Parallel Programming with Threads: Performance and Other Issues

Parallel Counting and Searching

Overhead of Mutex Locks

Cache Memories - False Sharing

Deadlock

Thread Safety

Bibliography

- [Pacheco]: Peter Pacheco, Matthew Malensek, Introduction to Parallel Programming, 2nd Edition, Morgan Kaufmann Publisher, March 2020, Chapters 4.10, 4.11

Parallel Searching and Counting

The Count Example - Serial

- Count how many times a value x appears in an array a ?

```
#define NELEM 50000000 // number of elements
int a[NELEM];
int x = 3; // value x to be searched for in array a
int count; // how many times appears x in array a?

void count_serial(void)
{

    count = 0;
    for (int i = 0; i <= NELEM - 1; i++)
        if (a[i] == x)
        {

            count = count + 1;
        }
}
```

The Count Example - Parallel

- Count how many times value **x** appears in an array a ?
- The final result will be in shared variable count
- The array a is divided in **chunks** given to N_THREADS threads.
- Each chunk has elem_per_thread=NELEM/N_THREADS elements.
- The thread with number id gets the chunk containing the elements between indexes start and end:
  - start=id*elem_per_thread
  - end=(id+1)*elem_per_thread-1
- Each thread searches in its own chunk
- This **parallelization pattern** is **Data Decomposition,** and here we have a case of **Input Data Partitioning**

Count Example – V1

- Version1: Each thread works directly with the global count at every iteration
- The global count must be protected by a mutex

```
#define NELEM 50000000  // number of elements
int a[NELEM];
int x = 3;   // value x to be searched for in array a
int count;   // how many times appears x in array a?

pthread_mutex_t count_lock;  // mutex protects variable count

#define N_THREADS 8    // number of threads
int elem_per_thread = NELEM / N_THREADS;  // elements per thread
```

Count Example – V1

```
void* count_fct1(void* var)
{
    int id = *(int*)var; // thread id
    int start = id * elem_per_thread; //first elem to be processed
    int end = (id + 1) * elem_per_thread - 1; //last elem
    if (id == N_THREADS - 1) end = NELEM - 1; //last thread

    for (int i = start; i <= end; i++)
        if (a[i] == x) {
            // must use mutex lock at every occurence of x!
            pthread_mutex_lock(&count_lock);
            count = count + 1;
            pthread_mutex_unlock(&count_lock);
        }
         return NULL;
}
```

Performance – Serial vs V1

- NELEM=50000000
- **Array a has been initialized with values 1,2,3. Search x=3**
- NTHREADS=8,on a 8-core laptop
- Measured runtime:

Version	Measured time	Speedup
Serial	0.097	
Parallel V1	0.404	0.24

Count Example – V2

- Version2: Each thread works with a *local variable* lcount and at the end merges its local result into the global counter

Count Example – V2

```
void* count_fct2(void* var)
{
    int id = *(int*)var; // thread id
    int start = id * elem_per_thread; //first elem to be processed
    int end = (id + 1) * elem_per_thread - 1; //last elem
    if (id == N_THREADS - 1) end = NELEM - 1;

    int lcount = 0;  // local counter
    for (int i = start; i <= end; i++)
        if (a[i] == x) lcount++;

    pthread_mutex_lock(&count_lock);
    count = count + lcount;
    pthread_mutex_unlock(&count_lock);
    return NULL;
}
```

Performance – Serial vs V1 vs V2

- NELEM=50000000
- Array a has been initialized with values 1,2,3. Search x=3
- NTHREADS=8,on a 8-core laptop
- Measured runtime:

Version	Measured time	Speedup
Serial	0.097	
Parallel V1	0.404	0.24
Parallel V2	0.019	5.10

Dangers of Using Mutex Locks

- Simple usage of a mutex lock involves an overhead
  - In V1, each thread locks mutex for 1562500 times
  - In V2, each thread locks mutex only once
- Unreasonable usage of mutex locks can serialize a big part of a program !

- **Good practice: whenever threads must contribute frequent updates by an associative operation towards a global result in a shared variable, try to give each thread a local copy of it and *merge local results at the end***
- ***This is a frequent pattern called reduction***
- ***Parallel programming languages or frameworks such as OpenMP provide optimized implementations of this pattern***

Count Example – V3

- Version3: We want to know how many occurrences are discovered by each thread. Each thread increments its own global counter
- Since we need a counter for every thread, and we have an array of threads, we **use a global array of counters**
- **No need for mutexes, each thread i accesses only counter[i]**

```
#define NELEM 50000000
int a[NELEM];
int x = 3;

#define N_THREADS 8
int elem_per_thread = NELEM / N_THREADS;

int count[N_THREADS];
```

Count Example – V3

```
void *count_fct3(void *var)
{
        int id = *(int *)var; // thread id
        int start = id * elem_per_thread; // first elem
        int end = (id + 1) * elem_per_thread - 1; // last elem
        if (id == N_THREADS - 1)
                end = NELEM - 1;

        for (int i = start; i <= end; i++)
                if (a[i] == x)
                        count[id]++;

        return NULL;
}
```

Performance – Serial vs V2 vs V3

- NELEM=50000000
- NTHREADS=8,on a 8-core laptop
- Measured runtime:

Version	Measured time	Speedup
Serial	0.097	
Parallel V2	0.019	5.10
Parallel V3	0.075	1.29

V3 has a performance drop due to **false sharing**

Problems Due to Cache Memories

Problems with caches

- The use of cache memory can have a huge impact on performance of programs
- It affects performance of single-thread (ST) and multi-thread (MT) programs
  - Cache miss rate (ST and MT)
  - Cache coherence (MT)
  - False sharing (MT)

Basics of caching

- Cache memory: collection of memory locations that can be accessed in less time than some other memory locations from the main memory.
- If a processor accesses main memory location *x* at time *t* , then it is likely that at times close to *t* it will access main memory locations close to *x*. (*Spatial* locality and *Temporal* locality)
- Data that fulfills *locality* condition is temporarily copied from main memory into cache, used there, and then copied back into main memory
- If a processor needs to access main memory location *x*, a ***block of memory*** containing *x* is transferred from/to the processor's cache. Such a block of memory is called a **cache line** or **cache block**.

Cache hit and miss

- *write-miss* occurs when a core tries to update a variable that's not in the cache, and it has to access main memory.
- a *read-miss* occurs when a core tries to read a variable that's not in the cache, and it has to access main memory
- Cache miss rate is a problem that affects performance in general, even with one single thread

- *Experiment to illustrate the effect of **cache miss rate***:
- On a system where 2D arrays (matrix) are *stored in row-major order*:
- Matrix addition, single thread, in 2 variants: matrix visited row-wise (for i for j) vs matrix visited column-wise(for j for i):
  - For matrix size 5000*5000 elements: **1.49**sec vs **9.82**sec

Cache coherence problem

- Consider x is a shared variable, and two threads, Thread0 and Thread1 read x from main memory into their separate caches
- There are three copies of x: the one in main memory, the one in Thread0's cache, and the one in Thread1's cache.
- Now suppose Thread0 executes the statement x ++;
- This is **the cache coherence problem:** ***A global variable has copies in multiple caches of multiple threads. If one copy gets changed, all copies must be invalidated!***

False sharing problem

- **Cache coherence is enforced at the “cache-line level”:**
  - 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***—not just the value that was written.
- Suppose two threads with separate caches access and update ***different variables that belong to the same cache line***
  - ***This is a form of “false sharing”***

False sharing Example

- Example: two global variables x and y happen to be in neighbour locations in memory, thus they will be *on the same cache line*.
- Thread 0 updates only x
- Thread1 updates only y.
- Even though neither thread updates a variable that the other thread is using, the cache controller invalidates every time 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 behaviour of the threads with respect to memory access is the same as if they were sharing a variable, hence the name ***false sharing***.

Count Example – V3

- Version3: we used an array of counters. No need for mutexes, each thread i accesses only counter[i]

```
int count[N_THREADS];
```

The elements of the array count are on the same cache line so it appears as **false sharing.**
To avoid this: force every counter used by another thread on a new cache line.

Count Example – V4

- Version4: we **introduce padding** in the array of counters in order to forcefully space every counter used by a thread on a new cache line

```
#define NELEM 50000000
int a[NELEM];
int x = 3;

#define N_THREADS 8
int elem_per_thread = NELEM / N_THREADS;

#define PADDINGSIZE 124

struct padded_int {
        int value;
        char padding[PADDINGSIZE];  // padding avoids false sharing!
} count[N_THREADS];
```

PADDINGSIZE + sizeof(value)
= length of cache line
64Bytes or 128Bytes

Count Example – V4

```
void *count_fct4(void *var)
{
        int id = *(int *)var;
        int start = id * elem_per_thread;
        int end = (id + 1) * elem_per_thread - 1;
        if (id == N_THREADS - 1) end = NELEM - 1;

        for (int i = start; i <= end; i++)
                if (a[i] == x)
                        (count[id].value)++;

         return NULL;
}
```

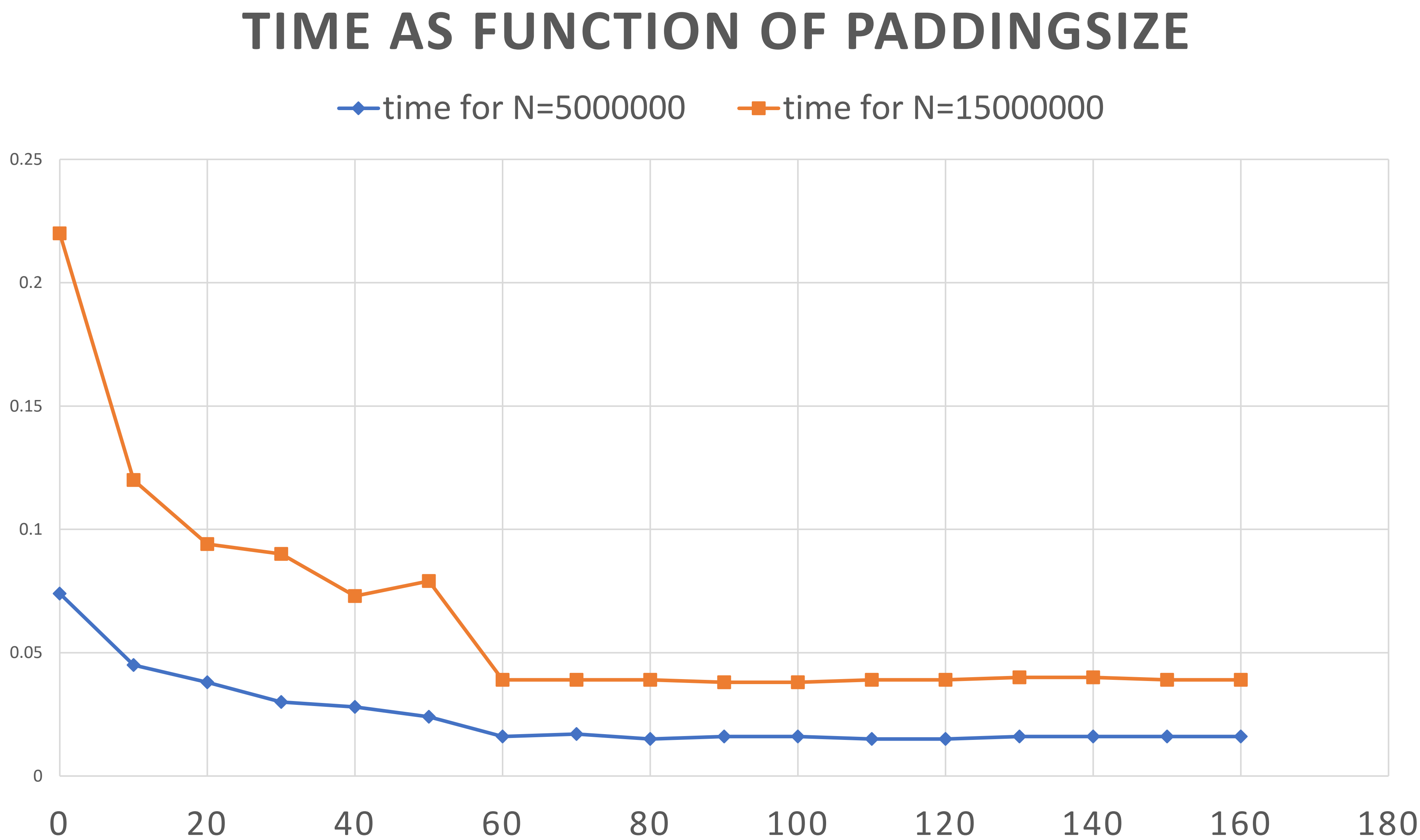
Performance – Serial vs V3 vs V4

- NELEM=50000000
- NTHREADS=8,on a 8-core laptop
- Measured runtime:

Version	Measured time	Speedup
Serial	0.097	
Parallel V3	0.075	1.29
Parallel V4	0.019	5.10

With false sharing

No false sharing


TIME AS FUNCTION OF PADDINGSIZE
time for N=5000000
time for N=15000000
0.25
0.2
0.15
0.1
0.05
0
0
20
40
60
80
100
120
140
160
180

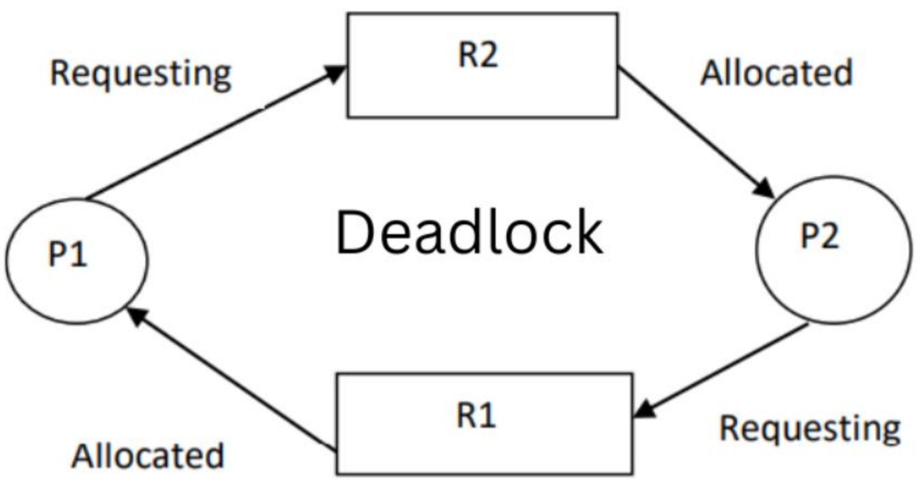
Source code

- performance_count1.c
- performance_count2.c
- performance_count3.c
- performance_count4.c

Deadlocks

Deadlocks

- All of the mutual exclusion mechanisms can cause deadlocks
- In a **deadlock**, a process or thread enters an ***infinite waiting*** state ***because a resource*** requested by it is being held by another waiting process, which in turn is waiting for another resource held by another waiting process and somewhere there is a dependency back to the original process or thread



Deadlock Example

- Example: a program has two shared data structures (*fork* and *knife*), each of which has an associated mutex. In order to perform its task of *eating*, a thread must acquire both mutex locks
- **Scenario that can lead to deadlock**: threads are programmed such that they try to acquire mutexes in a *different order*:
  - Thread0: first takes fork, then takes knife
  - Thread1: first takes knife, then takes fork.
- **Safe scenario**: **both threads should try to acquire *all mutexes in the same order***

```c
int theFork;
pthread_mutex_t fork_lock;
int knife;
pthread_mutex_t knife_lock;

void *eat_fct1(void *var)
{
        int id = *(int *)var; // thread id

        for (int i = 0; i < REPEAT; i++)
        {
                printf("Thread %d wants fork \n",id);
                pthread_mutex_lock(&fork_lock);
                printf("Thread %d has fork wants knife \n",id);
                pthread_mutex_lock(&knife_lock);
                // uses fork and knife
                sleep(1);
                printf("Thread %d has fork and knife \n",id);
                pthread_mutex_unlock(&knife_lock);
                pthread_mutex_unlock(&fork_lock);
        }

        return NULL;
}
```

Deadlock Example

```
C:\Users\ioana\Desktop\APD\eu>d
start
Thread 0 wants fork
Thread 0 has fork wants knife
Thread 0 has fork and knife
Thread 1 wants knife
Thread 1 has knife wants fork
Thread 1 has fork and knife

C:\Users\ioana\Desktop\APD\eu>d
start
Thread 0 wants fork
Thread 1 wants knife
Thread 1 has knife wants fork
Thread 0 has fork wants knife
```

The deadlock situation does *not* necessarily appear at *every* run.
Such a program is still faulty!

Code Examples

- deadlock.c

Avoiding Deadlocks

- Deadlock: situation in which a set of threads/processes cannot proceed because each requires resources held by another member of the set.
- Design resource allocation strategies that guarantee that no circular wait can happen
  - For complex scenarios: Resource Allocation Graphs
- Use timed versions of mutual exclusion primitives:
  - pthread_mutex_timedlock, sem_timedwait, pthread_cond_timedwait

Thread Safety

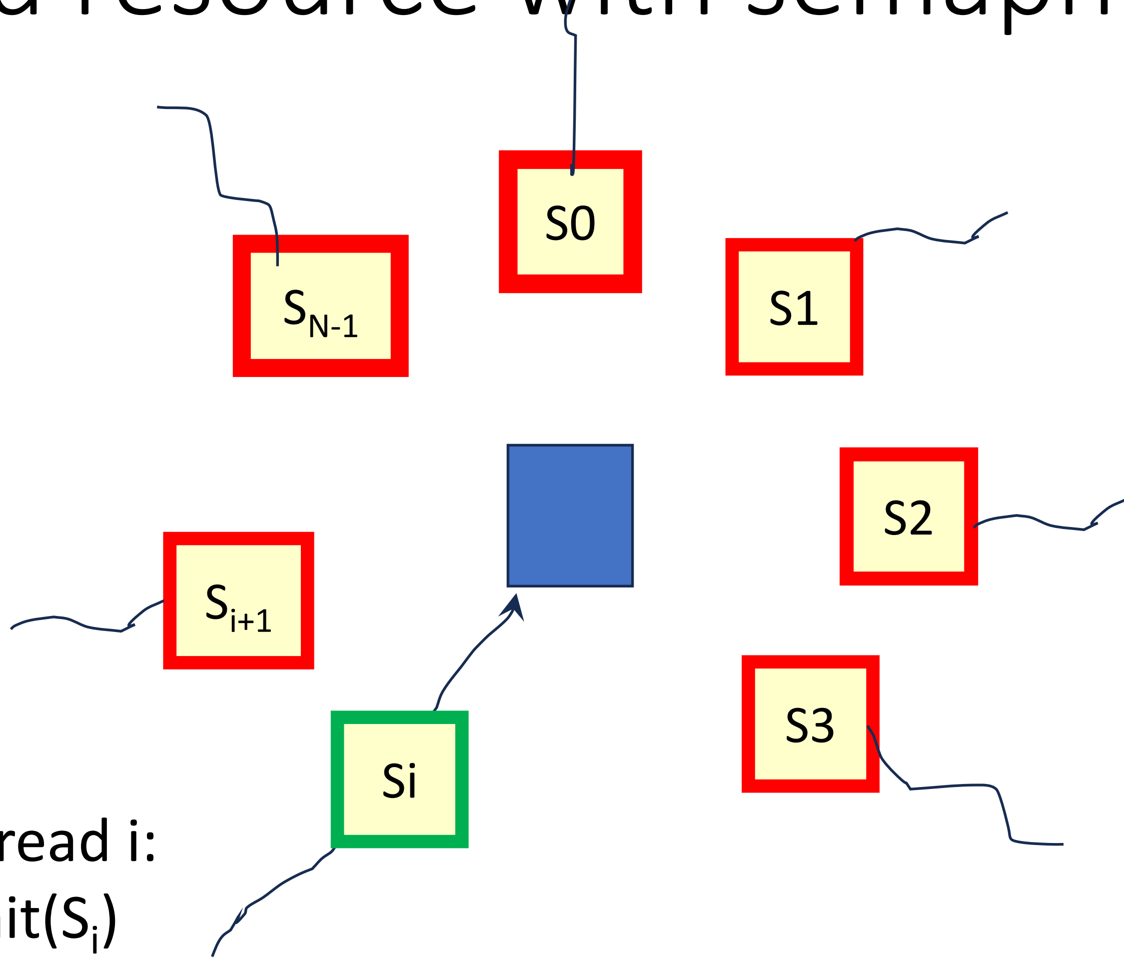
Thread-Safety

- A block of code is thread-safe if it can be simultaneously executed by multiple threads without causing problems
- Some C functions are implemented such that they cache data between calls by declaring variables to be static.
  - This is causing errors when multiple threads call the function
  - This type of function is not thread-safe!

Thread-safety Example: Round-robin tokenizer

- We want to use multiple threads to “tokenize” a file that contains text = words separated by white-space
- Simple approach:
  - Each thread reads a single line of input, and tokenizes the line using the strtok function.
  - We protect access to the file for reading a line using a mutual exclusion mechanism
  - We use ***semaphores*** in order to ***force*** threads to get lines in a ***round-robin*** way

Enforce *Round-robin* access to a shared resource with semaphores



Thread i:
wait(S_i)
post(S_{i+1})
Initialization: S0 with 1, all others with 0

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.
- 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 copy</u>.

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

Multi-threaded tokenizer (1)

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

   /* Force sequential reading of the input in round-robin way */
   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);

      count = 0;
      my_string = strtok(my_line, " \t\n");
```

Continues on next slide …

Multi-threaded tokenizer (2)

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

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

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

Problem with this tokenizer:

- strtok caches 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.
- This cached string is shared, not private!
- The strtok function is not thread-safe.
  If multiple threads call it simultaneously, the output may not be correct.
  - Example: thread 0's call to strtok with the third line of the input can overwrite the contents of thread 1's call with the second line.

“re-entrant” (thread safe) functions

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

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

Other unsafe C library functions

- The random number generator random in stdlib.h
- The time conversion function localtime in time.h

Thread-safety and POSIX

- https://unix.org/whitepapers/reentrant.html

Handling input-output

- From https://unix.org/whitepapers/reentrant.html
- See also https://pubs.opengroup.org/onlinepubs/9799919799/functions/V2_chap02.html#tag_16_05
- The POSIX.1 and C-language **functions that operate on standard character I/O streams (represented by pointers to objects of type FILE)** are required by POSIX.1c to be implemented in such a way that **reentrancy is achieved** (see ISO/IEC 9945:1-1996, §8.2).
- This requirement has a drawback - performance penalties because of the synchronization result
- POSIX.1c addresses this tradeoff between reentrancy (safety) and performance by introducing high-performance, but non-reentrant (potentially unsafe), versions of the following C-language standard I/O functions: getc(), getchar(), putc() and putchar(). The non-reentrant versions are named getc_unlocked(), and so on, to stress their unsafeness.
- To make it possible for multithreaded applications to use the non-reentrant versions of the standard I/O functions safely, POSIX.1c introduces the following character stream locking functions: flockfile(), ftrylockfile() and funlockfile().
- As stated in the description of the character stream locking functions, all standard I/O functions that reference character streams shall behave as if they use flockfile() and funlockfile() internally to obtain ownership of the character streams. Thus, when an application thread locks a character stream, the standard I/O functions cannot be used by other threads to operate on the character stream until the thread holding the lock releases it.

Summary and Conclusions

- Using mutual exclusions mechanisms (mutex locks, semaphores) in wrong ways brings several dangers:
  - **Overhead** of locking
  - **Excessive serialization** of a program
  - **Deadlocks**
- Cache memories significantly influence performance in both cases of serial and parallel (multithreaded) programs
  - The cache-miss rate influences both sequential and multithreaded programs
  - **False Sharing** appears in case of multithreaded programs
- There are several **not thread-safe** functions in C standard libraries
  - Some C functions cache data between calls by declaring static variables