Programmazione di Sistemi ~~Embedded e~~ Multicore

Teacher: Daniele De Sensi

Recap

Recap

- Pi calculation example
- Mutex
- Semaphores
- Barriers and Condition Variables

Q&A

```
void* fun(void* args){
    int thread_id = *((int*) args);
    printf("I am thread %d\n", thread_id);
    return NULL;
}

int main(){
    ...
    for(int i = 0; i < num_threads; i++){
        pthread_create(&thread_handles[i], NULL, fun, (void*) &i);
    }
    ...
}
```

It does not work, why?

Q&A

```
void* fun(void* args){
    int thread_id = *((int*) args);
    printf("I am thread %d\n", thread_id);
    return NULL;
}

int main(){
    ...
    for(int i = 0; i < num_threads; i++){
        pthread_create(&thread_handles[i], NULL, fun, (void*) &i);
    }
    ...
}
```

It does not work, why?

We have no guarantees that the thread has been created and that it read ((int*) args) when pthread_create returns

&i is a pointer to a variable which might get updated to i+1 before the thread manages to read it

Q&A

```
void* fun(void* args){
    int thread_id = *((int*) args);
    printf("I am thread %d\n", thread_id);
    return NULL;
}

int main(){
    ...
    int* ids = (int*) malloc(sizeof(int)*num_thread);
    for(int i = 0; i < num_threads; i++){
        ids[i] = i;
        pthread_create(&thread_handles[i], NULL, fun, (void*) &(ids[i]));
    }
     for(int i = 0; i < num_threads; i++){
        pthread_join(thread_handles[i], NULL);
    }
    free(ids);
    ...
}
```

Correct way of doing it

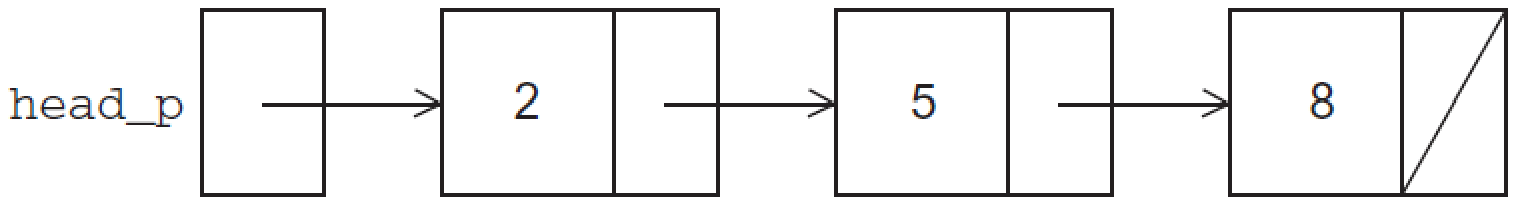
Questions?

Read-Write Locks

Controlling access to a large, shared data structure

- Let's look at an example.
- Suppose the shared data structure is a **sorted** linked list of ints, and the operations of interest are Member, Insert, and Delete.

Linked Lists

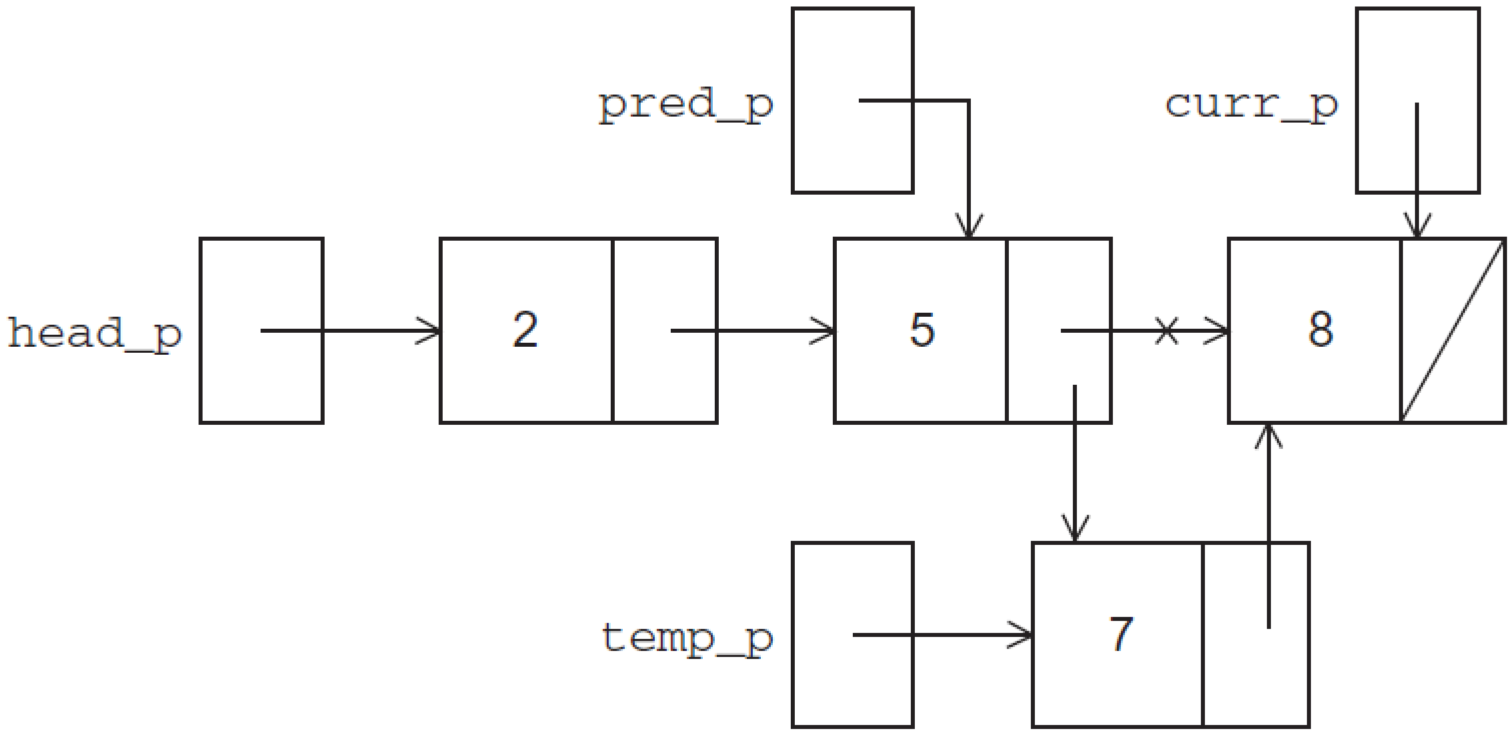



```
struct list_node_s {
   int data;
   struct list_node_s* next;
}
```

Linked List Membership

```
int  Member(int value, struct list_node_s* head_p) {
   struct list_node_s* curr_p = head_p;

   while (curr_p != NULL && curr_p->data < value)
      curr_p = curr_p->next;

   if (curr_p == NULL || curr_p->data > value) {
      return 0;
   } else {
      return 1;
   }
}  /* Member */
```

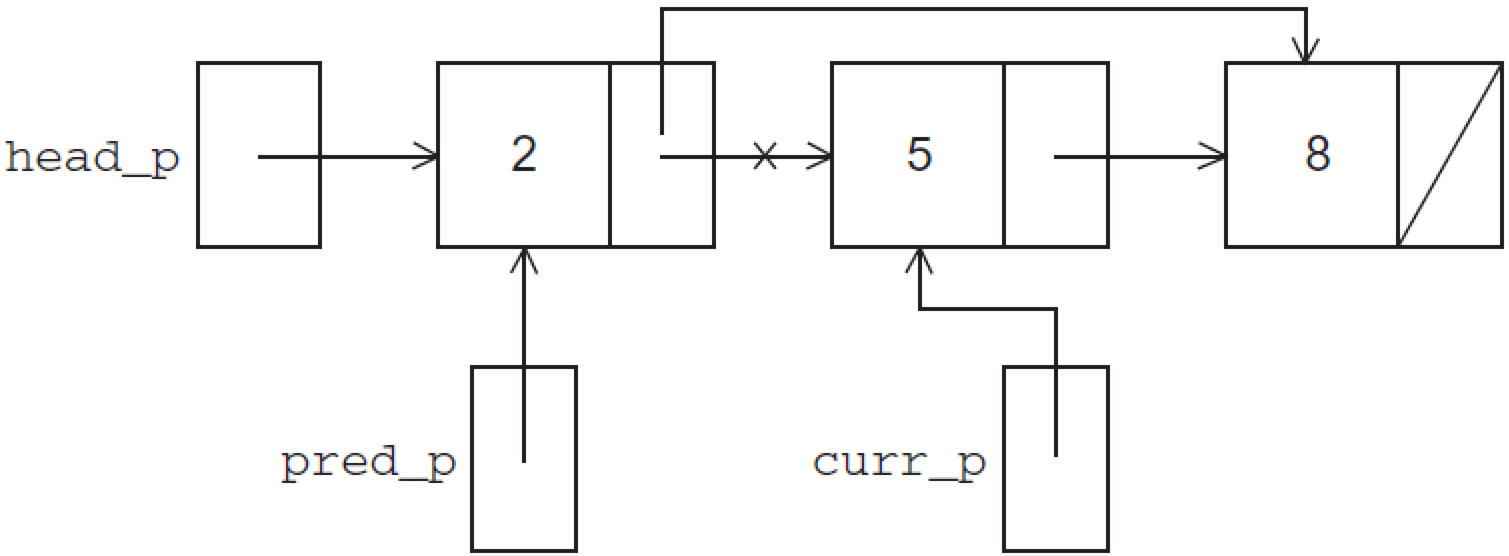
Inserting a new node into a list



Inserting a new node into a list

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

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

   if (curr_p == NULL || curr_p->data > value) {
      temp_p = malloc(sizeof(struct list_node_s));
      temp_p->data = value;
      temp_p->next = curr_p;
      if (pred_p == NULL)  /* New first node */
         *head_pp = temp_p;
      else
         pred_p->next = temp_p;
      return 1;
   } else { /* Value already in list */
      return 0;
   }
}  /* Insert */
```

Deleting a node from a linked list



Deleting a node from a linked list

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

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

   if (curr_p != NULL && curr_p->data == value) {
      if (pred_p == NULL) { /* Deleting first node in list */
         *head_pp = curr_p->next;
         free(curr_p);
      } else {
         pred_p->next = curr_p->next;
         free(curr_p);
      }
      return 1;
   } else { /* Value isn't in list */
      return 0;
   }
} /* Delete */
```

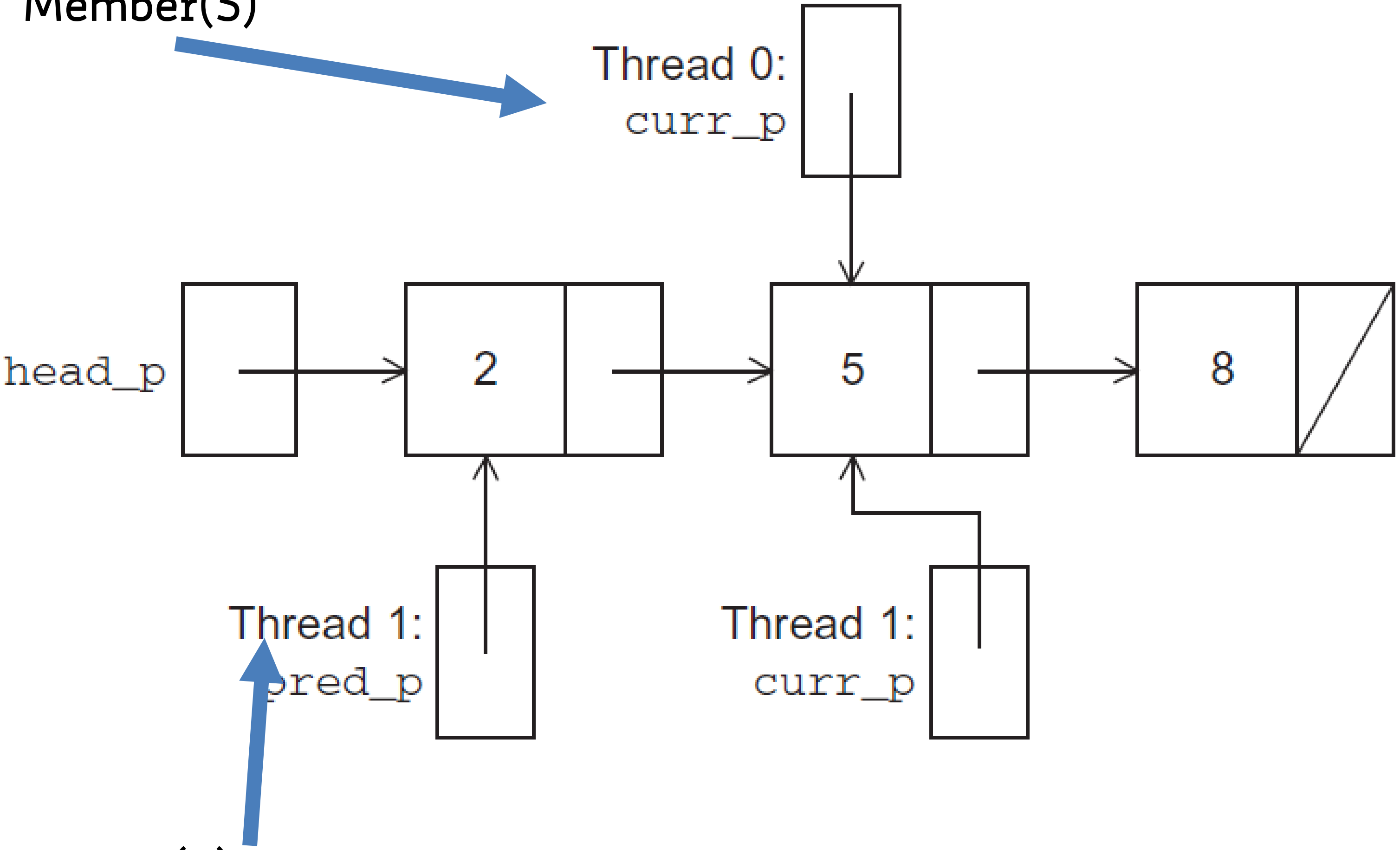
Questions?

A Multi-Threaded Linked List

- Let's try to use these functions in a Pthreads program.
- In order to share access to the list, we can define head_p to be a global variable.
- This will simplify the function headers for Member, Insert, and Delete, since we won't need to pass in either head_p or a pointer to head_p: we'll only need to pass in the value of interest.
- If multiple threads call Member at the same time, we are fine
- What if one thread calls Member while another thread is deleting an element?

Simultaneous access by two threads

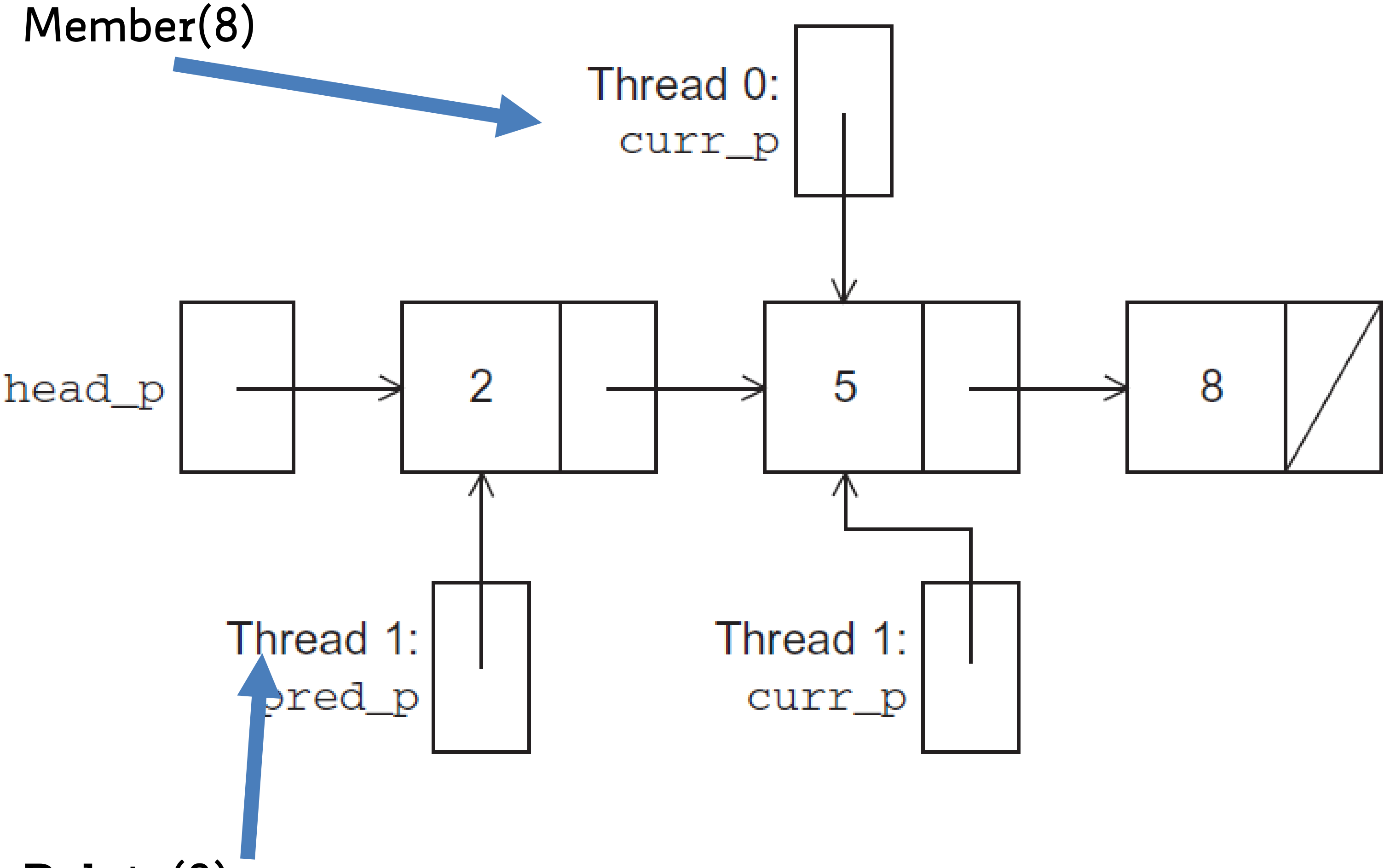
Executing **Member(5)**



Executing **Delete(5)**

Simultaneous access by two threads

Executing **Member(8)**



Executing **Delete(8)**

Solution #1

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

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

Issues

- We're serializing access to the list.
- If the vast majority of our operations are calls to Member, we'll fail to exploit this opportunity for parallelism.
- On the other hand, if most of our operations are calls to Insert and Delete, then this may be the best solution since we'll need to serialize access to the list for most of the operations, and this solution will certainly be easy to implement.

Solution #2

- Instead of locking the entire list, we could try to lock individual nodes.
- A "finer-grained" approach.

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

Issues

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

Questions?

Pthreads Read-Write Locks

- Neither of our multi-threaded linked lists exploits the potential for simultaneous access to any node by threads that are executing Member.
- The first solution only allows one thread to access the entire list at any instant.
- The second only allows one thread to access any given node at any instant.

Pthreads Read-Write Locks

- A read-write lock is somewhat like a mutex except that it provides two lock functions.

- The first lock function locks the read-write lock for reading, while the second locks it for writing.

Pthreads Read-Write Locks

- So multiple threads can simultaneously obtain the lock by calling the read-lock function, while only one thread can obtain the lock by calling the write-lock function.
- Thus, if any threads own the lock for reading, any threads that want to obtain the lock for writing will block in the call to the write-lock function.
- If any thread owns the lock for writing, any threads that want to obtain the lock for reading or writing will block in their respective locking functions.

pthread_rwlock functions

pthread_rwlock_init initializes the rwlock

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

pthread_rwlock_destroy frees the rwlock

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

We can set it to NULL.
Used to specify if readers must have priority over writers, or viceversa.
Check **pthread_rwlockattr_setkind_np**

Protecting our linked list functions

```
pthread_rwlock_rdlock(&rwlock);
Member(value);
pthread_rwlock_unlock(&rwlock);
. . .
pthread_rwlock_wrlock(&rwlock);
Insert(value);
pthread_rwlock_unlock(&rwlock);
. . .
pthread_rwlock_wrlock(&rwlock);
Delete(value);
pthread_rwlock_unlock(&rwlock);
```

Linked List Performance

Implementation	Number of Threads 1	2	4	8
Read-Write Locks	0.213	0.123	0.098	0.115
One Mutex for Entire List	0.211	0.450	0.385	0.457
One Mutex per Node	1.680	5.700	3.450	2.700

100,000 ops/thread

99.9% Member

0.05% Insert

0.05% Delete

8 (physical) cores

Linked List Performance

Implementation	Number of Threads 1	2	4	8
Read-Write Locks	0.213	0.123	0.098	0.115
One Mutex for Entire List	0.211	0.450	0.385	0.457
One Mutex per Node	1.680	5.700	3.450	2.700

100,000 ops/thread

99.9% Member

0.05% Insert

0.05% Delete

8 (physical) cores

Linked List Performance

Implementation	Number of Threads 1	2	4	8
Read-Write Locks	0.213	0.123	0.098	0.115
One Mutex for Entire List	0.211	0.450	0.385	0.457
One Mutex per Node	1.680	5.700	3.450	2.700

100,000 ops/thread

99.9% Member

0.05% Insert

0.05% Delete

8 (physical) cores

Linked List Performance

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

100,000 ops/thread

80% Member

10% Insert

10% Delete

8 (physical) cores

Linked List Performance

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

100,000 ops/thread

80% Member

10% Insert

10% Delete

8 (physical) cores

Take-home message

Read-write locks give performance gains as long as the number of insertion/deletion is small compared to the number of member operations

Questions?

Thread-Safety

Example

- A block of code is thread-safe if it can be simultaneously executed by multiple threads without causing problems.
- Suppose we want to use multiple threads to “tokenize” a file that consists of ordinary English text.
- The tokens are just contiguous sequences of characters separated from the rest of the text by white-space — a space, a tab, or a newline.

Simple approach

- Divide the input file into lines of text and assign the lines to the threads in a round-robin fashion.

- The first line goes to thread 0, the second goes to thread 1, . . . , the tth goes to thread t, the t +1st goes to thread 0, etc.

Simple approach

- We can serialize access to the lines of input using semaphores.
  - Why semaphores and not mutex?
- After a thread has read a single line of input, it can tokenize the line using the strtok function.

The strtok function

- The first time it's called the string argument should be the text to be tokenized.
  - Our line of input.
- For subsequent calls, the first argument should be NULL.

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

The strtok function

- The idea is that in the first call, strtok caches a pointer to string, and for subsequent calls it returns successive tokens <u>taken from the cached pointer</u>.

Multi-threaded tokenizer (1)

sems[0] initialized to 1
sems[i] (with i > 0) initialized to 0

```
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 */
```

Running with one thread

- It correctly tokenizes the input stream.

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

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 pointer to the input line by declaring a variable to have static storage class.
- 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.

What happened?

- 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.
- So the strtok function is not thread-safe. If multiple threads call it simultaneously, the output may not be correct.

Other unsafe C library functions

- Regrettably, it's not uncommon for C library functions to fail to be thread-safe.
- The random number generator random in stdlib.h.
- The time conversion function localtime in time.h.
- In older version of POSIX systems srand/rand were also not thread safe

"re-entrant" 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 */);
```

- In principle "re-entrant" != "thread safe"
- In practice, re-entrant functions are often also thread safe
- When in doubt, check the documentation!
  - (man strtok_r)

Questions?

Misc

Static Initializers

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
```

Thread Pinning

What if I want to force a thread to run on a specific core?

```
#define _GNU_SOURCE
#include <pthread.h>

void* thread_func(void* thread_args){
    ...
    cpu_set_t cpuset;
    pthread_t thread = pthread_self();
    /* Set affinity mask to include core 3 */
    CPU_ZERO(&cpuset);
    CPU_SET(3, &cpuset);
    s = pthread_setaffinity_np(thread, sizeof(cpuset), &cpuset);
    ...
}
```

Timing Code

```
#include <sys/time.h>

#define GET_TIME(now) { \
   struct timeval t; \
   gettimeofday(&t, NULL); \
   now = t.tv_sec + t.tv_usec/1000000.0; \
}

int main(){
    …
    double start, finish, elapsed;
    GET_TIME(start);
    . . .
    // Code to be timed
    . . .
    GET_TIME(finish);
    elapsed = finish - start;
    printf("The code to be timed took %e seconds\n", elapsed);
    …
}
```

Questions?

Concluding Remarks (1)

- A thread in shared-memory programming is analogous to a process in distributed memory programming.
- However, a thread is often lighter-weight than a full-fledged process.
- In Pthreads programs, all the threads have access to global variables, while local variables usually are private to the thread running the function.

Concluding Remarks (2)

- When indeterminacy results from multiple threads attempting to access a shared resource such as a shared variable or a shared file, at least one of the accesses is an update, and the accesses can result in an error, we have a race condition.

Concluding Remarks (3)

- A critical section is a block of code that updates a shared resource that can only be updated by one thread at a time.
- So the execution of code in a critical section should, effectively, be executed as serial code.

Concluding Remarks (4)

- Busy-waiting can be used to avoid conflicting access to critical sections with a flag variable and a while-loop with an empty body.
- It can be very wasteful of CPU cycles.
- It can also be unreliable if compiler optimization is turned on.

Concluding Remarks (5)

- A mutex can be used to avoid conflicting access to critical sections as well.
- Think of it as a lock on a critical section, since mutexes arrange for mutually exclusive access to a critical section.

Concluding Remarks (6)

- A semaphore is the third way to avoid conflicting access to critical sections.
- It is an unsigned int together with two operations: sem_wait and sem_post.
- Semaphores are more powerful than mutexes since they can be initialized to any nonnegative value.

Concluding Remarks (7)

- A barrier is a point in a program at which the threads block until all of the threads have reached it.
- A read-write lock is used when it's safe for multiple threads to simultaneously read a data structure, but if a thread needs to modify or write to the data structure, then only that thread can access the data structure during the modification.

Concluding Remarks (8)

- Some C functions cache data between calls by declaring variables to be static, causing errors when multiple threads call the function.
- This type of function is not thread-safe.

Chapter 5

Shared Memory Programming with OpenMP

Roadmap

- Writing programs that use OpenMP.
- Using OpenMP to parallelize many serial for loops with only small changes to the source code.
- Task parallelism.
- Explicit thread synchronization.
- Standard problems in shared-memory programming.

OpenMP

- An API for shared-memory parallel programming.
- MP = multiprocessing
- Designed for shared-memory systems.
- System is viewed as a collection of cores or CPU's, all of which have access to main memory.

OpenMP

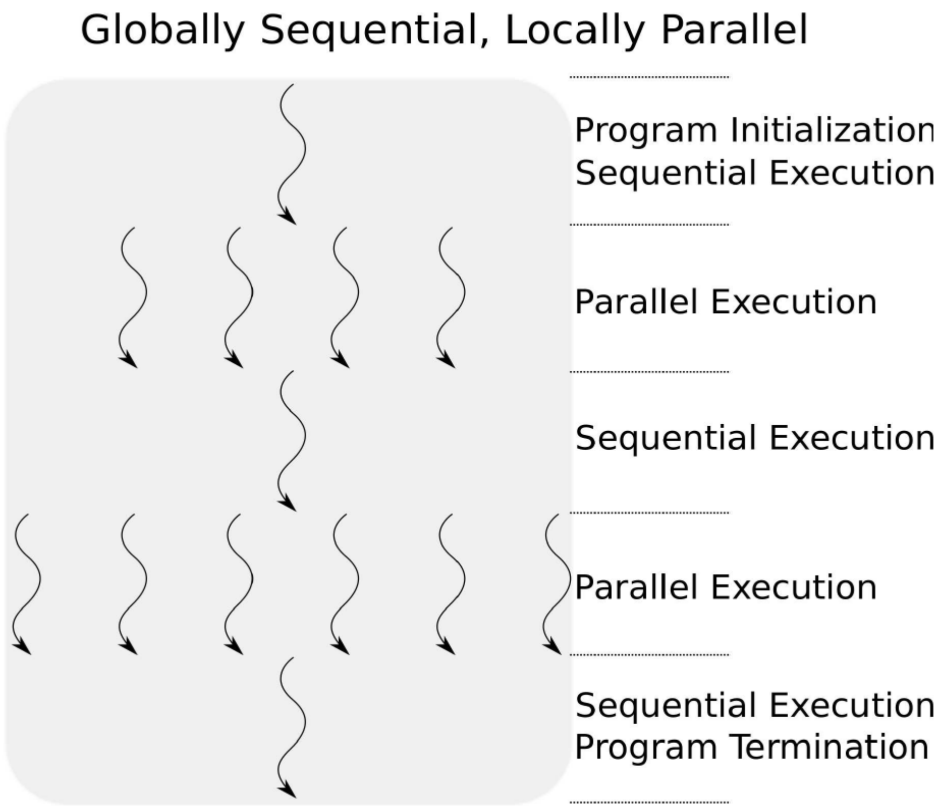
- OpenMP aims to decompose a sequential program into components that can be executed in parallel.

- OpenMP allows an "incremental" conversion of sequential programs into parallel ones, with the assistance of the compiler.

- OpenMP relies on compiler directives for decorating portions of the code that the compiler will attempt to parallelize.

OpenMP

- OpenMP programs are Globally Sequential, Locally Parallel.
- Programs follow the fork-join paradigm:



Pragmas

- Special preprocessor instructions.
- Typically added to a system to allow behaviors that aren't part of the basic C specification.
- Compilers that don't support the pragmas ignore them.

#pragma

OpenMP pragmas

- # pragma omp parallel
  - Most basic parallel directive.
  - The number of threads that run the following structured block of code is determined by the run-time system.

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void Hello(void);  /* Thread function */

int main(int argc, char* argv[]) {
   /* Get number of threads from command line */
   int thread_count = strtol(argv[1], NULL, 10);

#  pragma omp parallel num_threads(thread_count)
   Hello();

   return 0;
}  /* main */

void Hello(void) {
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   printf("Hello from thread %d of %d\n", my_rank, thread_count);

}  /* Hello */
```

```
gcc -g -Wall -fopenmp -o omp_hello omp_hello.c

./omp_hello 4
```

compiling

running with 4 threads

possible outcomes

```
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
```

```
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4
```

```
Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
```

Thread Team Size Control

- **Universally**: via the OMP_NUM_THREADS environmental variable:

```
$ echo ${OMP_NUM_THREADS}  # to query the value
$ export OMP_NUM_THREADS=4 # to set it in BASH
```

- **Program level** : via the **omp_set_num_threads** function, outside an OpenMP construct.
- **Pragma level** : via the **num_threads** clause.

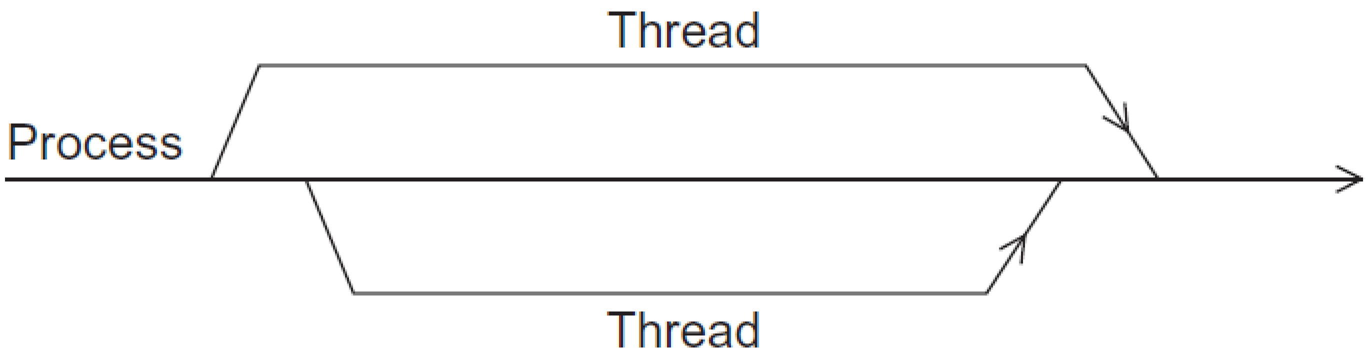
- Precedence:
  - Universally/env. Variable
  - Program level
  - Pragma level

Thread Team Size Control

- The **omp_get_num_threads** call returns the active threads in a parallel region. If it is called in a sequential part it returns 1.
- The **omp_get_thread_num** returns the id of the calling thread (similar to the rank in MPI)

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void Hello(void);  /* Thread function */

int main(int argc, char* argv[]) {
   /* Get number of threads from command line */
   int thread_count = strtol(argv[1], NULL, 10);

#  pragma omp parallel num_threads(thread_count)
   Hello();

   return 0;
}  /* main */

void Hello(void) {
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   printf("Hello from thread %d of %d\n", my_rank, thread_count);

}  /* Hello */
```

A process forking and joining two threads



clause

- Text that modifies a directive.
- The num_threads clause can be added to a parallel directive.
- It allows the programmer to specify the number of threads that should execute the following block.

```
# pragma omp parallel num_threads ( thread_count )
```

Of note…

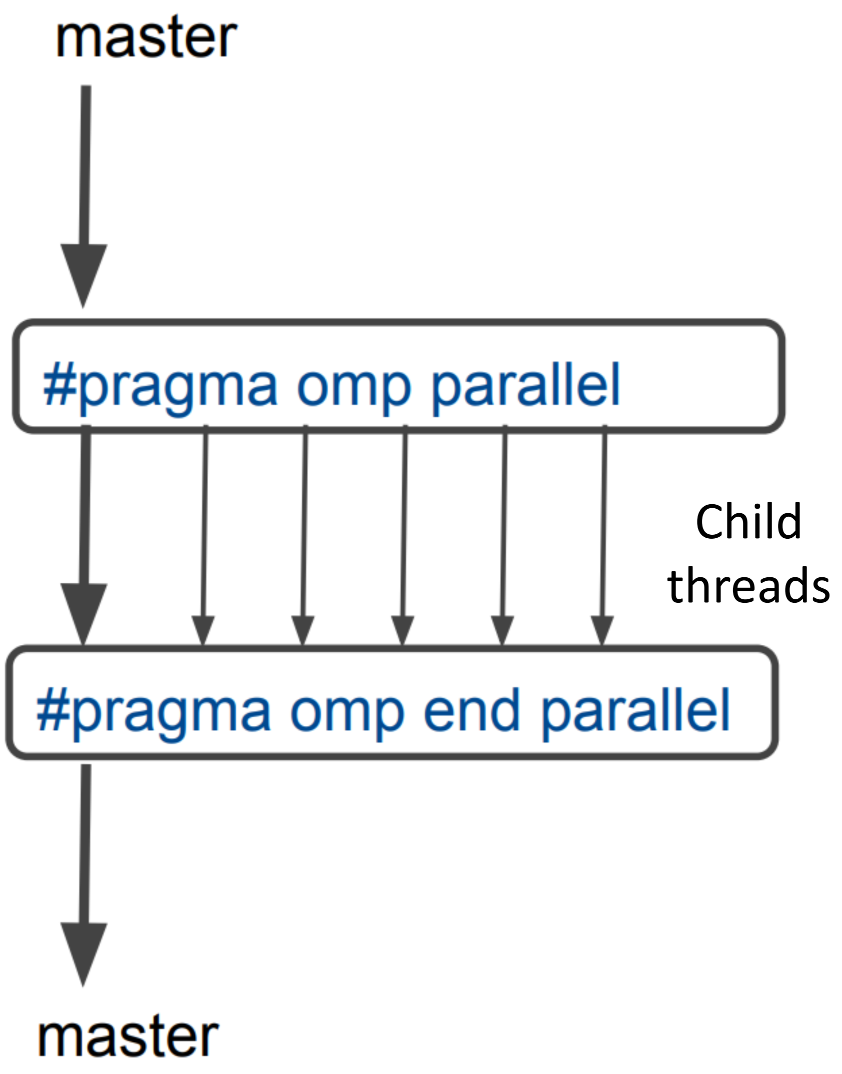
- There may be system-defined limitations on the number of threads that a program can start.
- The OpenMP standard doesn't guarantee that this will actually start thread_count threads.
- Most current systems can start hundreds or even thousands of threads.
- Unless we're trying to start a lot of threads, we will almost always get the desired number of threads.
- After a block is completed, there is an implicit barrier.

Some terminology

- In OpenMP parlance the collection of threads executing the parallel block — the original thread and the new threads — is called a team
- **master:** the original thread of execution
- **parent:** thread that encountered a parallel directive and started a team of threads.
- In many cases, the parent is also the master thread.
- **child:** each thread started by the parent is considered a child thread.

Parallel Construct

```
#pragma omp parallel
{
…
}
```



Parallel Construct

1

```c
#include<stdio.h>
#include<omp.h>

int main()
{
    printf("Only master thread here \n");

    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        printf("Hello I am thread number %d \n", tid);
    }

    printf("Only master thread here \n");
}
```

2

3

```
Only master thread here
Hello I am thread number 0
Hello I am thread number 1
Hello I am thread number 2
Hello I am thread number 3
Only master thread here
```

int tid

4

Each thread has a private copy of variable

- Specifying number of threads
  - export OMP_NUM_THREADS=4
- Execute:
  - ./basic.exe

In case the compiler doesn't support OpenMP

```
# include <omp.h>
```

```
#ifdef _OPENMP
# include <omp.h>
#endif
```

In case the compiler doesn't support OpenMP

```
# ifdef _OPENMP
   int my_rank = omp_get_thread_num ( );
   int thread_count = omp_get_num_threads ( );
# else
   int my_rank = 0;
   int thread_count = 1;
# endif
```