Introduction to Lab 2

Andreas Sandberg <andreas.sandberg@it.uu.se>
Division of Computer Systems
Uppsala University
2009-11-17

What lab 1?
... or what is consistency, and who cares anyway?

The purpose of this assignment is to give insights into:
1. how to program multi-processors
2. why synchronization is needed
3. how synchronization may be implemented
4. how memory consistency affects program behavior
5. performance issues specific to multiprocessors

What is a process?

A process contains the following:
- A set of memory mappings (heap, code, etc)
- Environment variables
- Signal handlers
- A list of open file descriptors (files, devices, network connections, etc)
- UID/GID/PID and some more TLAs
- One or more threads.

What is a thread?

A thread is an independent flow of control within a process.

1Three Letter Abbreviations
**What is a thread?**

A thread contains:
- A set of registers. Including:
  - Program Counter
  - Stack Pointer
- A scheduling priority

**Why do we need synchronization?**

```c
if (balance > amount)
    balance = balance - amount;
```

What happens if multiple threads execute the code above at the same time?

**How do we update shared state correctly?**

*Bringing order to chaos*

Two common approaches:
- Use critical sections
  - Heavy-weight approach.
  - Operating systems usually provide an API to do this.
- Atomic instructions
  - Relatively light-weight compared to above method.
  - Serializes memory accesses on the system.
  - May need to write assembly or use compiler pragmas/intrinsics.

**x86 memory ordering**

- Two different ways of ordering in write-back memory:
  - Causal Consistency for “normal” memory operations.
  - Total Lock Order for instructions with the lock prefix.
Consistency in the x86
POSIX Threads
Dekker's algorithm
Gauss-Seidel

**What is Causal Consistency?**
An incomplete description

In an individual processor:
- Writes are not reordered with other writes.
- Reads may be reordered with older writes to different locations.

In a multi-processor system:
- Writes by a single processor are observed in the same order by all processors.
- Writes from an individual processor are not ordered with respects to writes from other processors.
- Memory ordering obeys causality.
- Any two stores are seen in a consistent order by processors other than those performing the store.

Forcing memory order

It is possible to force memory ordering using memory fences.

Assembly:
MFENCE

GCC intrinsics:
__builtin_ia32_mfence();

Creating threads

```c
int pthread_create(
    pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*start_routine)(void *),
    void *arg);
```

Parameters:
- `thread` Where to store the thread ID.
- `attr` Attributes for the thread, NULL defaults.
- `start_routine` Procedure to call in the new thread.
- `arg` Argument passed to `start_routine`

Return Value:
0 if successful, error number otherwise.

**What is Posix Threads?**

Pthreads is...
- ...a standardized way to create and synchronize threads.
- ...the default threading API on most Unix systems. This includes:
  - GNU/Linux
  - (Net|Free)...BSD
  - Sun Solaris
  - Apple MacOS X
  - ...
**Waiting for threads to terminate**

```c
int pthread_join(
    pthread_t thread,
    void **value_ptr);
```

**Parameters:**
- `thread` Thread to wait for.
- `value_ptr` Pointer to variable to store return value in, NULL to discard return value.

**Return Value:**
- 0 if successful, error number otherwise.

**Mutexes**

```c
int pthread_mutex_init(
    pthread_mutex_t *mutex,
    const pthread_mutexattr_t *attr);
```

**Parameters:**
- `mutex` Pointer to mutex to initialize.
- `attr` Pointer to mutex attributes, NULL for default attributes.

**Return Value:**
- 0 if successful, error number otherwise.
**Mutexes**

**Cleanup**

```c
int pthread_mutex_destroy(
    pthread_mutex_t *mutex);
```

**Parameters:**
- `mutex` Pointer to mutex to destroy.

**Return Value:**
0 if successful, error number otherwise.

**Mutexes**

**Locking**

```c
int pthread_mutex_lock(
    pthread_mutex_t *mutex);
int pthread_mutex_unlock(
    pthread_mutex_t *mutex);
```

**Parameters:**
- `mutex` Pointer to mutex to lock or unlock.

**Return Value:**
0 if successful, error number otherwise.

**Example**

```c
static int balance = 512;
static pthread_mutex_t balance_mutex = PTHREAD_MUTEX_INITIALIZER;

static int withdraw(int amount) {
    int ret = 0;
    pthread_mutex_lock(&balance_mutex);
    if (balance > amount) {
        balance -= amount;
        ret = amount;
    }
    pthread_mutex_unlock(&balance_mutex);
    return ret;
}
```

**Barriers**

**Initialization**

```c
int pthread_barrier_init(
    pthread_barrier_t *barrier,
    const pthread_barrierattr_t *attr,
    unsigned count);
```

**Note:**
Barriers are optional in the Posix specification.

**Parameters:**
- `barrier` Pointer to barrier to initialize.
- `attr` Pointer to barrier attributes, NULL for defaults.
- `count` Number of threads to wait for.

**Return Value:**
0 if successful, error number otherwise.
**Introduction**

**Consistency in the x86**

**POSIX Threads**

**Dekker's algorithm**

**Gauss-Seidel**

**Summary**

**Barriers**

**Cleanup**

**Waiting**

```c
int pthread_barrier_destroy(
    pthread_barrier_t * barrier);
```

**Parameters:**
- **barrier** Pointer to barrier to destroy.

**Return Value:**
- 0 if successful, error number otherwise.

```c
int pthread_barrier_wait(
    pthread_barrier_t * barrier);
```

**Parameters:**
- **barrier** Pointer to barrier to wait for.

**Return Value:**
- 0 if successful, error number otherwise.

---

**Example**

```c
static pthread_barrier_t barrier;

static void init_barrier () {
    pthread_barrier_init(& barrier, NULL, 2);
}

static void destroy_barrier () {
    pthread_barrier_destroy(& barrier);
}

static void do_stuff() {
    /* TODO: Super-fancy algorithm here */
    pthread_barrier_wait(& barrier);
}
```

---

**Background**

... or who is this Dekker guy anyway?

- Dekker’s algorithm solves the critical section problem for 2 threads without fancy hardware support.
- Attributed to the Dutch mathematician **Theodorus J. Dekker** in a manuscript from 1965 by **Edsger W. Dijkstra**.
The algorithm

```c
flag, = True
while flag, do
  if turn  i then
    flag, = False
  while turn  i do
    Do nothing or sleep
  end while
  flag, = True
end if
Do critical work
turn, = j
flag, = False
```

Limitations

- Only works for two threads.
  - ...but we don't care.
- Does not work with weak consistency models.
  - Requires memory barriers to force the processor to order accesses.

In the lab

What you'll be doing (hopefully)

Your tasks are to:
- Implement a version of Dekker's algorithm in C.
- Insert the memory barriers to make the algorithm work on a real machine.

What is Gauss-Seidel?

...and why do I care?

Gauss-Seidel is...
- ...an iterative linear equation solver.
- ...ancient and low-performing on its own.
- ...used as a component in modern multi-grid solvers.
How we will use Gauss-Seidel

We will use Gauss-Seidel to solve the Laplace equation:

\[
\Delta u = 0 \quad \text{in } \Omega \\
u = 0 \quad \text{on } \partial \Omega
\]

Note: The equation above is not a linear equation system!

... but we can approximate it as one using finite differences!

\[
\Delta u_{ij} = \frac{u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{ij}}{h^2}
\]

The Gauss-Seidel algorithm

A sweep

Generally:

\[
x_{i}^{k+1} = \frac{b_i - \sum_{j \neq i} a_{ij} x_{j}^{k+1} - \sum_{j \neq i} a_{ij} x_{ji}^{k}}{a_{ii}}
\]

Applied to the Laplace equation (with \( h = 1 \)):

\[
u_{ij}^{k+1} = \frac{u_{i-1,j}^{k} + u_{i+1,j}^{k} + u_{i,j-1}^{k} + u_{i,j+1}^{k} - 4u_{ij}^{k}}{4}
\]

Testing for convergence

We define convergence as:

\[
\sum_{i} \sum_{j} |u_{ij}^{k} - u_{ij}^{k+1}| \leq t
\]

We say that the algorithm has converged when the absolute difference between two iterations is smaller than the tolerance.

Access pattern

Serial version

Each element is the average of its neighbors. The “new” value is used for the north and west neighbor.
We will parallelize column wise. This requires synchronization between the threads along the “border”.

Summary

You will...  
- ...implement Dekker’s algorithm and use memory barriers to make it run correctly on x86.
- ...implement a simple parallel version of the Gauss-Seidel algorithm.

- Complete lab manual on the course homepage

And remember...  
Thou shalt study thy libraries and strive not to reinvent them without cause, that thy code may be short and readable and thy days pleasant and productive.

Important dates

- Groups:  
  A: 2009-11-18, Room 1515, 08:15-12:00
  B: 2009-11-20, Room 1515, 13:15-17:00
- Deadline: 2009-11-26 08:14

Access pattern
Parallel version

Thread 0
Thread 1

http://www.it.uu.se/edu/course/homepage/dark2/ht09

http://www.lysator.liu.se/c/ten-commandments.html