Introduction to Lab 2

Mahdad Davari <mahdad.davari@it.uu.se>

Division of Computer Systems
Dept. of Information Technology
Uppsala University

2013-10-07

Lab 2 in a nutshell

- Multicores, Shared Memory, Synchronization, and Memory Ordering
  - Need for synchronization
  - Implementing synchronization using shared memory on multicores
  - Memory [consistency] model effect on programme behaviour
  - Heavy-weight synchronization vs. Light-weight alternatives

Process vs. Thread

- Process[1]:
  - Created by OS
  - Requires fair amount of overhead
  - Contains information about programme resources and programme execution state, e.g.
    - PID, GID, UID
    - Environment
    - Working directory
    - Instruction & data
    - Stack & heap
    - File descriptors
    - Signal action
    - IPC tools (msg. queues, pipes, semaphors, shared memory, etc.)
    - etc.

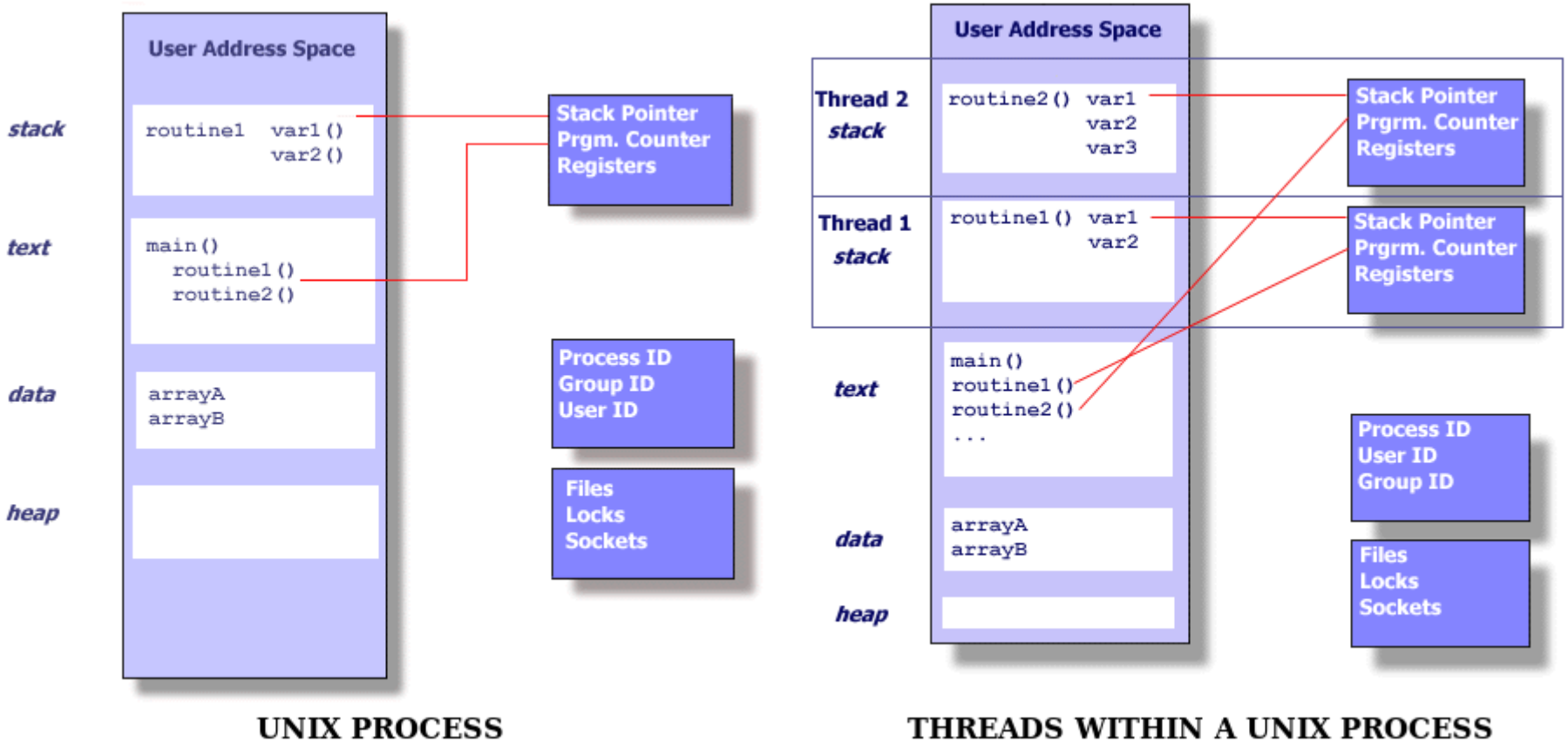
1. https://computing.llnl.gov/tutorials/pthreads

Process vs. Thread

- Threads[1]:
  - Exist within a process and share process resources
  - Scheduled individually by OS and run in parallel
  - Duplicate only bare essential resources
    - Stack pointer
    - PC
    - GP registers
    - Scheduling info.
    - Thread specific data

1. https://computing.llnl.gov/tutorials/pthreads

Process vs. Thread[1]



1. https://computing.llnl.gov/tutorials/pthreads

Synchronization Between Threads

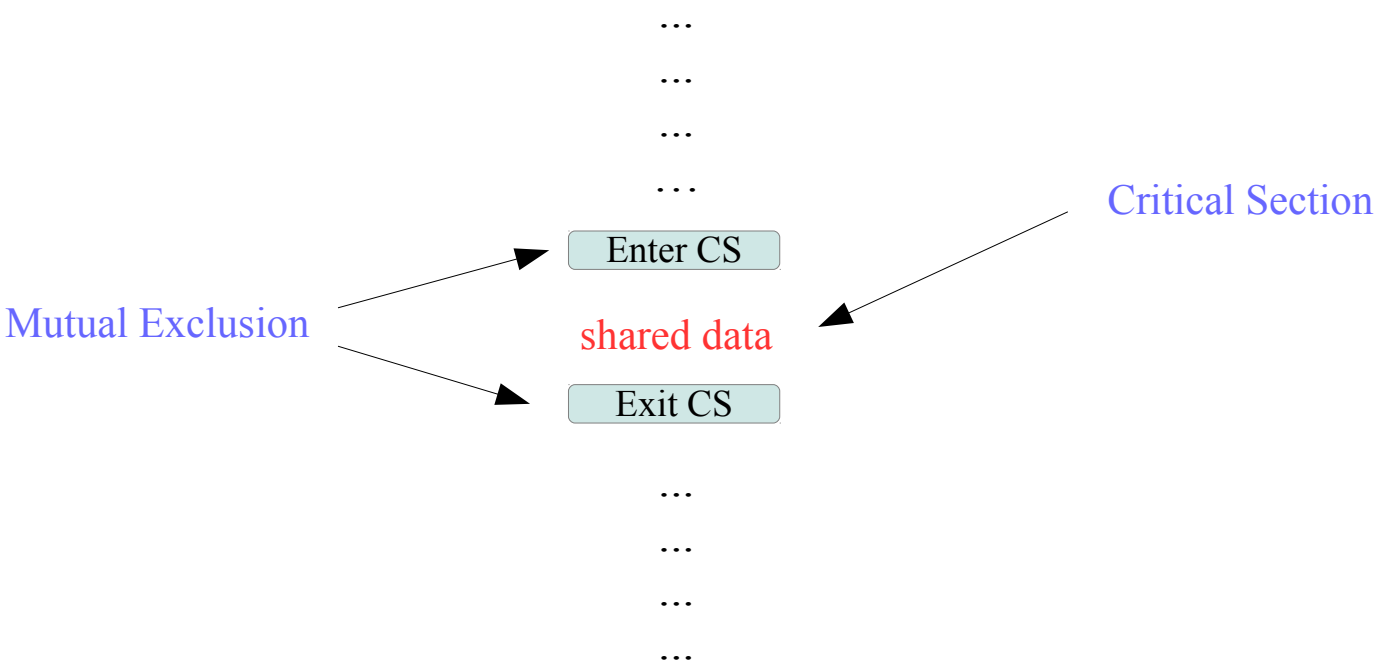
- Need for synchronization when accessing shared memory

```
if ( balance > amount ) {
        balance = balance – amount;
}
```

- will a positive balance be maintained if all threads can execute the above code at the same time ???

Critical Sections

- defining Critical Sections



Critical Sections

- CS advantages:
  - High-level programming → OS API, e.g.

    POSIX Threads (pthread) library

- CS disadvantages:
  - Heavy-weight (high overhead) → suitable for relatively large critical sections

Atomic Instructions

- using Atomic Instructions

  atomic instructions will serialize access to shared data → access by only one thread at any time

- Examples of atomic instructions
  - Intel:
    - xchg %eax, 0x0(%ebx)
    - lock inc 0x0(%eax)
    - lock cmpxchg %ebx, 0x0(%ecx) // implicit operand %eax
  - Alpha, PPC, MIPS, ARM
    - load-link and store-conditional (LL/SC)

Atomic Instructions

- advantages:
  - Light-weight (no OS overhead as in CS) → suitable for simple tasks, e.g. inc/dec counter
- Disadvantages:
  - might need to write assembly code

Memory Ordering

- Memory [consistency] model:
  - Could be described as the relationship between the order of loads/stores in a core and the order in which those loads/stores are made visible globally; i.e. in which order main memory (other cores) sees the loads/stores of a core

Memory Models

- SC:

  all loads/stores by a core should be made globally visible in the same order that occurred in the core, i.e. memory order respects programme order

[1]

- If $L(a) <_p L(b) \Rightarrow L(a) <_m L(b)$ /* Load→Load */
- If $L(a) <_p S(b) \Rightarrow L(a) <_m S(b)$ /* Load→Store */
- If $S(a) <_p S(b) \Rightarrow S(a) <_m S(b)$ /* Store→Store */
- If $S(a) <_p L(b) \Rightarrow S(a) <_m L(b)$ /* Store→Load */

$$L(a) = \text{Value of MAX}_{<_m} \{S(a) \mid S(a) <_m L(a)\},$$

1. Daniel J. Sorin, Mark D. Hill, and David A. Wood, A Primer on Memory Consistency and Cache Coherence

Memory Models

- Total Store Order (TSO)

  employed in x86. Similar to SC, except that “load” after “store” is not in the same order in programme and memory order.

[1]

- If $L(a) <_p L(b) \Rightarrow L(a) <_m L(b)$ /* Load → Load */
- If $L(a) <_p S(b) \Rightarrow L(a) <_m S(b)$ /* Load → Store */
- If $S(a) <_p S(b) \Rightarrow S(a) <_m S(b)$ /* Store → Store */
- ~~If S(a) <p L(b) ==> S(a) <m L(b) /* Store-->Load */~~ /* **Change 1: Enable FIFO Write Buffer** */

~~Value of L(a) = Value of MAX $_{<m}$ {S(a) | S(a) <m L(a)}~~ /* **Change 2: Need Bypassing** */

Value of L(a) = Value of MAX $_{<m}$ {S(a) | S(a) <m L(a) **or S(a) <p L(a)**}

Allows optimizations, such as FIFO write buffer. Strict SC is not always needed.

1. Daniel J. Sorin, Mark D. Hill, and David A. Wood, A Primer on Memory Consistency and Cache Coherence

Memory Models

- TSO can also achieve strict memory ordering similar to SC on demand by enforcing ordering using memory fences (barriers).

  [1]

  - If L(a) <p FENCE ⇒ L(a) <m FENCE /* Load → FENCE */
  - If S(a) <p FENCE ⇒ S(a) <m FENCE /* Store → FENCE */
  - If FENCE <p FENCE ⇒ FENCE <m FENCE /* FENCE → FENCE */
  - If FENCE <p L(a) ⇒ FENCE <m L(a) /* FENCE → Load */
  - If FENCE <p S(a) ⇒ FENCE <m S(a) /* FENCE → Store */

- Fence instructions in x86:
  - lfence
  - sfence
  - mfence

1. Daniel J. Sorin, Mark D. Hill, and David A. Wood, A Primer on Memory Consistency and Cache Coherence

Memory Ordering

- Memory ordering and atomic instructions in Intel architectures

  memory accesses are **not** reordered past atomic instructions → fences are not needed for sync. algorithms using atomic instructions

Dekker's Algorithm

- Back to Critical Sections:

  two approaches:

  - OS API → pthreads
  - Using mutual exclusion (mutex) algorithms, such as Dekker

    advantages:

    - independent of OS – no special OS support needed
    - independent of ISA – no special instruction, e.g. atomics

    disadvantage:

    - works only for two threads (as opposed to Peterson)
    - requires SC memory model

Dekker's Algorithm

```
flag_i ⇐ True
while flag_j do
   if turn ≠ i then
      flag_i ⇐ False
      while turn ≠ i do
         Do nothing or sleep
      end while
      flag_i ⇐ True
   end if
end while

Do critical work

turn ⇐ j
flag_i ⇐ False
```

In this lab

- Implement Dekker's algorithm
- Adding necessary memory barriers to Dekker's algorithm to make it run correctly on x86 machine
- Comparing performance of critical section implementation using pthread and Dekker's algorithm
- Updating shared memory using different atomic instructions
- Performance analysis and comparison of all the above implementations (CS using pthreads, CS using Dekker's algorithm, using atomic instructions to update shared memory)
- All the shared memory test will be done using Algorithm 1

In this lab

Algorithm 1 Test code for two threads sharing data. n is the number of iterations to execute and *thread* is the thread number. We would intuitively expect *shared_data* to be 0 after both threads have executed, which is only the case if the implementation is properly synchronized.

```
Require: shared_data = 0
  for i = 1 to n do
    if thread = 0 then
      shared_data ⇐ shared_data + 1
    else
      shared_data ⇐ shared_data − 1
    end if
  end for
```

In this lab

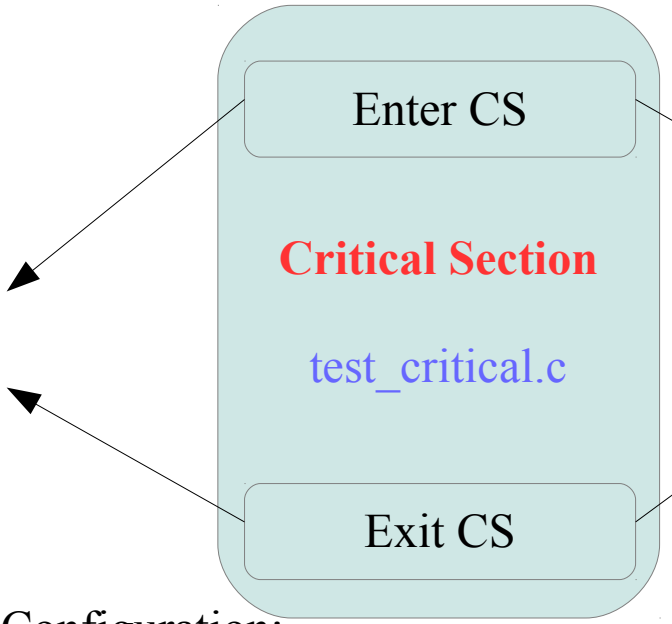
Download lab2 skeleton files from “piazza → course page → resources” and extract them in your home directory.

Lab2 skeleton:

- Makefile

  automates compilation: make clean, make all
- lab2.h & lab2.c: glue code for running the tests
- lab2_asm.h: inline assembler atomic implementations
- cs_pthread.c: reference sync implementation for CS using pthreads
- cs_dekker.c: implement sync for CS using Dekker's algorithm here
- test_critical.c: implements Algorithm 1 using CS
- test_incdec.c: implements Algorithm 1 using atomic inc/dec
- test_cmpxchg.c: implements Algorithm 1 using atomic cmpxchg

In this lab



cs_pthread.c

./lab2 -t critical -c pthreads

cs_dekker.c

./lab2 -t critical -c dekker

Configuration:

Test implementation: critical
Critical sections implementation: dekker
Iterations: 1000000

Statistics:

Thread 0: 0.2927 s (3.4166e+06 iterations/s)
Thread 1: 0.2927 s (3.4169e+06 iterations/s)
Average execution time: 0.2927 s
Avergage iterations/second: 3.4168e+06

NO INCONSISTENCY after 1000000 iterations

In this lab

test_incdec.c

./lab2 -t incdec_no_atomic

./lab2 -t incdec_atomic

test_cmpxchg.c

./lab2 -t cmpxchg_no_atomic

./lab2 -t cmpxchg_atomic

In this lab

./lab2 -c IMPL -t TEST -i ITER

./lab2 -h
Options:

-c IMPL	Use critical section implementation IMPL, use the 'help' IMPL to get a list of available implementations (Default: pthreads)
-t TEST	Use test implementation TEST, use the 'help' TEST to list available implementations. (Default: critical)
-i ITER	Run ITER iterations (Default: 1000000)
-h	Display usage

./lab2 -c help
Critical section implementations:

- **pthreads** - Pthread mutexes
- **null** - NULL implementation that does not enforce mutual exclusion
- **dekker** - Dekker's algorithm
- **queue** - CLH Queue Locks

In this lab

./lab2 -t help

critical - Modify a shared variable protected by critical sections
critical4 - Modify a shared variable protected by critical sections
critical8 - Modify a shared variable protected by critical sections
incdec_no_atomic - Modify a shared variable using inc/dec instructions
incdec_atomic - Modify a shared variable using atomic inc/dec instructions
cmpxchg_atomic - Modify a shared variable using atomic compare and exchange
cmpxchg_no_atomic - Modify a shared variable using compare and exchange

Lab Tasks

- Task 1: CS with pthreads → ./lab2 -t critical -c pthreads. Does the test pass? (look into test_critical.c).
- Task 2: enforcing mutual exclusion using “enter_critical” and “exit_critical” (cs_pthreads.c and cs_dekker.c) in “test_critical.c”
- Task 3: implement Dekker's algorithm (cs_dekker.c)
  - Why should “flag” and “turn” be volatile?
  - Does the test pass?
- Task 4: make Dekker's algorithm work by adding memory barriers (MFENCE) in cs_dekker.c. Use the already defined macro, and notice the gcc intrinsic for fences.
- Task 5: implement Algorithm 1 using “inc” and “dec” (test_incdec.c). Use functions in lab2_asm.h. Test non_atomic and atomic versions, and describe the result.
- Task 6: implement Algorithm 1 using “cmpxchg” (test_cmpxchg.c). Use functions in lab2_asm.h. Test non_atomic and atomic versions, and describe the result.
- Task 7: compare and explain the performance (faster/slower) of CS implementations and atomics.
- Task 8: compare and explain the performance of atomic and non_atomic versions.
- Task 9: Bonus. Implement queue locks “CLH” (cs_queue.c) using atomic instructions in lab2_asm.h. (hint: fences and atomic_xchg). (refer to lecture notes).

Lab Sign-up and Groups

Sign up using doodle: piazza → Course Page → Resources → General Resources

Lab Preparation: 2013-10-08, 08:00 ~ 12:00, room 1412

Group A: 2013-10-09, 13:00 ~ 17:00, room 1412

Group B: 2013-10-10, 13:00 ~ 17:00, room 1412

Group C: 2013-10-11, 13:00 ~ 17:00, room 1412