Shared Memory:
Virtual Shared Memory, Threads & OpenMP

Eugen Betke

University of Hamburg
Department Informatik
Scientific Computing

09.01.2012
Agenda

1. Introduction
   Architectures of Memory Systems

2. Virtual Shared Memory
   Memory Mapping
   Separated Address Space
   Fundamental Problems

3. Pthreads
   Example
   Race Conditions and Critical Sections
   Locks

4. OpenMP
   Example
   Race Conditions and Critical Sections
   Locks

5. Summary
Shared Memory vs. Distributed Memory [1]

- **Shared Memory**

- **Distributed Memory**
Shared Memory: Uniform Memory Access System

- Same access times for all the cores
- Direct connection to a block of memory
- Relative easy to program
Different access times
Different memory locations
# Table Of Content

1. **Introduction**  
   Architectures of Memory Systems

2. **Virtual Shared Memory**  
   Memory Mapping  
   Separated Address Space  
   Fundamental Problems

3. **Pthreads**  
   Example  
   Race Conditions and Critical Sections  
   Locks

4. **OpenMP**  
   Example  
   Race Conditions and Critical Sections  
   Locks

5. **Summary**
- A system that uses physical addressing
Virtual Memory: Mapping I

- A system that uses virtual addressing
- MMU - Memory Management Unit
Virtual Memory: Mapping II

**Physical page number or disk address**

<table>
<thead>
<tr>
<th>PTE 0</th>
<th>0</th>
<th>null</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Memory resident page table (DRAM)

<table>
<thead>
<tr>
<th>PP 0</th>
<th>VP 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 2</td>
<td></td>
</tr>
<tr>
<td>VP 7</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PP 3</th>
<th>VP 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 7</td>
<td></td>
</tr>
</tbody>
</table>

Physical memory (DRAM)

Virtual memory (disk)

Eugen Betke
Virtual Memory: Mapping III

Virtual Address

Page table base register (PTBR)

Valid

The VPN acts as index into the page table

If valid=0 then page not in memory (page fault)

Physical page number (PPN)

Physical page offset (PPO)

Eugen Betke
Virtual Shared Memory: Separated Address Space

- Each process has its own Virtual Address Space
Virtual Shared Memory: Separated Address Space II

- **Simplifying linking**
  - Allows a linker to produce fully linked executables that are independent of the ultimate location of the code and data in the physical memory

- **Simplifying loading**
  - Easy to load executable and shared objects into memory

- **Simplifying sharing**
  - Mapping to the same physical page

- **Simplifying memory allocation**
  - Allocation of contiguous space
Cache Coherence

Example

x is initialized to 2
y0 is private and owned by core 0
y1, z1 are private and owned by core 1

<table>
<thead>
<tr>
<th>Time</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>y0 = x;</td>
<td>y1 = 3 * x;</td>
</tr>
<tr>
<td>1</td>
<td>x = 7;</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>z1 = 4 * x;</td>
</tr>
</tbody>
</table>

What is the value of z1?

z1 = 4 * 7 = 28
or
z1 = 4 * 2 = 8
Problem: Cache Coherence

Snooping cache coherence

Cores share a bus. When core 0 updates the copy of \( x \) stored in its cache, it broadcasts this information across the bus. Core 1 is "snooping" the bus and it will see that \( x \) has been updated. He marks his copy of \( x \) as invalid.

Directory-based cache coherence

A directory stores the status of each cache line. A directory is typically distributed and each CPU/Memory pair is responsible to update the status of its own local memory. If core 0 reads a cache line in its own local cache, it writes in the directory, that he has a copy of this cache line in his local cache. When an other core modifies a variable, that lies in that cache line, the cache controller invalidates the copies in corresponding local caches.
Problem: False Sharing

Example (Serial program)

As an example, suppose we want to repeatedly call a function $f(i,j)$ and add the computed values into a vector.

```c
int i, j, m, n;
double y[m];
/* Assign y = 0 */
...
for (i = 0; i < m; i++)
    for (j = 0; i < n; i++)
        y[i] += f(i,j);
```
Problem: False Sharing II

Example (Parallel program)

```c
/* Private variable */
int i, j, iter_count;
/* Shared variables initialized by one core */
int m, n, core_count;
double y[m];

iter_count = m/core_count;

/* Core 0 does this */
for (i = 0; i < iter_count; i++)
    for (j = 0; i < n; i++)
        y[i] += f(i,j);

/* Core 1 does this */
for (i = iter_count + 1; i < 2 * iter_count; i++)
    for (j = 0; i < n; i++)
        y[i] += f(i,j);
```

▶ m = 8
▶ doubles are 8 bytes
▶ cache line can store eight doubles (64 bytes)
▶ y takes one full cache line

What happens?
Pthreads

- Pthreads - POSIX threads
- Specifies a library for Unix-like systems
- It exists other specifications like: Java threads, Windows threads, Solaris threads. All of the specifications support the same basic ideas.
 Threads are often called light-weight processes
 Master thread forks slave threads
 Slave threads joins to the master thread
 Typical approaches to thread startup:
  - Static threads: all threads are created before computation
    - Example: Computation of scalar product
  - Dynamic threads: threads are created at demand
    - Example: Web server applications, that responds to client requests
# Pthreads: Example I

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

/* Global variable: accessible to all threads */
int thread_count;

void *Hello(void* rank); /* Thread function */
```
```c
int main(int argc, char* argv[]) {
    long thread; /* Use long in case of a 64-bit system */
    pthread_t* thread_handles;

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

    thread_handles = malloc (thread_count*sizeof(pthread_t));

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

    printf("Hello from the main thread\n");

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

    free(thread_handles);
    return 0;
} /* main */
```
void *Hello(void *rank) {
    long my_rank = (long) rank; /* Use long in case of 64-bit system */
    printf("Hello from thread\%ld of \%d\n", my_rank, thread_count);
    return NULL;
} /* Hello */
Pthreads: Forking and Joining functions

- **Starting/forking threads**

```c
int pthread_create(
    pthread_t* thread_p,
    const pthread_attr_t* attr_p,
    void* (*start_routine)(void*),
    void* arg_p);
```

- **Thread function**

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

- **Stopping/joining threads**

```c
int pthread_join(
    pthread_t thread,
    void** ret_val_p);
```
Pthreads: Example execution

- **Compilation**

  ```
gcc -g -Wall -o pth_hello pth_hello.c -lpthreads
  ```

- **Execution with 1 thread**

  ```
  ./pth_hello 1
  ```

  1. Hello from the main thread
  2. Hello from thread 0 of 1

- **Execution with 4 threads results in a non-deterministic output**

  ```
  ./pth_hello 4
  ```

  1. Hello from the main thread
  2. Hello from thread 0 of 4
  3. Hello from thread 1 of 4
  4. Hello from thread 2 of 4
  5. Hello from thread 3 of 4

  1. Hello from thread 0 of 4
  2. Hello from thread 2 of 4
  3. Hello from thread 1 of 4
  4. Hello from the main thread
  5. Hello from thread 3 of 4
Race condition

When several threads attempt 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 access can result in an error, we have a race condition.
Example

Suppose that we have two threads. Each thread computes a value and stores it in a private variable \( y \). We want add these both values to the shared variable \( x \).

```
1. \( y = \text{Compute}(\text{my\_rank}); \)
2. \( x = x + y; \)
```

<table>
<thead>
<tr>
<th>Time</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Started by main thread</td>
<td>Started by main thread</td>
</tr>
<tr>
<td>2</td>
<td>Call \text{Compute}()</td>
<td>Call \text{Compute}()</td>
</tr>
<tr>
<td>3</td>
<td>Assign ( y = 1 )</td>
<td>Assign ( y = 2 )</td>
</tr>
<tr>
<td>4</td>
<td>Put ( x = 0 ) and ( y = 1 ) into registers</td>
<td>Put ( x = 0 ) and ( y = 2 ) into registers</td>
</tr>
<tr>
<td>5</td>
<td>Add 0 and 1</td>
<td>Add 0 and 2</td>
</tr>
<tr>
<td>6</td>
<td>Store 1 in memory location ( x )</td>
<td>Store 2 in memory location ( x )</td>
</tr>
</tbody>
</table>

The “winner” result will be overwritten by the “loser”.  

Eugen Betke
Pthreads: Locks

- Locks
  - busy-waiting
  - mutexes (mutual exclusions)
  - semaphores
  - read-write locks
Formula for computing $\pi$

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

Serial computation of $\pi$

```c
double Serial_pi(long long n) {
    double sum = 0.0;
    long long i;
    double factor = 1.0;
    for (i = 0; i < n; i++, factor = -factor) {
        sum += factor/(2*i+1);
    }
    return 4.0*sum;
} /* Serial_pi */
```
void* Thread_sum(void* rank) {
    long my_rank = (long) rank;
    double factor;
    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) {
        sum += factor/(2*i+1); // Critical Section
    }

    return NULL;
} /* Thread_sum */
Pthreads: Busy-Waiting

"Concept of Busy-Waiting"

1. \( y = \text{Compute}(\text{my\_rank}); \)
2. while (flag != my\_rank);
3. \( x = x + y; \)
4. flag++;

- Thread keep re-executing the test until the test is false
- Simple implementation with a busy-wait loop
- Programmer can control the order of execution of threads
- Consumes CPU cycles
- Can seriously degrade performance
Pthreads: Example of Busy-Waiting

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 */
Abbreviation of mutual exclusions

A mutex is a special type of a variable
A variable of type `pthread_mutex_t` needs to be initialized before it (a mutex) can be used.

- **Initialization of a mutex**
  ```c
  int pthread_mutex_init(
      pthread_mutex_t* mutex_p,
      const pthread_mutexattr_t* attr_p);
  ```

- **Destruction of a mutex**
  ```c
  int pthread_mutex_destroy(pthread_mutex_t* mutex_p);
  ```

- **Gain access to a critical section**
  ```c
  int pthread_mutex_lock(pthread_mutex_t* mutex_p);
  ```

- **Unlock critical section**
  ```c
  int pthread_mutex_unlock(pthread_mutex_t* mutex_p);
  ```
Semaphores can be thought as a special type of unsigned int
- They can take values 0, 1, 2, ...
- Binary semaphore takes 0 and 1 as values
- Value 0 means “locked” and 1 means “unlocked”

Semaphores are not a part of Pthreads and it’s necessary to add the following preprocessor directive:

```c
#include <semaphore.h>
```

It’s possible to control the order in which the threads execute the critical section
A variable of type `sem_t` needs to be initialized before it (a semaphore) can be used.

- **Initialization of a semaphore**

  ```c
  int sem_init(
      sem_t* semaphore_p,
      int shared,
      unsigned initial_val);
  ```

- **Destruction of a mutex**

  ```c
  int sem_destroy(sem_t* semaphore_p);
  ```

- **Increment semaphore**

  ```c
  int sem_post(sem_t* semaphore_p);
  ```

- **Decrement semaphore**

  ```c
  int sem_wait(sem_t* semaphore_p);
  ```
Pthreads: Read-Write Locks

- Low-level locking
- Provides two lock-functions
  - One lock function locks the read-write lock for reading
  - The other lock function locks the read-write lock for writing
A variable of type `pthread_rwlock_t` needs to be initialized before it (a rwlock) can be used.

### Initialization of rwlock

```c
int pthread_rwlock_init(
    pthread_rwlock_t* rwlock_p,
    const pthread_rwlockattr_t*);
```

### Destruction of rwlock

```c
int pthread_rwlock_destroy(pthread_rwlock_t* rwlock_p);
```

### Read-write lock for reading

```c
int pthread_rwlock_rdlock(pthread_rwlock_t* rwlock_p);
```

### Read-write lock for writing

```c
int pthread_rwlock_wrlock(pthread_rwlock_t* rwlock_p);
```

### Unlock

```c
int pthread_rwlock_unlock(pthread_rwlock_t* rwlock_p);
```
# Table Of Content

1. **Introduction**
   - Architectures of Memory Systems

2. **Virtual Shared Memory**
   - Memory Mapping
   - Separated Address Space
   - Fundamental Problems

3. **Pthreads**
   - Example
   - Race Conditions and Critical Sections
   - Locks

4. **OpenMP**
   - Example
   - Race Conditions and Critical Sections
   - Locks

5. **Summary**
OpenMP (OpenMultiProcessing)

- Standard for programming shared memory systems
- Uses library functions and preprocessor directives (pragmas)
- Requires compiler support
- Developers could incrementally parallelize existing serial programs
- Higher-level than Pthreads
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

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

int main(int argc, char* argv[]) {
    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 */
OpenMP: Example II

- **Compilation**
  
  ```bash
  gcc -g -Wall -fopenmp -o omp_hello omp_hello.c
  ```

- **Execution with 4 threads**
  
  ```bash
  ./omp_hello 4
  ```

- **Output is non-deterministic**

  ```plaintext
  Hello from thread 0 of 4
  Hello from thread 1 of 4
  Hello from thread 2 of 4
  Hello from thread 3 of 4
  ```
OpenMP pragmas always begin with `# pragma omp`. They are followed by a directive. Strings after a directive are called clauses. Clauses provide additional information for the directive.

**Example**

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

- The `parallel` directive specifies, that the structured block of code that follows should be executed by multiple threads.
- The clause `num_threads(thread_count)` specifies how many threads of the structured block below should be created.
OpenMP: Library

- `#include <omp.h>` provides predefined constants and OpenMP functions
  - `int omp_get_thread_num(void);` returns the ID of the current thread
  - `int omp_get_num_threads(void);` returns the number of threads in the team
- `#include omp.h` is only needed, if we use predefined constants or call OpenMP functions
Race Conditions and Critical Sections

Race Condition: multiple threads are attempting to access a shared resource, at least one of the accesses is an update, and the accesses can result in an error.

Example

<table>
<thead>
<tr>
<th>Time</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>global_result = 0 to register</td>
<td>finish my_result</td>
</tr>
<tr>
<td>1</td>
<td>my_result = 1 to register</td>
<td>global_result = 0 to register</td>
</tr>
<tr>
<td>2</td>
<td>add my_result to global_result</td>
<td>my_result = 2 to register</td>
</tr>
<tr>
<td>3</td>
<td>store global_result = 1</td>
<td>add my_result to global_result</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>store global_result = 2</td>
</tr>
</tbody>
</table>

1 # pragma omp critical
2 \hspace{1em} global_result = my_result;

- Only one thread can execute after \# pragma omp critical following structured block of code.
- No other thread can start execute this code until the first thread has finished.
OpenMP: Locks

- # pragma omp critical
- # pragma omp atomic
- Lock-functions in omp.h
OpenMP: critical Directive

1  # pragma omp critical(name)
2   <structured block>

- Blocks protected with critical directives with different names can be executed simultaneously.
OpenMP: atomic Directive

1 # pragma omp atomic
2 <single C assignment statement>

- Can only protect critical sections that consist of a single C assignment statement.
- Statement must have one of the following form:
  - x <op> = <expression>;
  - x++;
  - ++x;
  - x--;
  - --x;
- <op> can be one of the binary operators:
  - +, *, -, /, &, ^, |, <<, >>
- <expression> must not reference x
A variable of type `omp_lock_t` needs to be initialized before it (a lock) can be used.

- **Initialization**
  ```c
  void omp_init_lock(omp_lock_t* lock_p);
  ```

- **Set lock**
  ```c
  void omp_set_lock(omp_lock_t* lock_p);
  ```

- **Unset lock**
  ```c
  void omp_unset_lock(omp_lock_t* lock_p);
  ```

- **Destroy lock**
  ```c
  void omp_destroy_lock(omp_lock_t* lock_p);
  ```
OpenMP: reduction Clause

1. `reduction(<operator>): <variable list>`) 

- `<operator>` can be +, *, -, & , |, ^ , &&, ||
- OpenMP creates for each variable in `variable list` a private variable and stores there the result of computation
- OpenMP creates a critical section, where the results from the private variables are computed with the corresponding variable
OpenMP: Reduction Clause in C-Language

Example

Code without reduction clause:

```c
1     global_result = 0.0;
2     #pragma omp parallel num_threads(thread_count)
3       {
4           double my_result = 0.0 /* private */
5           my_result += Local_trap(double a, double b, int n);
6     #pragma omp critical
7           global_result += my_result;
8       }
```

Equivalent code with reduction clause:

```c
1     #pragma omp parallel num_threads(thread_count) \
2       reduction(+: global_result)
3     global_result += Local_trap(double a, double b, int n);
```
OpenMP provides many more other directives, clauses and library functions.

- Parallelization of for-loops
- Barriers and condition variables
Table Of Content

1 Introduction
Architectures of Memory Systems

2 Virtual Shared Memory
Memory Mapping
Separated Address Space
Fundamental Problems

3 Pthreads
Example
Race Conditions and Critical Sections
Locks

4 OpenMP
Example
Race Conditions and Critical Sections
Locks

5 Summary
Summary

- Memory Systems
  - Shared and distributed systems
  - Uniform and non-uniform memory systems
- Virtual shared memory
  - Virtual addressing
  - Problems: cache coherence and false sharing
- Pthreads and OpenMP
  - Example
  - Race conditions
  - Locks