Parallel Programming

Shared-Memory Programming with Pthreads

(Part II)

Professor Yi-Ping You (游逸平)
Department of Computer Science
http://www.cs.nctu.edu.tw/~ypyou/

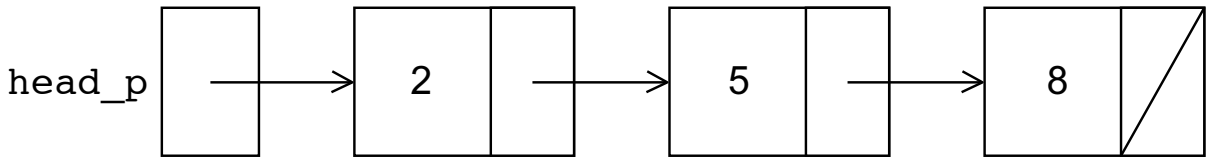
Outline

- A Multi-threaded Linked List
  - Using One Mutex for Entire List
  - Using One Mutex per Node
  - Using Read-Write Lock
  - Performance of the Various Implementations
- Caches, Cache Coherence, and False Sharing
- Thread-Safety

Controlling Access to Large, Shared Data Structures

- Let's take a look at the problem of controlling access to a large, shared data structure, which can be either simply searched or updated by the threads
  - E.g., a sorted linked list of `int`s with operations `Member`, `Insert`, and `Delete`

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



Linked List Functions: Member

- The Member function uses a pointer to traverse the list until it either finds the desired value or determines that the desired value cannot be in the 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 */
```

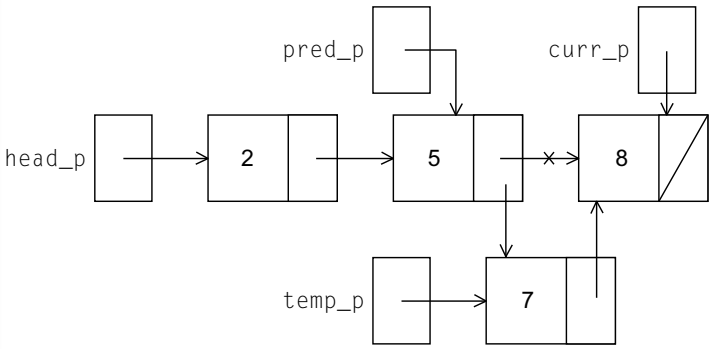
Linked List Functions: `Insert`

- The `Insert` function begins by searching for the correct position in which to insert the new node

```
int Insert(int value, struct list_node_s** head_p) {
   struct list_node_s* curr_p = *head_p;
   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_p = temp_p;
      else
         pred_p->next = temp_p;
      return 1;
   } else { /* Value already in list */
      return 0;
   }
}  /* Insert */
```



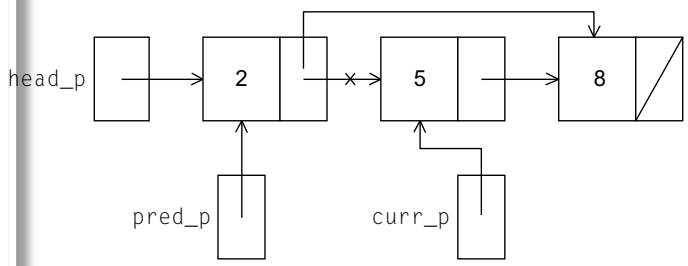
Linked List Functions: Delete

- The Delete function is similar to the Insert function in that it also needs to keep track of the predecessor of the current node while it's searching for the node to be deleted

```
int Delete(int value, struct list_node_s** head_p) {
   struct list_node_s* curr_p = *head_p;
   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_p = 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 Multithreaded Linked List

- Now 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
- What now are the consequences of having multiple threads simultaneously execute the three functions?
  - The `Member` function
    - Multiple threads can simultaneously *read* a memory location without conflict
    - It should be clear that multiple threads can simultaneously execute `Member`
  - The `Insert` and `Delete` functions
    - `Insert` and `Delete` *write* to memory locations, so there may be problems if we try to execute either of these operations at the same time as another operation

Outline

- A Multi-threaded Linked List
  - Using One Mutex for Entire List
  - Using One Mutex per Node
  - Using Read-Write Lock
  - Performance of the Various Implementations
- Caches, Cache Coherence, and False Sharing
- Thread-Safety

An Obvious Solution: Using a Mutex

- An obvious solution is to simply lock the list any time that a thread attempts to access it

```
pthread_mutex_lock(&list_mutex);
Member(value);
pthread_mutex_unlock(&list_mutex);
```

```
pthread_mutex_lock(&list_mutex);
Insert(value);
pthread_mutex_unlock(&list_mutex);
```

```
pthread_mutex_lock(&list_mutex);
Delete(value);
pthread_mutex_unlock(&list_mutex);
```

- Problem: we are 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

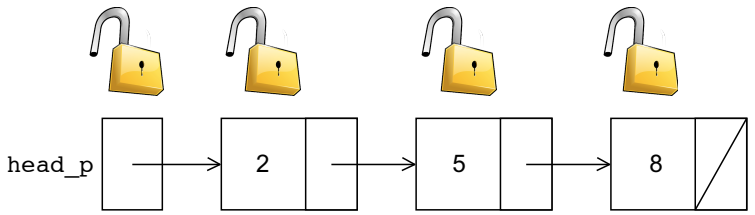
Outline

- A Multi-threaded Linked List
  - Using One Mutex for Entire List
  - Using One Mutex per Node
  - Using Read-Write Lock
  - Performance of the Various Implementations
- Caches, Cache Coherence, and False Sharing
- Thread-Safety

Another Solution: Finer-Grained Locking

- An alternative to this approach involves "finer-grained" locking
  - Instead of locking the entire list, we could try to lock individual nodes and also the `head_p`

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



Member Function for Finer-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 */
```

Finer-Grained Locking: Disadvantages

- This implementation is *much* more complex than the original Member function
- It is also much slower, in general, each time a node is accessed, a mutex must be locked and unlocked
- A further problem is that the addition of a mutex field to each node will substantially increase the amount of storage needed for the list

Outline

- A Multi-threaded Linked List
  - Using One Mutex for Entire List
  - Using One Mutex per Node
  - Using Read-Write Lock
  - Performance of the Various Implementations
- Caches, Cache Coherence, and False Sharing
- Thread-Safety

Pthreads Read-Write Locks

- A read-write lock is somewhat like a mutex except that it provides two lock functions
  - Locks the read-write lock for *reading*
  - Locks the read-write lock for *writing*
- 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
  - Furthermore, 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

Multithreaded Linked List Using Read-Write Locks

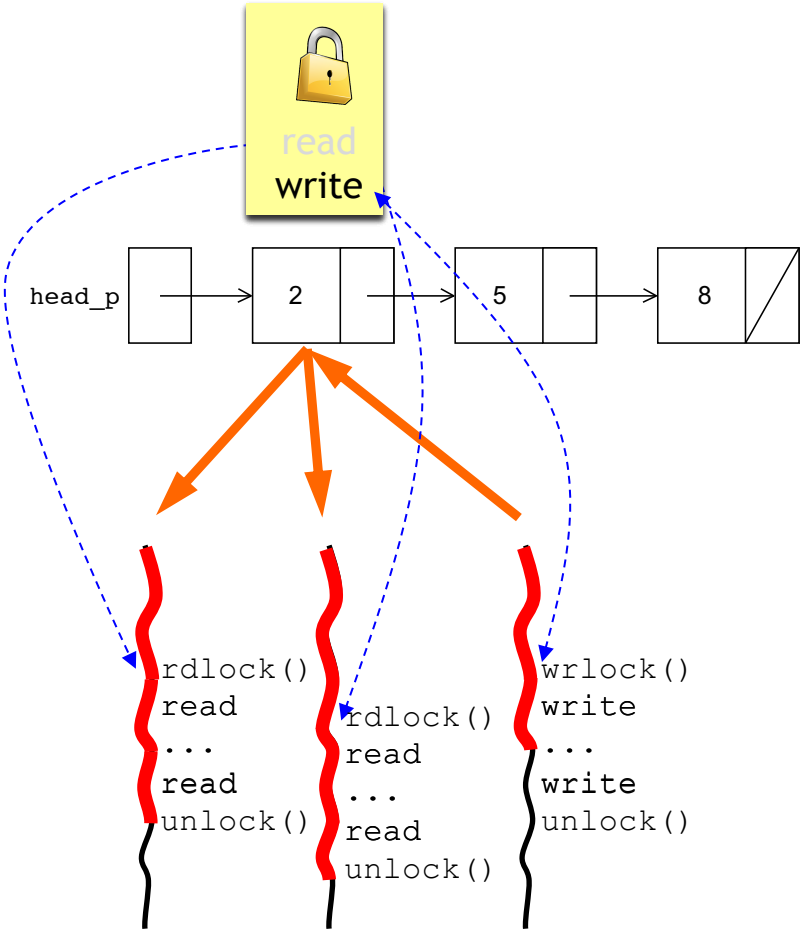
- Using Pthreads read-write locks, we can protect our linked list functions with the following code (we're ignoring function return values):

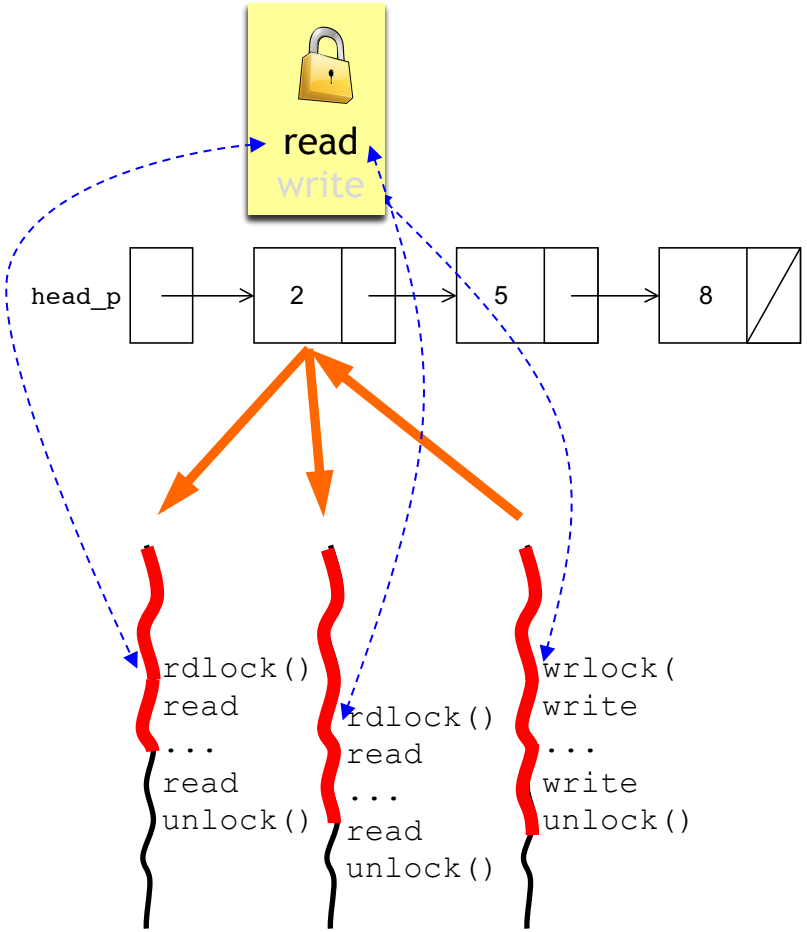
```
pthread_rwlock_rdlock(&rwlock);
Member(value);
pthread_rwlock_unlock(&rwlock);
```

```
pthread_rwlock_wrlock(&rwlock);
Insert(value);
pthread_rwlock_unlock(&rwlock);
```

```
pthread_rwlock_wrlock(&rwlock);
Delete(value);
pthread_rwlock_unlock(&rwlock);
```

Read-Write Locks: Examples





Initialization and Destroy

- As with mutexes, read-write locks should be initialized before use and destroyed after use:

```
int pthread_rwlock_init(
      pthread_rwlock_t*            rwlock_p  /* out */,
      const pthread_rwlockattr_t*  attr_p    /* in  */);
```

```
int pthread_rwlock_destroy(pthread_rwlock_t* rwlock_p /* in/out */);
```

Outline

- A Multi-threaded Linked List
  - Using One Mutex for Entire List
  - Using One Mutex per Node
  - Using Read-Write Lock
  - Performance of the Various Implementations
- Caches, Cache Coherence, and False Sharing
- Thread-Safety

Performance Comparison

- The main thread first inserts a number of randomly generated keys into an empty list
- After being started by the main thread, each thread carries out a number of operations (`Member`, `Insert`, `Delete`) on the list
- Four dual-core processors

Table 4.3 Linked List Times: 1000 Initial Keys, 100,000 ops, 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

Table 4.4 Linked List Times: 1000 Initial Keys, 100,000 ops, 80% `Member`, 10% `Insert`, 10% `Delete`

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

Discussions (1/3)

Table 4.3 Linked List Times: 1000 Initial Keys, 100,000 ops, 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

Table 4.4 Linked List Times: 1000 Initial Keys, 100,000 ops, 80% `Member`, 10% `Insert`, 10% `Delete`

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

- Single thread
  - The run-times for the read-write locks and the single-mutex implementations are about the same
    - The operations are serialized, and since there is no contention for the read-write lock or the mutex
  - The implementation that uses one mutex per node is *much* slower
    - Each time a single node is accessed there will be two function calls
      - one to lock the node mutex and one to unlock it

Discussions (2/3)

Table 4.3 Linked List Times: 1000 Initial Keys, 100,000 ops, 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

Table 4.4 Linked List Times: 1000 Initial Keys, 100,000 ops, 80% `Member`, 10% `Insert`, 10% `Delete`

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

- Multiple threads
  - The inferiority of the implementation that uses one mutex per node persists
  - When there are very few `Insert`s and `Delete`s, the read-write lock implementation is far better than the single-mutex implementation
    - The single-mutex implementation will serialize all the operations
    - If there are very few `Insert`s and `Delete`s, the read-write locks do a very good job of allowing concurrent access to the list
  - On the other hand, if there are a relatively large number of `Insert`s and `Delete`s, there's very little difference between the performance of the read-write lock implementation and the single-mutex implementation

Discussions (3/3)

Table 4.3 Linked List Times: 1000 Initial Keys, 100,000 ops, 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

Table 4.4 Linked List Times: 1000 Initial Keys, 100,000 ops, 80% `Member`, 10% `Insert`, 10% `Delete`

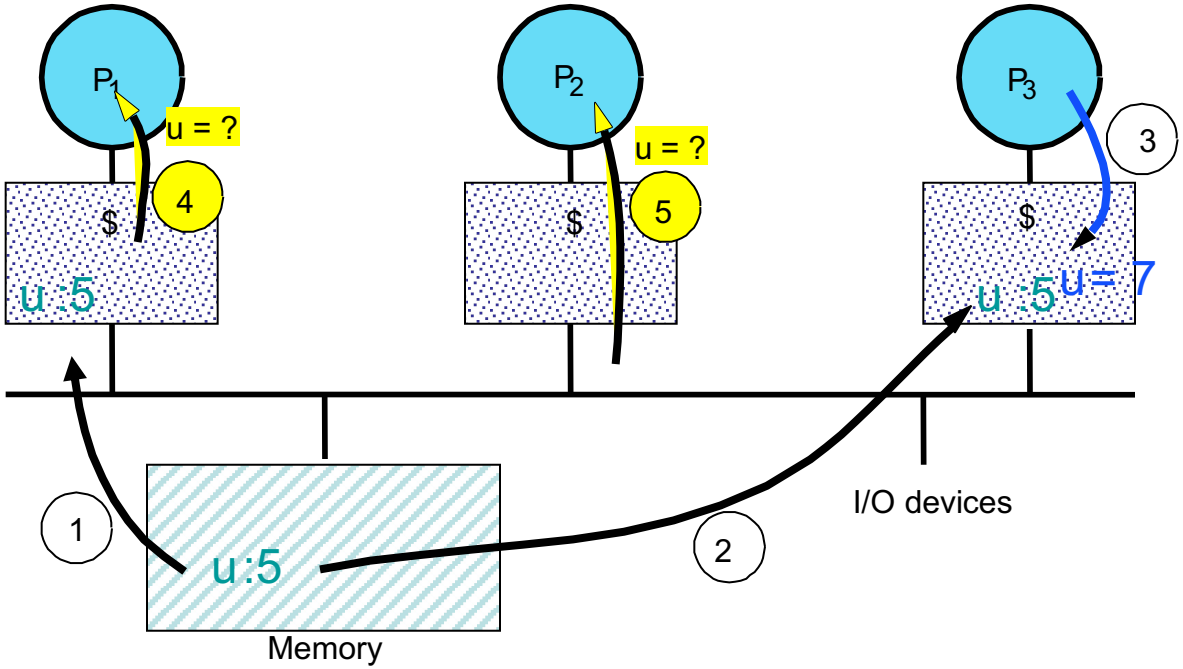
Implementation	Number of Threads			
	1	2	4	8
Read-Write Locks	2.48	4.97	4.69	4.71
One Mutex for Entire List	2.50	5.13	5.04	5.11
One Mutex per Node	12.00	29.60	17.00	12.00

- If we use the mutex approaches, the program is always as fast or faster when it's run with one thread
  - Effectively accesses to the list are serialized
- When the number of inserts and deletes is relatively large, the read-write lock program is also faster with one thread
  - When there are a substantial number of write-locks, there is too much contention for the locks
- In summary
  - The read-write lock implementation is superior to the one mutex and the one mutex per node implementations
  - However, unless the number of inserts and deletes is small, a serial implementation will be superior

Outline

- A Multi-threaded Linked List
  - Using One Mutex for Entire List
  - Using One Mutex per Node
  - Using Read-Write Lock
  - Performance of the Various Implementations
- Caches, Cache Coherence, and False Sharing
- Thread-Safety

Recall: Cache Coherence Problem



- Things to note:
  - Processors could see different values for u after event 3
  - With write back caches, value written back to memory depends on happenstance of which cache flushes or writes back value when
- How to fix with a bus: Coherence Protocol
  - Use bus to broadcast writes or invalidations
  - Simple protocols rely on presence of broadcast medium

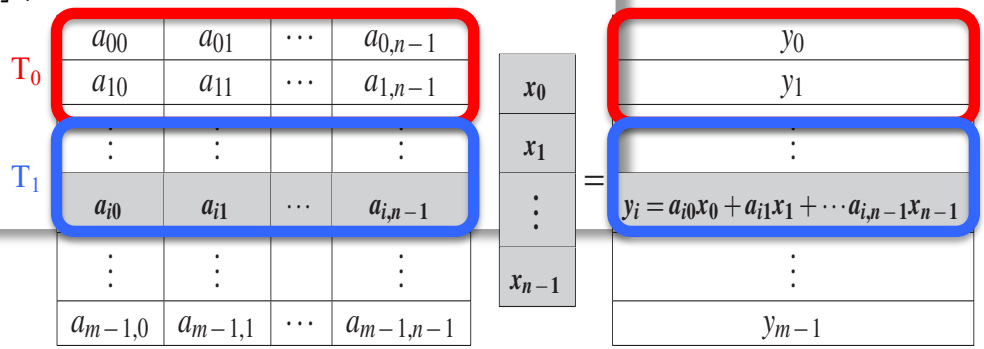
Recall: Matrix-Vector Multiplication

- The main thread initialized an *m***n* matrix *A* and an *n*-dimensional vector **x**

```
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*n+j]*x[j];
   }

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



Evaluation Platform

- Two dual-core processors
  - Each processor has its own cache
  - Cache-line level coherence
    - Each time any value in a cache line is written, if the line is also stored in another processor's cache, the entire line will be invalidated
- A cache line is 64 bytes
  - `y` is `double` type (8 bytes)
  - A single cache line can store 8 `double`s

Evaluation of Matrix-Vector Multiplication

- In each case, the total number of floating point additions and multiplications is 64,000,000

$$E = \frac{S}{t} = \frac{\left(\frac{T_{\text{serial}}}{T_{\text{parallel}}}\right)}{t} = \frac{T_{\text{serial}}}{t \times T_{\text{parallel}}}$$

Table 4.5 Run-Times and Efficiencies of Matrix-Vector Multiplication (times are in seconds)

	Matrix Dimension					
	$8{,}000{,}000 \times 8$		8000×8000		$8 \times 8{,}000{,}000$	
Threads	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

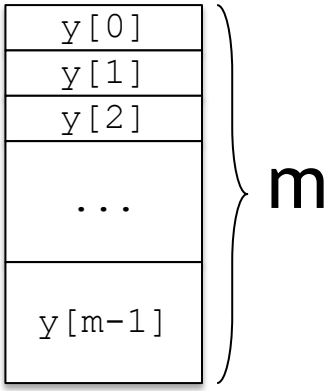
Slow Due to Cache Misses (Write-Misses)

Table 4.5 Run
Multiplication (

Threads	$8,000,000 \times 8$ Time	$8,000,000 \times 8$ Eff.	8000×8000 Time	8000×8000 Eff.	$8 \times 8,000,000$ Time	$8 \times 8,000,000$ Eff.
1	0.393	1.000	0.345	1.000	0.441	1.000
2			0.1	18	0.30	735
4			0.	50	0.38	290

m=8,000,000 m=8,000 m=8

Has far more cache write-misses than either of the other inputs




```
void *Pth_mat_vect(void* rank) {
   ...
   for (i = my_first_row; i <= my_last_row; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++)
         y[i] += A[i*n+j]*x[j];
   }
   return NULL;
}  /* Pth_mat_vect */
```

Slow Due to Cache Misses (Read-Misses)

Table 4.5 Run
Multiplication (

	$8,000,000 \times 8$		8000×8000		$8 \times 8,000,000$	
Threads	**Time**	**Eff.**	**Time**	**Eff.**	**Time**	**Eff.**
1	0.393	1.000	0.345	1.000	0.441	1.000
2	0.21	.906	0.	018		
4	0.1	.707	0.	50		

n=8

n=8,000

n=8,000,000

Has far more cache read-misses than either of the other inputs

A0,0	A0,1		A0,n-1
A1,0	A1,1		A1,n-1
A2,0	A2,1		A2,n-1
...	...	...	...
Am-1,0	Am-1,1		Am-1,n-1

n

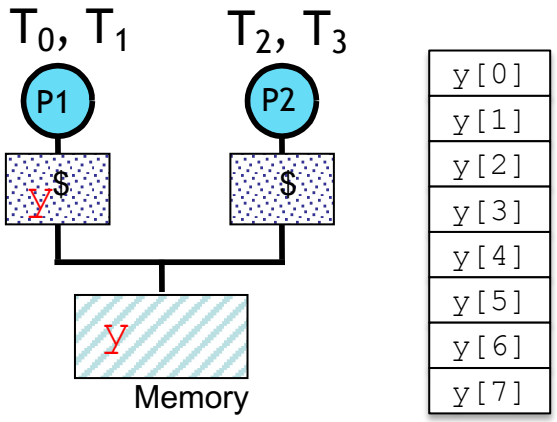
```
void *Pth_mat_vect(void* rank) {
   ...
   for (i = my_first_row; i <= my_last_row; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++)
          y[i] += A[i*n+j]*x[j];
   }
   return NULL;
}  /* Pth_mat_vect */
```

x0	x1	...	xn-1

n

Slow Due to Cache Coherence (1/2)

Threads	Matrix Dimension					
	$8{,}000{,}000 \times 8$		8000×8000		$8 \times 8{,}000{,}000$	
	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

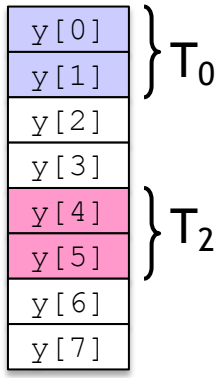


- With the $8 \times 8{,}000{,}000$ input, `y` has 8 components, so each thread is assigned 2 components
  - Suppose that threads 0 and 1 are assigned to one of the processors and threads 2 and 3 are assigned to the other
  - Also suppose that all of `y` is stored in a single cache line
- Every write to some element of `y` will invalidate the line in the other processor's cache

```
y[i] += A[i*n+j]*x[j];
```

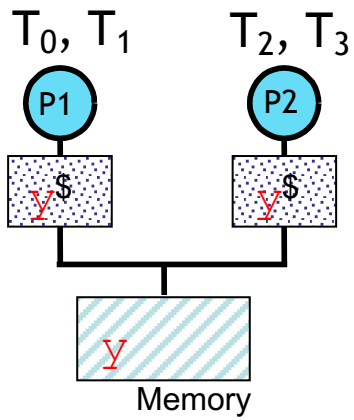
Slow Due to Cache Coherence (1/2)

- If thread 0 updates `y` and thread 2 is also accessing `y`, thread 2 have to reload `y`
  - Even though `y` in P2's cache holds the value that thread 2 needs
  - This scenario is called *false sharing*



- Each thread will update each of its components 8,000,000 times

```
void *Pth_mat_vect(void* rank) {
   ...
   for (i = my_first_row; i <= my_last_row; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++) // n = 8,000,000
         y[i] += A[i*n+j]*x[j];
   }
   return NULL;
}  /* Pth_mat_vect */
```



False Sharing

- Suppose two threads with separate caches access different variables that belong to the same cache line
- Further suppose at least one of the threads updates its variable
- Then even though neither thread has written to a variable that the other thread is using, the cache controller invalidates the entire cache line and forces the threads to get the values of the variables from main memory
- The threads aren't sharing anything (except a cache line), but the behavior of the threads with respect to memory access is the same as if they were sharing a variable, hence the name *false sharing*

Slow Due to Other Factors

- There may be other factors that are affecting the relative performance of the single-threaded program with the differing inputs
  - E.g., virtual memory
    - How frequently does the CPU need to access the page table in main memory?

Outline

- A Multi-threaded Linked List
  - Using One Mutex for Entire List
  - Using One Mutex per Node
  - Using Read-Write Lock
  - Performance of the Various Implementations
- Caches, Cache Coherence, and False Sharing
- Thread-Safety

Thread-Safety

- Let's look at another potential problem that occurs in shared-memory programming: *thread-safety*
- A block of code is **thread-safe** if it can be simultaneously executed by multiple threads without causing problems

Example: Multithreaded Tokenizers

- As an example, suppose we want to use multiple threads to "tokenize" a file
- A simple approach to this problem is to divide the input file into lines of text
- One way to do this is to use the `strtok` function in `string.h`

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

Usage of `strtok`

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

- Unusual usage:
  - The first time it's called the string argument should be the text to be tokenized
  - For subsequent calls, the first argument should be `NULL`
- In the first call, `strtok` caches a pointer to `string`, and for subsequent calls it returns successive tokens taken from the cached copy

A Multithreaded Tokenizer (1/2)

```
int thread_count;
sem_t* sems; // one semaphore for each thread

int main(int argc, char* argv[]) {
   long       thread;
   pthread_t* thread_handles;

   thread_count = atoi(argv[1]);
   thread_handles = (pthread_t*) malloc (thread_count*sizeof(pthread_t));
   sems = (sem_t*) malloc(thread_count*sizeof(sem_t));
   // sems[0] should be unlocked, the others should be locked
   sem_init(&sems[0], 0, 1);
   for (thread = 1; thread < thread_count; thread++)
      sem_init(&sems[thread], 0, 0);

   for (thread = 0; thread < thread_count; thread++)
      pthread_create(&thread_handles[thread], (pthread_attr_t*) NULL,
          Tokenize, (void*) thread);

   for (thread = 0; thread < thread_count; thread++)
      pthread_join(thread_handles[thread], NULL);

   for (thread=0; thread < thread_count; thread++)
      sem_destroy(&sems[thread]);

   free(sems);
   free(thread_handles);
   return 0;
}  /* main */
```

A Multithreaded Tokenizer (2/2)

```
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;

   /* Force sequential reading of the input */
   sem_wait(&sems[my_rank]);
   fg_rv = fgets(my_line, MAX, stdin); // read a line of input
   sem_post(&sems[next]); // allows next thread to read a line
   while (fg_rv != NULL) {
      printf("Thread %ld > my line = %s", my_rank, my_line);

      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 */
```

The thread reads a line of input for the first time

Identify tokens

The thread reads a line of input for any additional input

Running the Multithreaded Tokenizer

- Input:

Pease porridge hot.
Pease porridge cold.
Pease porridge in the pot
Nine days old.

- Output:

```
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.
```

What Happened to the Multithreaded Tokenizer?

- Recall that `strtok` caches the input line
  - Done by declaring a variable to have `static` storage class
- Unfortunately for us, this cached string is shared, not private. Thus, thread 0's call to `strtok` with the third line of the input has apparently overwritten the contents of thread 1's call with the second line
- The `strtok` function is *not* thread-safe
  - `strtok_r`: a thread-safe version of `strtok`

Incorrect Programs Can Produce Correct Output

- In the example of a multithreaded tokenizer, the program might produce correct output. It wasn't until a later run that we saw an error
  - Not a rare occurrence in parallel programs
  - Especially common in shared-memory programs
  - Due to nondeterministic execution of threads

References

- Peter Pacheco, An Introduction to Parallel Programming, Morgan Kaufmann; 1 edition (January 21, 2011)
- LLNL Pthread Tutorial: http://www.llnl.gov/computing/tutorials/pthreads/