPUs execute:

```
for( i=0; i<n; i++ )
        a[i] = b[i];
```



a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]

cache line

- Avoid to write consecutive global variables from different threads
  - Use thread-specific local/private space as much as possible.
  - Pad frequently-modified global variables so they are not stored close to each other in memory and will not be held together within a cacheline.

Deadlock and Starvation

- **Deadlock** – two or more threads are waiting indefinitely for an event that can be only caused by one of these waiting threads
- **Starvation** – indefinite blocking (in a waiting queue forever).
  - Let S and Q be two mutex locks:

P_0	P_1
Lock(S);	Lock(Q);
Lock(Q);	Lock(S);
.	.
.	.
.	.
Unlock(Q);	Unlock(S);
Unlock(S);	Unlock(Q);

Deadlock Avoidance

- **Order the locks and always acquire the locks in that order.**
- **Eliminate circular waiting**

- :

P_0	P_1
Lock(S);	Lock(S);
Lock(Q);	Lock(Q);
.	.
.	.
.	.
Unlock(Q);	Unlock(Q);
Unlock(S);	Unlock(S);

Thread-Safety

- A block of code is thread-safe if it can be simultaneously executed by multiple threads without causing problems.
- When you program your own functions, you know if they are safe to be called by multiple threads or not.
- You may forget to check if system library functions used are thread-safe.
  - Unsafe function: strtok()from C string.h library
  - Other example.
    - The random number generator random in stdlib.h.
    - The time conversion function localtime in time.h.

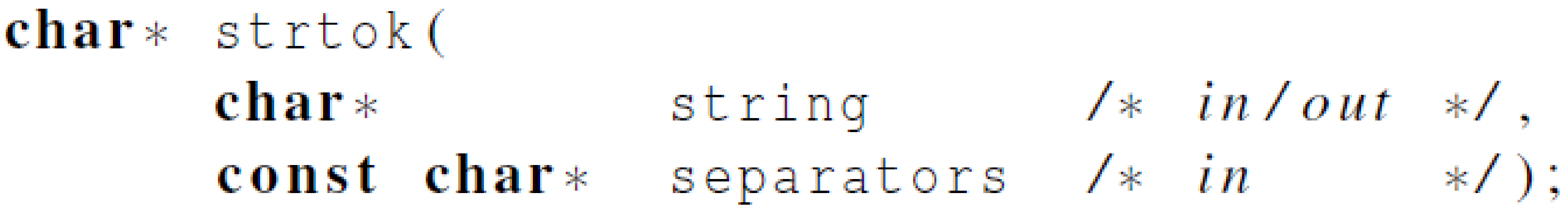
Example of using strtok()

- **“Tokenize” a English text file**
  - Tokens are contiguous sequences of characters separated by a white-space, a tab, or a newline.
  - Example: “Take UCSB CS140”

  → Three tokens: “Take”, “UCSB”, “CS140”
- **Divide the input file into lines of text and assign the lines to the threads in a round-robin fashion.**
  - Each thread tokenizes a line using strtok()
  - Line 1 → thread 0, Line 2 → thread 1, . . . , the tth goes to thread t, the t +1st goes to thread 0, etc.
  - Serialize access to input lines using semaphores

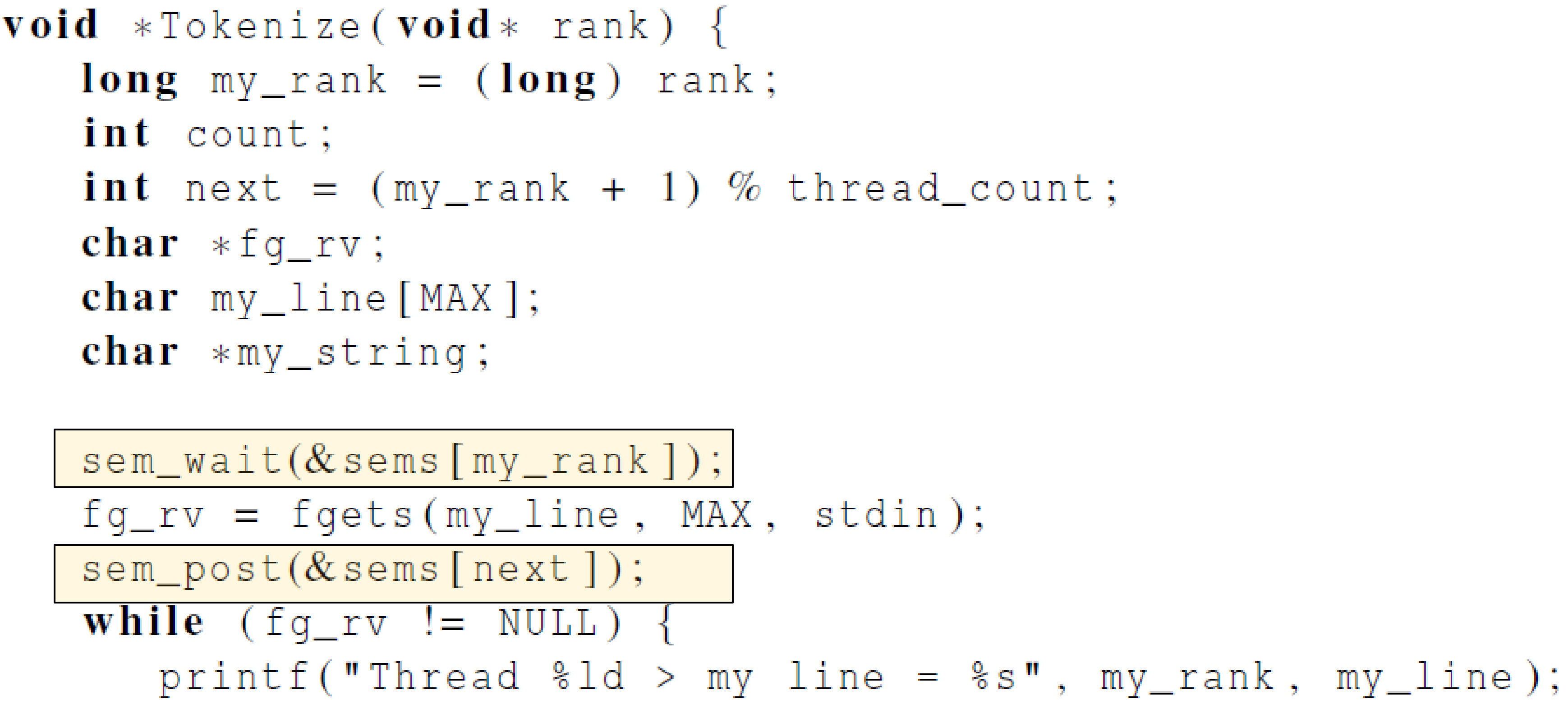
The strtok function

- **The first time it's called,**
  - the string argument is the text to be tokenized (Our line of input)
  - strtok caches a pointer to string
- **For subsequent calls, it returns successive tokens taken from the cached copy**
  - the first argument should be NULL.



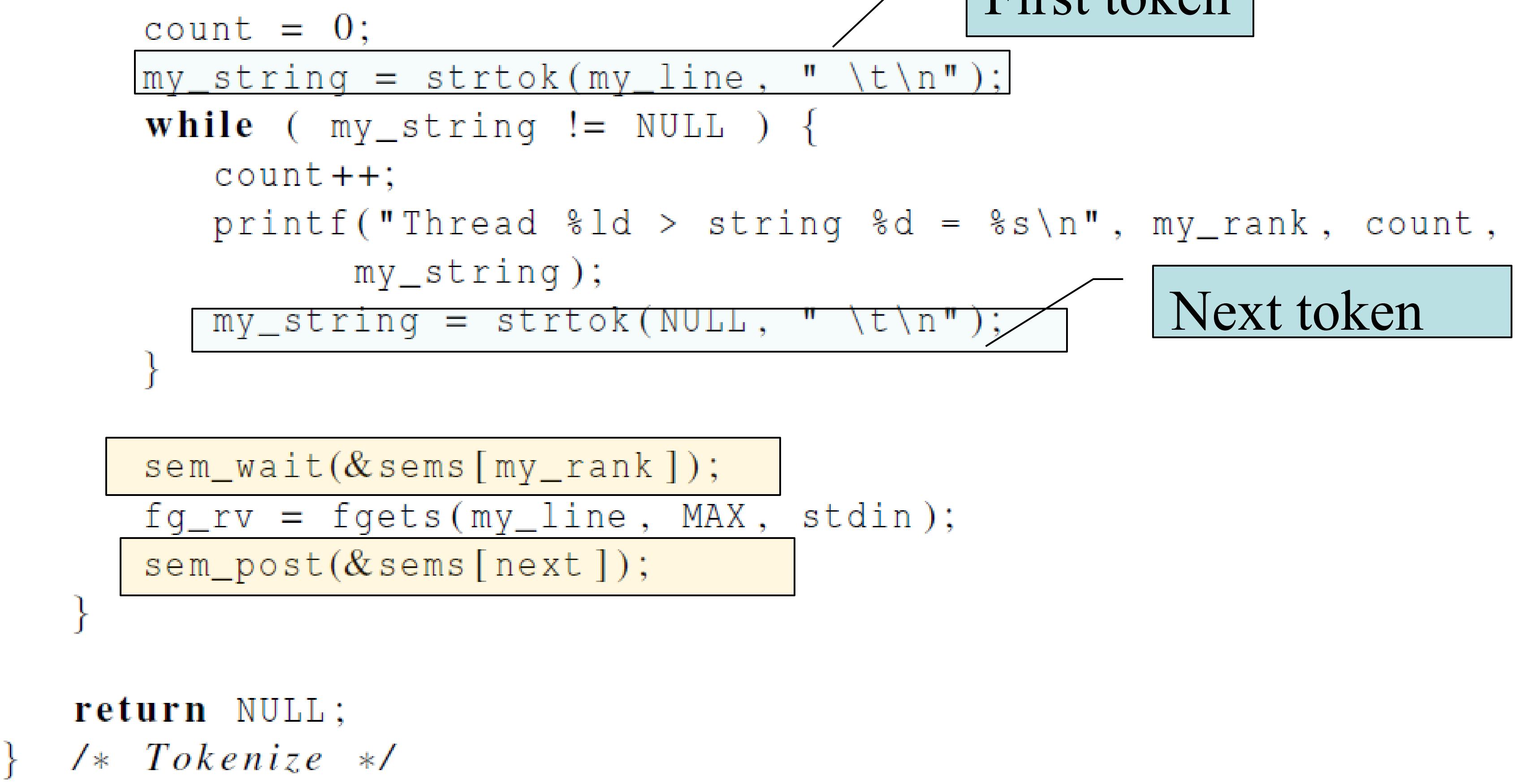
```
char* strtok(
      char*          string       /* in/out */,
      const char*    separators   /* in     */);
```

Multi-threaded tokenizer (1)



```
void *Tokenize(void* rank) {
   long my_rank = (long) rank;
   int count;
   int next = (my_rank + 1) % thread_count;
   char *fg_rv;
   char my_line[MAX];
   char *my_string;

   sem_wait(&sems[my_rank]);
   fg_rv = fgets(my_line, MAX, stdin);
   sem_post(&sems[next]);
   while (fg_rv != NULL) {
      printf("Thread %ld > my line = %s", my_rank, my_line);
```

Multi-threaded tokenizer (2)



```
      count = 0;
      my_string = strtok(my_line, " \t\n");
      while ( my_string != NULL ) {
         count++;
         printf("Thread %ld > string %d = %s\n", my_rank, count,
               my_string);
         my_string = strtok(NULL, " \t\n");
      }

      sem_wait(&sems[my_rank]);
      fg_rv = fgets(my_line, MAX, stdin);
      sem_post(&sems[next]);
   }

   return NULL;
}  /* Tokenize */
```

First token

Next token

Running with one thread

Input file:

Pease porridge hot.

Pease porridge cold.

Pease porridge in the pot

Nine days old.

- **It correctly tokenizes the input stream with 1 thread**

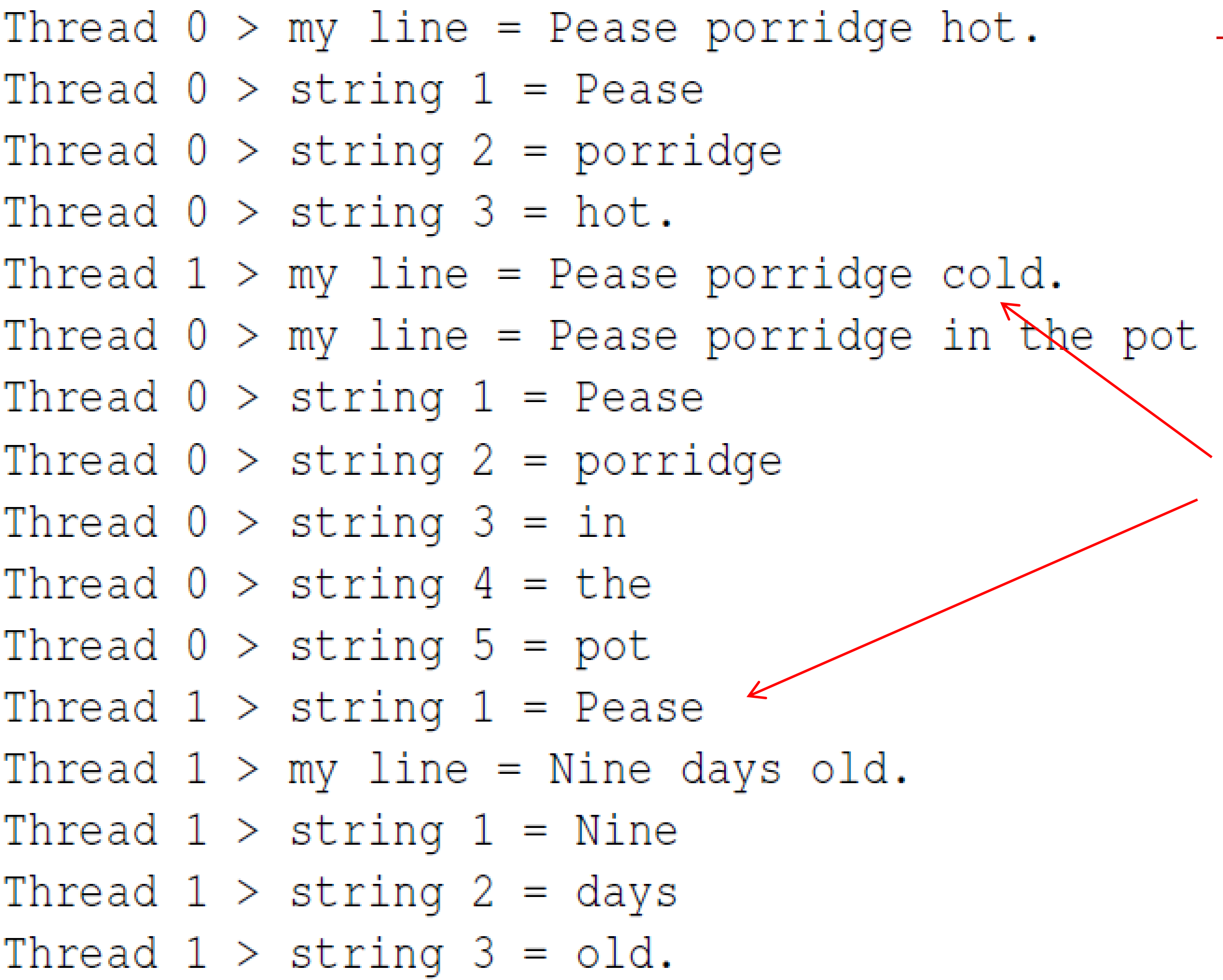
Pease

porridge

hot

. . .

Running with two threads



```
Thread 0 > my line = Pease porridge hot.
Thread 0 > string 1 = Pease
Thread 0 > string 2 = porridge
Thread 0 > string 3 = hot.
Thread 1 > my line = Pease porridge cold.
Thread 0 > my line = Pease porridge in the pot
Thread 0 > string 1 = Pease
Thread 0 > string 2 = porridge
Thread 0 > string 3 = in
Thread 0 > string 4 = the
Thread 0 > string 5 = pot
Thread 1 > string 1 = Pease
Thread 1 > my line = Nine days old.
Thread 1 > string 1 = Nine
Thread 1 > string 2 = days
Thread 1 > string 3 = old.
```

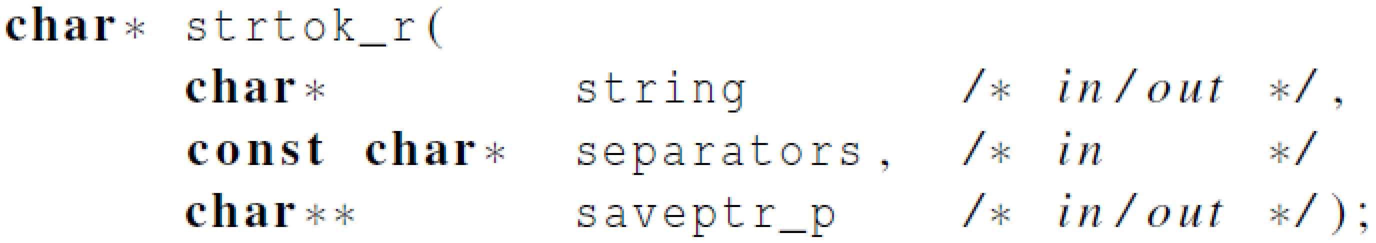
Oops!

What happened?

- strtok caches the input line by declaring a variable to have static (persistent) storage class.
  - Unfortunately this cached string is shared, not private.
- Thus, thread 0’s call to strtok with the third line of the input has apparently <u>overwritten</u> 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.

“re-entrant” (thread safe) functions

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



```
char* strtok_r(
        char*          string        /* in/out */,
        const char*    separators,   /* in     */
        char**         saveptr_p     /* in/out */);
```

Concluding Remarks

- A read-write lock is used when it’s safe for multiple threads to simultaneously read a data structure while only one write thread can access the data structure during the modification.
- False sharing happens when two threads/cores frequently read/write different data items stored in the same cacheline.
- Deadlocks can happen when using thread synchronization.
- Thread-safe functions.
  - Some thread-unsafe C functions cache data between calls by declaring variables to be static, causing errors when multiple threads call the function.