High-Performance Computing

– Shared Memory Programming with Pthreads –

Christian Plessl

High-Performance IT Systems Group

Paderborn University

version 1.1.2 2017-11-24

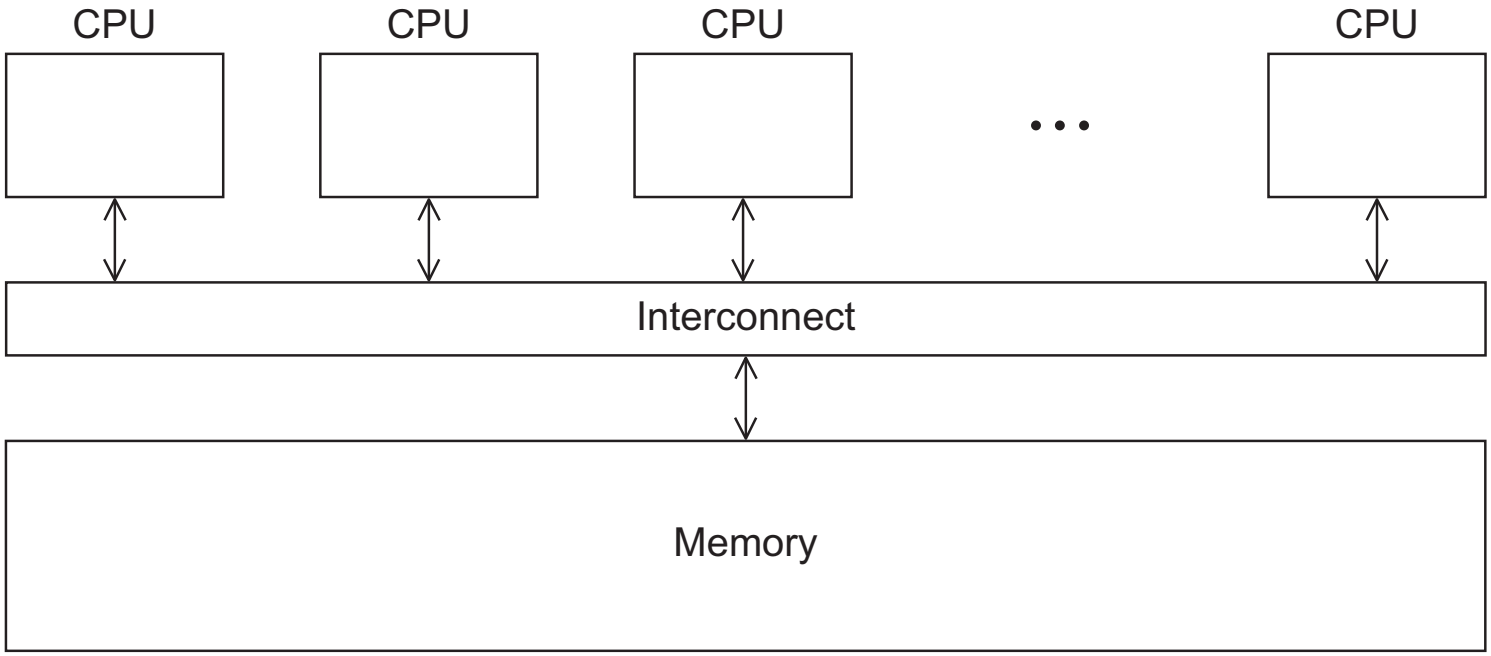
UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Paderborn Center for Parallel Computing

Outline

- **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



Processes and Threads

- **A process is an instance of a running (or suspended) program**
- **Threads are analogous to "light-weight" processes**
- **In a shared memory program a single process may have multiple threads of control**
- **Threads are not restricted to HPC**
  - can be used for parallel processing (share computational work, performance is key)
  - can be used for concurrent processing (e.g. producer/consumer, background processing, interactive user interfaces, etc.)

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**
- **Availability**
  - widely available in Unix-derived systems: Linux, macOS, BSD, Solaris
  - not available on Windows, but compatibility libraries exist
- **Pthreads are general-purpose basic building blocks for multi-threaded applications**
  - low level of abstraction, but explicit control over threads
  - in future lectures we will also look at higher-level approaches that use multi-threading under the hood (e.g. OpenMP tasking)

Hello World (1)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4
5  /* Global variable:  accessible to all threads */
6  int thread_count;
7
8  void* Hello(void* rank);  /* Thread function */
9
10 int main(int argc, char* argv[]) {
11    long       thread;  /* Use long in case of a 64-bit system */
12    pthread_t* thread_handles;
13
14    /* Get number of threads from command line */
15    thread_count = strtol(argv[1], NULL, 10);
16
17    thread_handles = malloc (thread_count*sizeof(pthread_t));
```

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

allocate memory in master thread to store handles to child threads

Hello World (2)

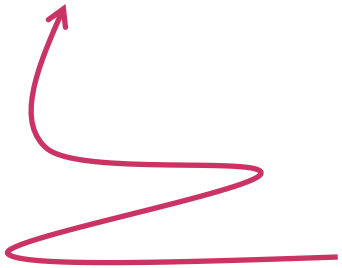
```
18
19      for (thread = 0; thread < thread_count; thread++)
20         pthread_create(&thread_handles[thread], NULL,
21             Hello, (void*) thread);
22
23      printf("Hello from the main thread\n");
24
25      for (thread = 0; thread < thread_count; thread++)
26         pthread_join(thread_handles[thread], NULL);
27
28      free(thread_handles);
29      return 0;
30   }  /* main */
31
32   void* Hello(void* rank) {
33      long my_rank = (long) rank
              /* Use long in case of 64-bit system */
34
35      printf("Hello from thread %ld of %d\n", my_rank,
              thread_count);
36
37      return NULL;
38   }  /* Hello */
```

spawn threads in master

join threads

Compiling a Pthread program

```
gcc –g –Wall –pthread pth_hello pth_hello.c
```



enable Pthread macros and link with 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
```

Global Variables

- **Threads share the same memory space, i.e. all variables of the master are accessible (read and write) in the spawned threads**
- **Use of global variables (in particular inadvertent use) can introduce subtle and confusing bugs because of race conditions**
  - limit use of global variables to situations in which they are really needed (shared data)
  - use local variables wherever possible

Starting the Threads

- Processes in MPI are usually started by a launcher program (e.g. mpirun)
- In Pthreads the threads are explicitly started by the program executable

pthread_t: one object (handle) for each thread for referencing thread after creation

```
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

- Implemented as opaque data structure
- 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 is associated

pthread_create in Detail (1)

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

1 **returns handle to thread, memory must be allocate before calling**

2 **attributes to control configurable thread attributes (stack size, scheduling policy, etc.) Typically the default behavior is OK, can pass in NULL for this case.**

pthread_create in Detail (2)

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

3 pointer to the argument(s) that are passed to the thread function `start_routine`

4 pointer to function that is executed by the thread

Function Started by pthread_create

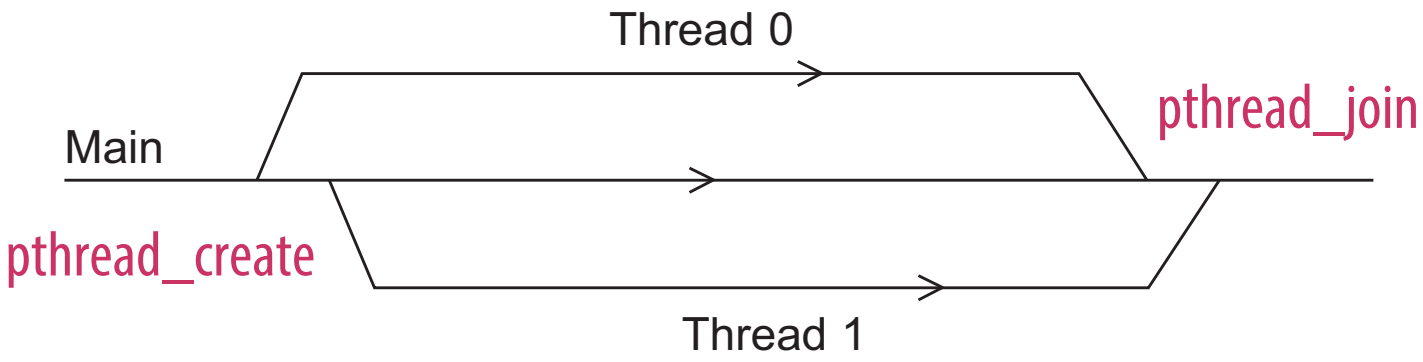
- **Function to be executed is passed as a function pointer to a function with void arguments and void return value**
- **Prototype**

```
void*  thread_function ( void*  args_p ) ;
```

- **Argument void* can be cast to any pointer type in C**
  - e.g. if the `thread_function` needs more than one argument `args_p` can also point to a list containing one or more values
- **Return value void***
  - can point to a list of one or more values.

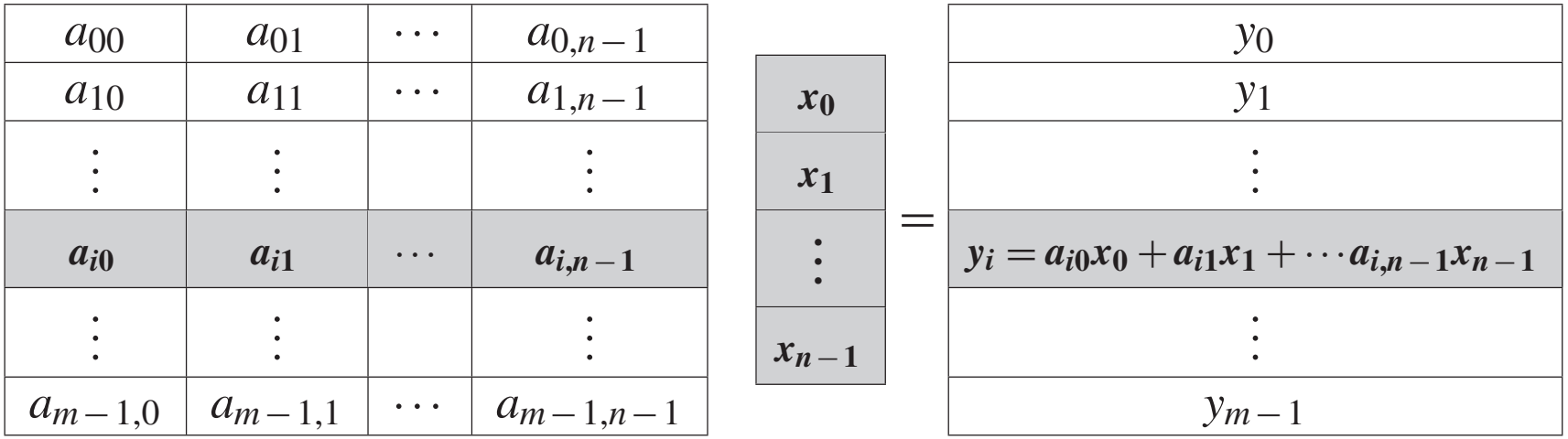
Running and Joining Threads

- **Lifecycle of threads**



- **Joining a thread**
  - call pthread_join once for each thread (identify thread with pthread_t handle)
  - blocks until the associated thread terminates

Use Case: Matrix-Vector Multiplication w/ Pthreads



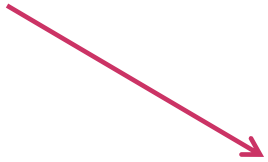
Serial Pseudo-Code

```
/* For each row of A */
for (i = 0; i < m; i++) {
    y[i] = 0.0;
    /* For each element of the row and each element of x */
    for (j = 0; j < n; j++)
        y[i] += A[i][j]* x[j];
}
```

$$y_i = \sum_{j=0}^{n-1} a_{ij} x_j.$$

Using 3 Threads

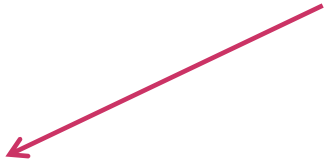
Thread	Components of y
0	y[0], y[1]
1	y[2], y[3]
2	y[4], y[5]



thread 0

```
y[0] = 0.0;
for (j = 0; j < n; j++)
   y[0] += A[0][j]* x[j];
```

general case



```
y[i] = 0.0;
for (j = 0; j < n; j++)
   y[i] += A[i][j]*x[j];
```

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 */
```

rank passed as argument to pthread_create

m: number of rows

Critical Sections

Estimating π

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

series expansion

```
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;
```

serial code

Multithreaded Computation of π

- **simple parallelization strategy**
  - each thread computes n/thread_count terms
  - make sum a shared variable

```
1  void* Thread_sum(void* rank) {
2     long my_rank = (long) rank;
3     double factor;
4     long long i;
5     long long my_n = n/thread_count;
6     long long my_first_i = my_n*my_rank;
7     long long my_last_i = my_first_i + my_n;
8
9     if (my_first_i % 2 == 0)  /* my_first_i is even */
10       factor = 1.0;
11    else  /* my_first_i is odd */
12       factor = -1.0;
13
14    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
15       sum += factor/(2*i+1);
16    }
17
18    return NULL;
19 }  /* Thread_sum */
```

code for each thread

Results from Multi-Threaded Execution

	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

- **Different results for single and multi-threaded execution**
  - with increasing n the estimate of the single-threaded code is getting better and better
- **Race condition**
  - both threads write to global variable sum

Prevent Data Races with 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!
- Drawback: wastes computing resources because a waiting thread continually uses the CPU accomplishing nothing

```
1   y = Compute(my_rank);
2   while (flag != my_rank);
3   x = x + y;
4   flag++;
```

flag initialized to 0 by main thread

compact but somehow obscure, better write loop like this

```
while (flag != my_rank) {
   // do nothing
}
```

Pthreads Global Sum with Busy-Waiting

```
1   void* Thread_sum(void* rank) {
2      long my_rank = (long) rank;
3      double factor;
4      long long i;
5      long long my_n = n/thread_count;
6      long long my_first_i = my_n*my_rank;
7      long long my_last_i = my_first_i + my_n;
8
9      if (my_first_i % 2 == 0)
10        factor = 1.0;
11     else
12        factor = -1.0;
13
14     for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
15        while (flag != my_rank);
16        sum += factor/(2*i+1);
17        flag = (flag+1) % thread_count;
18     }
19
20     return NULL;
21  }  /* Thread_sum */
```

high overhead, because busy-waiting in every loop iteration

Optimization: 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 */
```

1) perform local sum reduction on local variable first

2) perform global sum reduction on local variable first

Mutexes

- **Mutex (mutual exclusion) special type of variable for restricting access to a critical section to a single thread at a time**
- **Guarantees that one thread "excludes" all other threads while it executes the critical section**
- **No waste of computing resources, in contrast to busy-waiting**
  - excluded threads wait in blocking queue
- **The Pthreads standard includes a special (opaque) type `pthread_mutex_t` for mutexes**
- **Manipulation of mutexes with dedicated function**

Mutex Handling in the Pthreads library

- **A mutex needs to be created and initialized before the first use**

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

- **To enter a critical section a thread calls**

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

- **To leave a critical section a thread calls**

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

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

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

Global Sum Function using a Mutex

assumption: main thread has created and initialized a global variable for the mutex

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex,NULL);
```

```
1  void* Thread_sum(void* rank) {
2     long my_rank = (long) rank;
3     double factor;
4     long long i;
5     long long my_n = n/thread_count;
6     long long my_first_i = my_n*my_rank;
7     long long my_last_i = my_first_i + my_n;
8     double my_sum = 0.0;
9
10    if (my_first_i % 2 == 0)
11       factor = 1.0;
12    else
13       factor = -1.0;
14
15    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
16       my_sum += factor/(2*i+1);
17    }
18    pthread_mutex_lock(&mutex);
19    sum += my_sum;
20    pthread_mutex_unlock(&mutex);
21
22    return NULL;
23 }  /* Thread_sum */
```

Global Sum: Busy-Wait vs. Mutex

Table 4.1 Run-Times (in Seconds) of π Programs Using $n = 10^8$ Terms on a System with Two Four-Core Processors

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

critical section only entered once, hardly any difference between busy-wait and mutex if threads <= cores

significant differences if threads > cores **why?**

Run-times (in seconds) of π programs using n = 108 terms on a system with two four-core processors.

Global Sum: Busy-Wait vs. Mutex (2)

- Possible sequence of events with busy-waiting and more threads than cores
- Busy-waiting enforces a fixed order of tasks entering the critical section
- Thread scheduling may cause delays

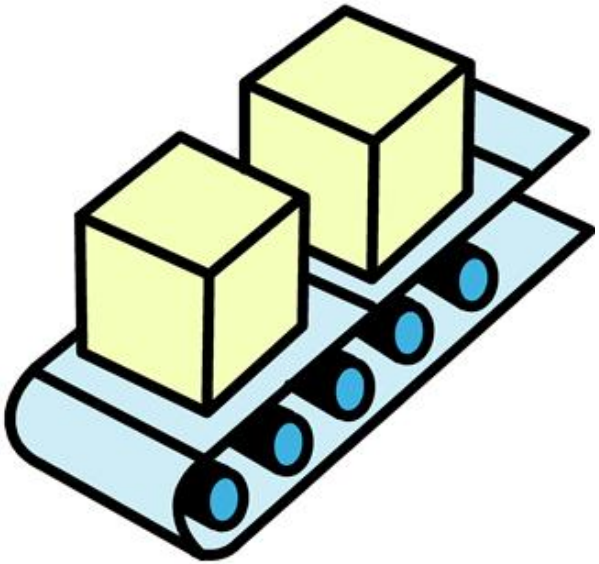
Table 4.2 Possible Sequence of Events with Busy-Waiting and More Threads than Cores

		Thread				
Time	`flag`	**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

no task makes any progress

scheduler runs task 2 again

Semaphores for Producer-Consumer Synchronization



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**
  - for example, reductions where the order of operations must not be changed (e.g. floating-point operations)

Attempt: Synchronize Messages Passing with Pthreads

- Each thread should receive exactly one message
- Problem: the more threads are used (#threads > cores) the higher the chance that the message is still uninitialized because the thread has not been scheduled yet

```
1  /* messages has type char**. It's allocated in main. */
2  /* Each entry is set to NULL in main.                */
3  void* Send_msg(void* rank) {
4     long my_rank = (long) rank;
5     long dest = (my_rank + 1) % thread_count;
6     long source = (my_rank + thread_count - 1) % thread_count;
7     char* my_msg = malloc(MSG_MAX*sizeof(char));
8
9     sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
10    messages[dest] = my_msg;
11
12    if (messages[my_rank] != NULL)
13       printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
14    else
15       printf("Thread %ld > No message from %ld\n", my_rank,
                source);
16
17    return NULL;
18 }  /* Send_msg */
```

Attempt: Synchronize Messages Passing with Pthreads (2)

- **Busy-waiting solves problem, but enforces order and wastes computing time**

```
1   /* messages has type char**. It's allocated in main. */
2   /* Each entry is set to NULL in main.                */
3   void* Send_msg(void* rank) {
4      long my_rank = (long) rank;
5      long dest = (my_rank + 1) % thread_count;
6      long source = (my_rank + thread_count - 1) % thread_count;
7      char* my_msg = malloc(MSG_MAX*sizeof(char));
8
9      sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
10     messages[dest] = my_msg;
11
12     if (messages[my_rank] != NULL)   while(messages[my_rank] == NULL) {};
13        printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
14     else
15        printf("Thread %ld > No message from %ld\n", my_rank,
                source);
16
17     return NULL;
18  }  /* Send_msg */
```

- **What we actually would like to achieve is informing thread dest that a message is available after executing line 10**

Syntax of Semaphore Manipulation Functions

semaphores are port of POSIX but not part of Pthreads library

```
#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 */);
```

Synchronize Message-Passing w/ Semaphores

```
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 */
```

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**
- **Many Pthreads implementations do not contain ready to use barriers**
- **Hence, we have to build barriers from other mechanisms**
  - busy-waiting and mutex
  - semaphores
  - condition variables

Use-Cases for Barriers

- **Measure execution time of 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;
```

- **Debugging**

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

Barrier with Busy-waiting and 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

Barrier with Busy-waiting and Mutex (2)

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

counter counts how many threads have reached the barrier

- **Problem 1: busy-waiting wastes CPU resources**
- **Problem 2: reusing barrier safely is not possible**
  - resetting counter in master thread may lead to the situation that not all threads have seen counter == thread_count and are stuck in the waiting loop
  - hence, we need one counter for each barrier instance

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);
   }
   . . .
}
```

count_sem initialized to unlocked, first thread calling sem_wait on this semaphore is not blocked

last thread reaches the barrier, notify all waiting threads

there are still threads that need to reach the barrier, block until then

- **Much more efficient than busy-waiting**
- **Problem: Reusing the barrier safely is still not possible, race condition for barrier_sem (see Pacheco Chapter 4)**

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
- Typically use in code like this

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

Condition Variable Handling in the Pthreads library

- **Condition variables have type `pthread_cond_t`**
- **The function pthread_cond_wait**

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

atomically blocks the current thread on the condition variable `cond_var_p` and releases the mutex specified by `mutex_p`. The waiting thread unblocks only after another thread calls `pthread_cond_signal` or `pthread_cond_broadcast` for the same condition variable and the thread reacquires the lock on `mutex_p` again

- **Essentially pthead_cond_wait performs these functions atomically:**

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

Condition Variable Handling in the Pthreads library (2)

- **Condition variables are initialized and destroyed using the functions**

```
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 */);
```

- **One of the threads waiting for the condition variable can be unblocked with**

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

- **All of the threads waiting for the conditional variable can be unblocked with**

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

Implementing a Barrier with Condition Variables

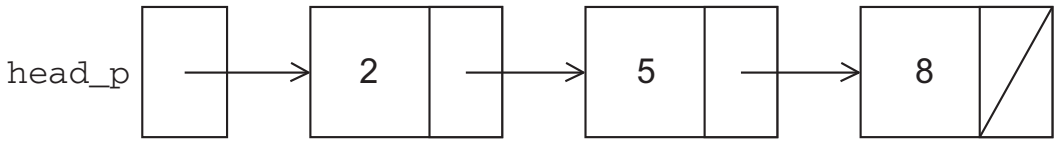
```
/* Shared */
int counter = 0;
pthread_mutex_t mutex;
pthread_cond_t cond_var;
. . .
void* Thread_work(. . .) {
   . . .
   /* Barrier */
   pthread_mutex_lock(&mutex);
   counter++;
   if (counter == thread_count) {
      counter = 0;
      pthread_cond_broadcast(&cond_var);
   } else {
      while (pthread_cond_wait(&cond_var, &mutex) != 0);
   }
   pthread_mutex_unlock(&mutex);
   . . .
}
```

pthread_cond_wait is called in a loop checking for success, because there are other events that may cause the thread to unblock (e.g. cancel)

Read-Write Locks

Controlling access shared data structures

- **How can we control concurrent access to large shared data structures?**
- **Example**
  - shared data structure is a sorted linked list of integers
  - operations of interest are Member, Insert, and Delete

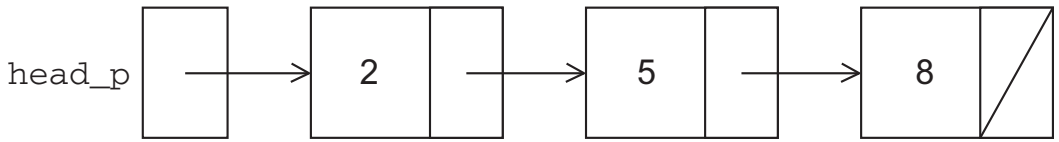



```
struct list_node_s {
   int data;
   struct list_node_s* next;
}
```

Linked List: Member

- **Check whether element is part of the list**
  - traverse list from beginning until the end is reached or a larger element is found

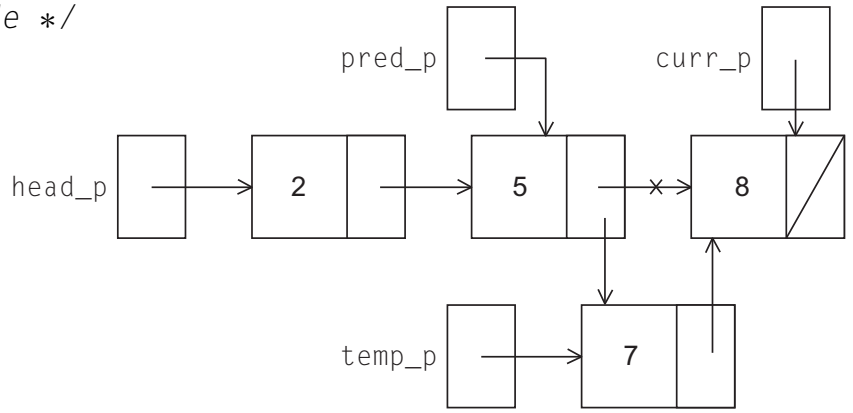
```
1   int  Member(int value, struct list_node_s* head_p) {
2      struct list_node_s* curr_p = head_p;
3
4      while (curr_p != NULL && curr_p->data < value)
5         curr_p = curr_p->next;
6
7      if (curr_p == NULL || curr_p->data > value) {
8         return 0;
9      } else {
10        return 1;
11     }
12  }  /* Member */
```



Linked List: Insert

- **Insert a new element into the sorted linked list**

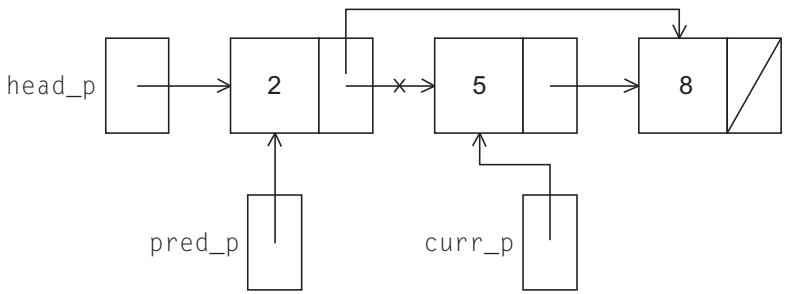
```
1  int Insert(int value, struct list_node_s** head_p) {
2     struct list_node_s* curr_p = *head_p;
3     struct list_node_s* pred_p = NULL;
4     struct list_node_s* temp_p;
5
6     while (curr_p != NULL && curr_p->data < value) {
7        pred_p = curr_p;
8        curr_p = curr_p->next;
9     }
10
11    if (curr_p == NULL || curr_p->data > value) {
12       temp_p = malloc(sizeof(struct list_node_s));
13       temp_p->data = value;
14       temp_p->next = curr_p;
15       if (pred_p == NULL)  /* New first node */
16          *head_p = temp_p;
17       else
18          pred_p->next = temp_p;
19       return 1;
20    } else { /* Value already in list */
21       return 0;
22    }
23 }  /* Insert */
```



Linked List: Delete

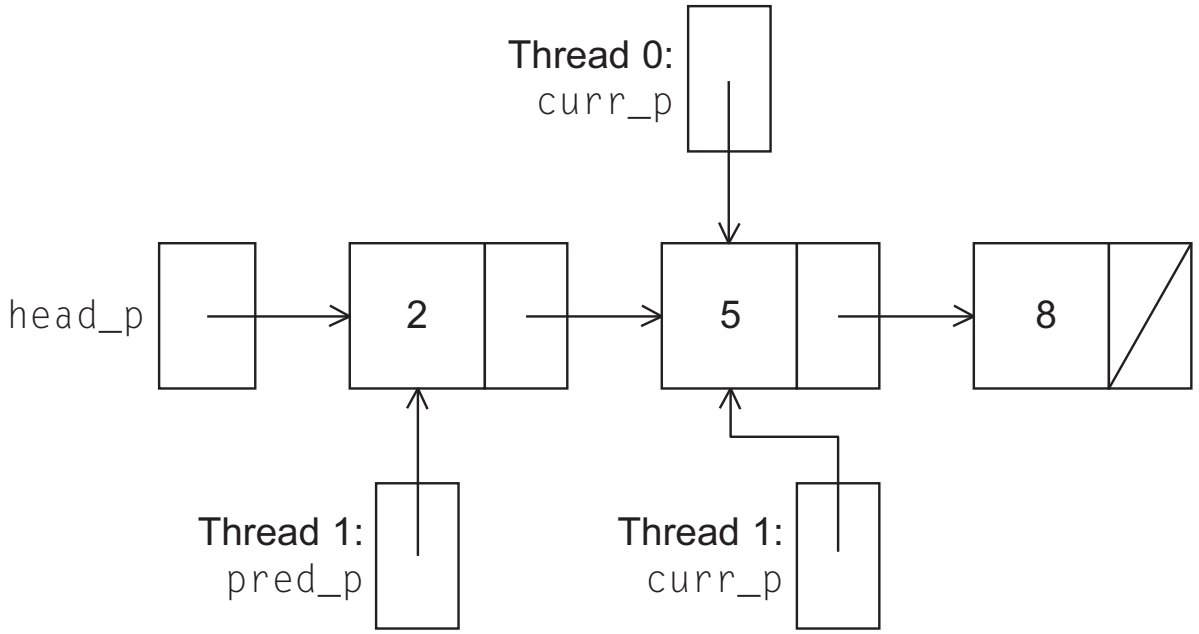
- **Delete an existing element from the list**

```
1  int Delete(int value, struct list_node_s** head_p) {
2     struct list_node_s* curr_p = *head_p;
3     struct list_node_s* pred_p = NULL;
4
5     while (curr_p != NULL && curr_p->data < value) {
6        pred_p = curr_p;
7        curr_p = curr_p->next;
8     }
9
10    if (curr_p != NULL && curr_p->data == value) {
11       if (pred_p == NULL) { /* Deleting first node in list */
12          *head_p = curr_p->next;
13          free(curr_p);
14       } else {
15          pred_p->next = curr_p->next;
16          free(curr_p);
17       }
18       return 1;
19    } else {  /* Value isn't in list */
20       return 0;
21    }
22 }  /* Delete */
```



A Multi-Threaded Linked List

- **Let's try to use these functions in a Pthreads program**
  - to share access to the list, we define head_p to be a global variable
  - this simplifies the Member, Insert, and Delete functions, because no pointer to head_p is needed, only the value of interest
- **What happens if the Member, Insert, Delete functions are called simultaneously from different threads?**



Solution #1: Coarse-Grained Locking

- **Obvious solution**
  - simply lock the list any time that a thread attempts to access it
  - protect call to the Member, Insert, Delete function by mutex

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

Instead of calling
`Member(value)`

- **Problems**
  - access to list is serialized
  - fail to exploit opportunity for parallelism for read-only operations (Member), hurts performance if this is the frequent case
  - if most of our operations are calls to Insert and Delete, then this may be the best solution because we need to serialize access for these operations and the solution is simple to implement

Solution #2: Fine-Grained Locking

- **Lock individual elements instead of entire list**
- **Requires modification to list elements, each element is protected by mutex**

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

Member function w/ 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 */
```

idea: always hold mutex to currently used elements, lots of special case handling (beginning/end of list)

Problems with Fine-Grained Locking

- Implementation much more complex and error prone than the original Member function
- Much slower, because for each access to a node a mutex must be locked and unlocked
- The addition mutex field for each node substantially increases 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**
  - coarse-grained locking only allows one thread to access the entire list at any instant
  - fine-grained locking only allows one thread to access any given node at any instant
- **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
  - the second locks it for writing
- **Usage**
  - multiple threads can simultaneously obtain the read
  - only one thread can obtain the write lock
  - 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

Pthreads Read-Write Lock Functions

- **The following functions are available for acquiring a read lock, acquiring a read/write lock, and releasing a lock:**

```
int pthread_rwlock_rdlock(pthread_rwlock_t*  rwlock_p /* in/out */);
int pthread_rwlock_wrlock(pthread_rwlock_t*  rwlock_p /* in/out */);
int pthread_rwlock_unlock(pthread_rwlock_t*  rwlock_p /* in/out */);
```

- **Like mutexes, read/write locks need to be initialized and destroyed:**

```
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 */);
```

Protecting the Linked-List Functions w/ Locks

```
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

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

Caches, Cache-Coherence, and False Sharing

- **Cache memory can have a huge impact on shared-memory**
  - significant performance difference between cache access and main memory access
  - cache coherency protocols ensure correct cache access for multi-threaded applications
  - threads influence each other indirectly through cache memory (eviction, dirty cache lines, etc.)

```
1   void *Pth_mat_vect(void* rank) {
2      long my_rank = (long) rank;
3      int i, j;
4      int local_m = m/thread_count;
5      int my_first_row = my_rank*local_m;
6      int my_last_row = (my_rank+1)*local_m - 1;
7
8      for (i = my_first_row; i <= my_last_row; i++) {
9         y[i] = 0.0;
10        for (j = 0; j < n; j++)
11           y[i] += A[i][j]*x[j];
12     }
13
14     return NULL;
15  }  /* Pth_mat_vect */
```

shared variables

Pthreads matrix-vector multiplication

Pthreads Matrix-Vector Multiplication

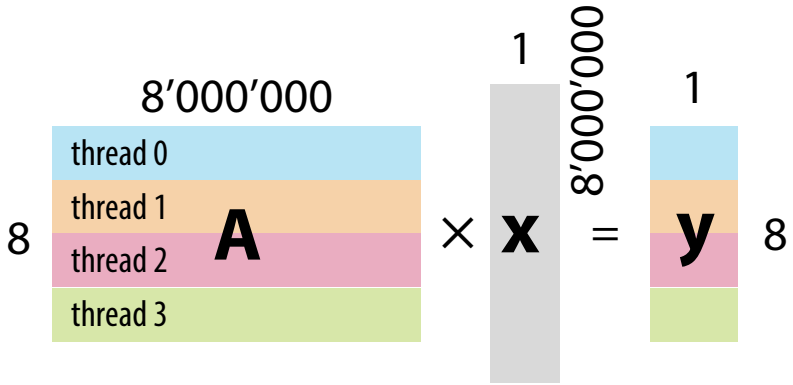
- **Number of operations in matrix-vector multiplication**
  - matrix dimensions: m × n, vector dimension: p
  - number of multiplications and additions ≈ m*n*p
- **Experiment**
  - multi-threaded matrix-vector multiplication with a constant number of operations (m*n*p)
  - different "aspect ratios" of matrix and vector
  - efficiency for increasing number of threads varies widely with aspect ratio
- **What is going on?**

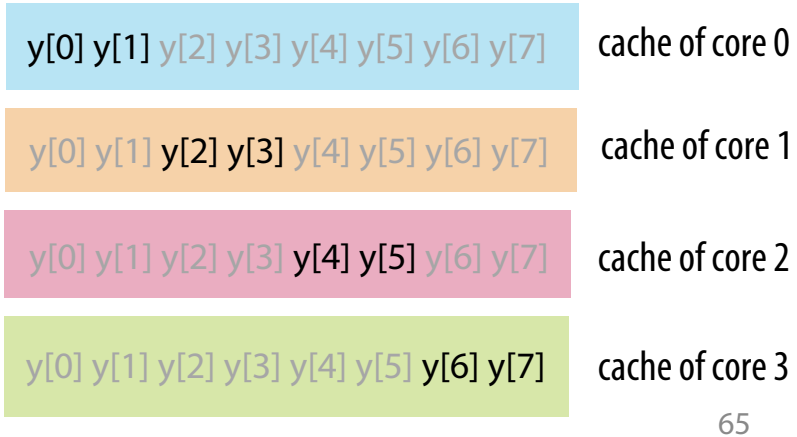
	Matrix Dimension					
	$8,000,000 \times 8$		8000×8000		$8 \times 8,000,000$	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0.393	1.000	0.345	1.000	0.441	1.000
2	0.217	0.906	0.188	0.918	0.300	0.735
4	0.139	0.707	0.115	0.750	0.388	0.290

Pthreads Matrix-Vector Multiplication (2)

- **Assumptions**
  - x, y, A store values of type double
  - cache line stores 8 doubles
- **Each thread computes a part of y[]**
  - result vector y[] fits into a single cache line
  - each thread has its own copy of y[]
- **Cache coherency**
  - cache coherency protocols work on the granularity of a a cache line
  - whenever a thread writes to y[] the cached copy of y[] stored by other threads must be invalidated
  - but y[] is not actually shared, each thread always writes to a distinct part of y[]
  - this effect is called false sharing and causes unnecessary overheads

```
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];
}
```





Thread-Safety

Thread-Safety

- **A block of code is thread-safe if it can be simultaneously executed by multiple threads without causing problems**
- **Example**
  - we want to "tokenize" a file of text using multiple threads
  - tokens are contiguous sequences of characters separated from the rest of the text by white-space like spaces, tabs, or newlines
- **Simple approach**
  - divide input file into lines of text, assign lines to threads in a round-robin fashion
  - first line goes to thread 0, the second goes to thread 1, . . . , the t-th goes to thread t, the t +1st goes to thread 0, etc
  - serialize access to the file using semaphores
  - after a thread has read a single line of input, it can tokenize the line using the strtok function

The strtok Function

- **Function is called repeatedly until the the complete string is tokenized**
  - at the first call, the string argument must be the text to be tokenized
  - for subsequent calls, the first argument should be NULL
  - the second argument is a string containing all separators e.g. "\n\r\t "

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

- **The idea is that strtok does the bookkeeping**
  - keeps track of the progress of tokenization
  - works from a cached copy of the string

Multi-Threaded Tokenizer (Incorrect)

```
1  void* Tokenize(void* rank) {
2     long my_rank = (long) rank;
3     int count;
4     int next = (my_rank + 1) % thread_count;
5     char *fg_rv;
6     char my_line[MAX];
7     char *my_string;
8
9     sem_wait(&sems[my_rank]);
10    fg_rv = fgets(my_line, MAX, stdin);
11    sem_post(&sems[next]);
12    while (fg_rv != NULL) {
13       printf("Thread %ld > my line = %s", my_rank, my_line);
14
15       count = 0;
16       my_string = strtok(my_line, " \t\n");
17       while ( my_string != NULL ) {
18          count++;
19          printf("Thread %ld > string %d = %s\n", my_rank, count,
20                my_string);
21          my_string = strtok(NULL, " \t\n");
22       }
23
24       sem_wait(&sems[my_rank]);
25       fg_rv = fgets(my_line, MAX, stdin);
26       sem_post(&sems[next]);
27    }
28
29    return NULL;
30 }  /* Tokenize */
```

local copy of line

prevent concurrent file access with semaphores

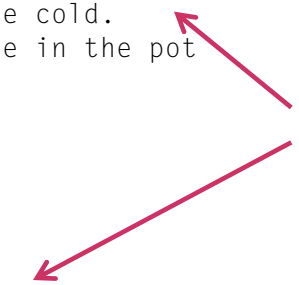
tokenization

Looks simple. What could possibly go wrong?

Running with Two Threads

- Running single threaded works perfectly
- Running with two threads shows a bug

```
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: Something unexpected happened!

What Happened?

- **strtok caches the input line**
  - string is stored in a variable with static storage class (static keyword in C)
  - this causes the value stored in this variable to persist from one call to the next
- **Unfortunately for us, this cached string is shared, not private**

```
char * strtok(char * string, const char * separators) {
  static char *string_cache;
  if (string != NULL) {
    string_cache = strdup(string);
  }
  ...
  return current_token;
}
```

pseudo code for strtok

  - 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
- **We denote functions like strtok as not thread-safe or not re-entrant**
  - if multiple threads call a non thread-safe function the result is undetermined

Other C Library Functions

- **Many C library functions predate multi-threading**
  - in particular the old, standard library functions
  - regrettable it is thus not uncommon for C library functions to be not thread-safe
- **Examples of common not thread-safe functions**
  - random number generator random in stdlib.h
  - time conversion function localtime in time.h
- **Warning**
  - be careful to check that all library functions used in multi-threaded code is thread-safe
  - ignoring thread-safety can lead to terribly hard to find bugs because code may execute correctly most of the time

“re-entrant” (Thread Safe) Functions

- **In some cases, the C standard specifies an alternate, thread-safe, version of a function**
  - typically with same name and a suffix like “_r”
  - e.g. the standard library defines a re-entrant strtok function name strtok_r

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

  - the third argument is a pointer to keep track of the state of the function (in this case the cached string that was previously stored as a static variable)

Multi-Threaded Tokenizer (Corrected)

```
1  void* Tokenize(void* rank) {
2     long my_rank = (long) rank;
3     int count;
4     int next = (my_rank + 1) % thread_count;
5     char *fg_rv;
6     char my_line[MAX];
7     char *my_string;
8
9     sem_wait(&sems[my_rank]);
10    fg_rv = fgets(my_line, MAX, stdin);
11    sem_post(&sems[next]);
12    while (fg_rv != NULL) {
13       printf("Thread %ld > my line = %s", my_rank, my_line);
14       char *saveptr;
15       count = 0;
16       my_string = strtok(my_line, " \t\n");   my_string = strtok_r(my_line, " \t\n ", &saveptr);
17       while ( my_string != NULL ) {
18          count++;
19          printf("Thread %ld > string %d = %s\n", my_rank, count,
20                my_string);
21          my_string = strtok(NULL, " \t\n");   my_string = strtok_r(NULL, " \t\n ", &saveptr);
22       }
23
24       sem_wait(&sems[my_rank]);
25       fg_rv = fgets(my_line, MAX, stdin);
26       sem_post(&sems[next]);
27    }
28
29    return NULL;
30 }  /* Tokenize */
```

Concluding Remarks (1)

- **A thread in shared-memory programming is analogous to a process in distributed memory programming**
  - However, a thread is lighter-weight than a full-fledged process
  - in Pthreads programs, all the threads have access to global variables, while local variables are private to the thread running the function
- **A race condition denotes the possibility of an errors by non-deterministic behavior resulting from multiple threads attempting to access a shared resource (e.g. shared variable or file) concurrently**
- **A critical section is a block of code that protects a shared resource such that it can only be updated by one thread at a time**
  - the execution of code in a critical section should, effectively, be executed as serial code

Concluding Remarks (2)

- **Busy-waiting can be used to avoid conflicting access to critical sections with a flag variable and a while-loop with an empty body**
  - can be very wasteful of CPU cycles
  - can also be unreliable if compiler optimization is turned on
- **A mutex can be used to avoid conflicting access to critical sections as well**
  - a mutex is a lock on a critical section that ensures mutually exclusive access to a critical section
  - efficiently implemented because blocking threads are de-scheduled by the operating system
- **A semaphore is a third way to avoid conflicting access to critical sections**
  - a (counting) semaphore is an abstract data structure providing an unsigned int together with two operations: sem_wait and sem_post
  - semaphores are more powerful than mutexes because they can be initialized to any nonnegative value

Concluding Remarks (3)

- **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 for protecting data structures**
  - threads can signal whether they want to access the data read-only or with write priviledges
  - multiple threads can simultaneously read data
  - if a thread needs to modify the data structure, then only that thread can access the data structure during the modification
- **C functions called concurrently from multi-threaded code need to be thread-safe**
  - stateful functions that implicitly remember their state between calls are problematic
  - disregading thread-safety can lead to bugs that are very hard to find

Acknowledgements

- **Peter S. Pacheco / Elsevier**
  - for providing the lecture slides on which this presentation is based

Change log

- **1.1.2 (2017-11-24)**
  - fix typo on slide 72
- **1.1.1 (2017-11-21)**
  - cosmetics
- **1.1.0 (2017-11-20)**
  - updated for winter term 2017/18
  - fix typo on slides 4, 27, 71
  - simplify slides 35, 36
- **1.0.1 (2017-01-18)**
  - revised version of slides
- **1.0.0 (2017-01-13)**
  - initial version of slides