Synchronization 3 and GPUs

CS 3410, Spring 2014
Computer Science
Cornell University

See P&H Chapter: 6.7
Administrivia

Next 3 weeks

• Prelim2 Thu May 1st: 7:30-9:30
  – Olin 155: Netid [a-g]*
  – Uris G01: Netid [h-z]*
• Proj3 tournament: Mon May 5 5pm-7pm (Pizza!)
• Proj4 design doc meetings May 5-7 (doc ready for mtg)

Final Project for class

• Proj4 due Wed May 14
• Proj4 demos: May 13 and 14
• Proj 4 release: in labs this week
• Remember: No slip days for PA4
Academic Integrity

All submitted work must be your own
  • OK to study together, but do not share soln’s
  • Cite your sources

Project groups submit joint work
  • Same rules apply to projects at the group level
  • Cannot use of someone else’s soln

Closed-book exams, no calculators

• Stressed? Tempted? Lost?
  • Come see us before due date!

Plagiarism in any form will not be tolerated
Synchronization

- Threads
- Critical sections, race conditions, and mutexes
- Atomic Instructions
  - HW support for synchronization
  - Using sync primitives to build concurrency-safe data structures
- Example: thread-safe data structures
- Language level synchronization
- Threads and processes
Synchronization in MIPS

Load linked: \texttt{LL \texttt{rt}, offset(rs)}

Store conditional: \texttt{SC \texttt{rt}, offset(rs)}

- Succeeds if location not changed since the LL
  - Returns 1 in rt
- Fails if location is changed
  - Returns 0 in rt

Any time a processor intervenes and modifies the value in memory between the LL and SC instruction, the SC returns 0 in $t0

Use this value 0 to try again
Mutex from LL and SC

Linked load / Store Conditional

m = 0; // 0 means lock is free; otherwise, if m ==1, then lock locked

mutex_lock(int m) {
    while(test_and_set(&m)){}
}

int test_and_set(int *m) {
    old = *m;
    LL   Atomic
    *m = 1;
    SC
    return old;
}
Mutex from LL and SC

Linked load / Store Conditional

m = 0;
mutex_lock(int *m) {
    while(test_and_set(m)){}
}
int test_and_set(int *m) {
    try:
        LI $t0, 1
        LL $t1, 0($a0)
        SC $t0, 0($a0)
        BEQZ $t0, try
        MOVE $v0, $t1
    }
Mutex from LL and SC

m = 0;
mutex_lock(int *m) {
    test_and_set:
        LI $t0, 1
        LL $t1, 0($a0)
        BNEZ $t1, test_and_set
        SC $t0, 0($a0)
        BEQZ $t0, test_and_set
}
mutex_unlock(int *m) {
    *m = 0;
}
This is called a Spin lock
Aka spin waiting

m = 0;
mutex_lock(int *m) {
    test_and_set:
        LI $t0, 1
        LL $t1, 0($a0)
        BNEZ $t1, test_and_set
        SC $t0, 0($a0)
        BEQZ $t0, test_and_set
}
mutex_unlock(int *m) {
    SW $zero, 0($a0)
}
## Mutex from LL and SC

```c
m = 0;
mutex_lock(int *m) {
```

<table>
<thead>
<tr>
<th>Time Step</th>
<th>Thread A</th>
<th>Thread B</th>
<th>Thread A $t0</th>
<th>Thread A $t1</th>
<th>Thread B $t0</th>
<th>Thread B $t1</th>
<th>Mem M[$a0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>try: LI $t0, 1</td>
<td>try: LI $t0, 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>LL $t1, 0($a0)</td>
<td>LL $t1, 0($a0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>BNEZ $t1, try</td>
<td>BNEZ $t1, try</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>SC $t0, 0($a0)</td>
<td>SC $t0, 0($a0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>BEQZ $t0, try</td>
<td>BEQZ $t0, try</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Mutex from LL and SC

```c
m = 0;
mutex_lock(int *m) {
```

<table>
<thead>
<tr>
<th>Time Step</th>
<th>Thread A</th>
<th>Thread B</th>
<th>Thread A $t0</th>
<th>Thread A $t1</th>
<th>Thread B $t0</th>
<th>Thread B $t1</th>
<th>Mem M[$a0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>try: LI $t0, 1</td>
<td>try: LI $t0, 1</td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>LL $t1, 0($a0)</td>
<td>LL $t1, 0($a0)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>BNEZ $t1, try</td>
<td>BNEZ $t1, try</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>SC $t0, 0($a0)</td>
<td>SC $t0, 0 ($a0)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>BEQZ $t0, try</td>
<td>BEQZ $t0, try</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Mutex from LL and SC

m = 0;
mutex_lock(int *m) {

<table>
<thead>
<tr>
<th>Time Step</th>
<th>Thread A</th>
<th>Thread B</th>
<th>Thread A $t0</th>
<th>Thread A $t1</th>
<th>Thread B $t0</th>
<th>Thread B $t1</th>
<th>Mem M[$a0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>try: LI $t0, 1</td>
<td>try: LI $t0, 1</td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>LL $t1, 0($a0)</td>
<td>LL $t1, 0($a0)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>BNEZ $t1, try</td>
<td>BNEZ $t1, try</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>SC $t0, 0($a0)</td>
<td>SC $t0, 0 ($a0)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>BEQZ $t0, try</td>
<td>BEQZ $t0, try</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>try: LI $t0, 1</td>
<td>Critical section</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summary

Need parallel abstraction like for multicore

Writing correct programs is hard
    Need to prevent data races

Need critical sections to prevent data races
    Mutex, mutual exclusion, implements critical section
    Mutex often implemented using a lock abstraction

Hardware provides synchronization primitives such as LL and SC (load linked and store conditional) instructions to efficiently implement locks
Topics

Synchronization

- Threads
- Critical sections, race conditions, and mutexes
- Atomic Instructions
  - HW support for synchronization
  - Using sync primitives to build concurrency-safe data structures
- Example: thread-safe data structures
- Language level synchronization
- Threads and processes
Next Goal

How do we use synchronization primitives to build concurrency-safe data structure?

Let’s look at a ring buffer
 Attempt#1: Producer/Consumer

Access to shared data must be synchronized
  • goal: enforce datastructure invariants

  // invariant:
  // data is in A[h ... t-1]
  char A[100];
  int h = 0, t = 0;

  // producer: add to list tail
  void put(char c) {
    // Need: check if list full
    A[t] = c;
    t = (t+1)%n;
  }

  // consumer: take from list head
  char get() {
    while (empty()) { };
    char c = A[h];
    h = (h+1)%n;
    return c;
  }
Testing the invariant

Various ways to implement empty()

1. h==t
   
   but then the put has to be careful

2. could be a separate boolean

...
Attempt#1: Producer/Consumer

// invariant:  
// data is in A[h ... t-1]
char A[100];
int h = 0, t = 0;

// producer: add to list tail  
// consumer: take from list head

void put(char c) { ....... char get() {
    A[t] = c;
    t = (t+1)%n;
}

    while (empty()) {
    char c = A[h];
    h = (h+1)%n;
    return c;
}

Error: could miss an update to t or h due to lack of synchronization
Current implementation will break invariant:
    only produce if not full and only consume if not empty

Need to synchronize access to shared data
Attempt#2: Protecting an invariant

// invariant: (protected by mutex m)
// data is in A[h ... t-1]
pthread_mutex_t *m = pthread_mutex_create();
char A[100];
int h = 0, t = 0;

// consumer: take from list head
char get() {
    pthread_mutex_lock(m);
    while(empty()) {}
    char c = A[h];
    h = (h+1)%n;
    pthread_mutex_unlock(m);
    return c;
}

Rule of thumb: all access and updates that can affect invariant become critical sections
Attempt#2: Protecting an invariant

// invariant: (protected by mutex m)
// data is in A[h ... t-1]
pthread_mutex_t *m = pthread_mutex_create();
char A[100];
int h = 0, t = 0;

// consumer: take from list head
char get() {
    pthread_mutex_lock(m);
    while(empty()) {}
    char c = A[h];
    h = (h+1)%n;
    pthread_mutex_unlock(m);
    return c;
}

BUG: Can’t wait while holding lock

Rule of thumb: all access and updates that can affect invariant become critical sections
Guidelines for successful mutexing

Insufficient locking can cause races
  • Skimping on mutexes? Just say no!

But poorly designed locking can cause deadlock

P1: lock(m1);    P2: lock(m2);
    lock(m2);    lock(m1);

Circular Wait

• Know why you are using mutexes!
• Acquire locks in a consistent order to avoid cycles
• Use lock/unlock like braces (match them lexically)
  – lock(&m); ...; unlock(&m)
  – Watch out for return, goto, and function calls!
  – Watch out for exception/error conditions!
Attempt#3: Beyond mutexes

Writers must check for full buffer
& Readers must check if for empty buffer

- ideal: don’t busy wait... go to sleep instead

```c
char get() {
    while (empty()) {
    lock (L);
    char c = A[h];
    h = (h+1)%n;
    unlock (L);
    return c;
}
```

Cannot check condition while Holding the lock,
BUT, empty condition may no longer hold in critical section

Dilemma: Have to check while holding lock
Attempt#3: Beyond mutexes

Writers must check for full buffer & Readers must check if for empty buffer

• ideal: don’t busy wait… go to sleep instead

```c
char get() {
    lock (L);
    while (empty()) {};
    char c = A[h];
    h = (h+1)%n;
    unlock (L);
    return c;
}
```

Dilemma: Have to check while holding lock, but cannot wait while holding lock
Attempt#4: Beyond mutexes

Writers must check for full buffer & Readers must check if for empty buffer

• ideal: don’t busy wait... go to sleep instead

```c
char get() {
    do {
        lock (L);
        if (!empty()) {
            c = A[h];
            h = (h+1)%n;
        }
        unlock (L);
    } while (empty);
    return c;
}
```
Language-Level Synchronization

Condition variables

  Wait for condition to be true
  Thread sleeps while waiting
  Can wake up one thread or all threads

Monitors, ...
Summary

Hardware Primitives: test-and-set, LL/SC, barrier, ...
... used to build ...

Synchronization primitives: mutex, semaphore, ...
... used to build ...

Language Constructs: monitors, signals, ...
PRELIM 2 CONTENT TILL HERE
Abstraction of Processes

How do we cope with lots of activity?

Simplicity? Separation into processes
Reliability? Isolation
Speed? Program-level parallelism
Process and Program

Process
OS abstraction of a running computation
- The unit of execution
- The unit of scheduling
- Execution state + address space

From process perspective
- a virtual CPU
- some virtual memory
- a virtual keyboard, screen, ...

Program
“Blueprint” for a process
- Passive entity (bits on disk)
- Code + static data

<table>
<thead>
<tr>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
</tr>
<tr>
<td>Initialized data</td>
</tr>
<tr>
<td>BSS</td>
</tr>
<tr>
<td>Symbol table</td>
</tr>
<tr>
<td>Line numbers</td>
</tr>
<tr>
<td>Ext. refs</td>
</tr>
</tbody>
</table>
Process and Program

Process:
- mapped segments
- DLL's
- Stack
- Heap
- BSS
- Initialized data
- Code

Program:
- Header
- Code
- Initialized data
- BSS
- Symbol table
- Line numbers
- Ext. refs
Role of the OS

Role of the OS

Context Switching
  • Provides illusion that every process owns a CPU

Virtual Memory
  • Provides illusion that process owns some memory

Device drivers & system calls
  • Provides illusion that process owns a keyboard, ...

To do:
  How to start a process?
  How do processes communicate / coordinate?
How to create a process?

Q: How to create a process?
A: Double click

After boot, OS starts the first process
...which in turn creates other processes
  • parent / child → the process tree
pstree example

```
$ pstree | view -
init+-+NetworkManager+++dhclient
  |-apache2
  |-chrome++-chrome
     `-chrome
  |-chrome---chrome
  |-clementine
  |-clock-applet
  |-cron
  |-cupsd
  |-firefox---run-mozilla.sh---firefox-bin++-plugin-cont
     |-gnome-screensaver
     |-grep
     |-in.tftpd
     |-ntpd
    `sshda---sshda---sshda---bash++-gcc---gcc---cc1
       `-pstree
       `-vim
      ` -view
```
Processes Under UNIX

Init is a special case. For others...
Q: How does parent process create child process?
A: fork() system call

int fork() returns TWICE!
Example

```c
main(int ac, char **av) {
    int x = getpid(); // get current process ID from OS
    char *hi = av[1]; // get greeting from command line
    printf("I’m process %d\n", x);
    int id = fork();
    if (id == 0)
        printf("%s from %d\n", hi, getpid());
    else
        printf("%s from %d, child is %d\n", hi, getpid(), id);
}
$ gcc -o strange strange.c
$ ./strange “Hey”
I’m process 23511
Hey from 23512
Hey from 23511, child is 23512
```
Inter-process Communication

Parent can pass information to child
  • In fact, *all parent data* is passed to child
  • But isolated after (copy-on-write ensures changes are invisible)

Q: How to continue communicating?
A: Invent OS “IPC channels” : send(msg), recv(), ...

Inter-process Communication

Parent can pass information to child

• In fact, *all parent data* is passed to child
• But isolated after (C-O-W ensures changes are invisible)

Q: How to continue communicating?
A: Shared (Virtual) Memory!
Processes and Threads
<table>
<thead>
<tr>
<th>Processes and Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Process</strong></td>
</tr>
<tr>
<td>OS abstraction of a running computation</td>
</tr>
<tr>
<td>• The unit of execution</td>
</tr>
<tr>
<td>• The unit of scheduling</td>
</tr>
<tr>
<td>• Execution state + address space</td>
</tr>
<tr>
<td><strong>From process perspective</strong></td>
</tr>
<tr>
<td>• a virtual CPU</td>
</tr>
<tr>
<td>• some virtual memory</td>
</tr>
</tbody>
</table>
| • a virtual keyboard, screen, ...
| **Thread**            |
| OS abstraction of a single thread of control |
| • The unit of scheduling |
| • Lives in one single process |
| **From thread perspective** |
| • one virtual CPU core on a virtual multi-core machine |
Multithreaded Processes

The diagram illustrates the differences between a single-threaded process and a multithreaded process. In a single-threaded process, there is a single thread that manages the execution of code, data, and files, along with registers and a stack. In contrast, a multithreaded process contains multiple threads, each managing its own set of registers and stacks.

Thread transitions and shared resources are highlighted to demonstrate the complex interactions in a multithreaded environment.
Threads

#include <pthread.h>
int counter = 0;

void PrintHello(int arg) {
    printf("I'm thread %d, counter is %d\n", arg, counter);
    ... do some work ...
    pthread_exit(NULL);
}

int main () {
    for (t = 0; t < 4; t++) {
        printf("in main: creating thread %d\n", t);
        pthread_create(NULL, NULL, PrintHello, t);
    }
    pthread_exit(NULL);
}
Threads versus Fork

in main: creating thread 0
I’m thread 0, counter is 0
in main: creating thread 1
I’m thread 1, counter is 1
in main: creating thread 2
in main: creating thread 3
I’m thread 3, counter is 2
I’m thread 2, counter is 3
Processes and Threads are the abstraction that we use to write parallel programs.

Fork and Join and Interprocesses communication (IPC) can be used to coordinate processes.

Threads are used to coordinate use of shared memory within a process.
GPUs
The supercomputer in your laptop

GPU: Graphics processing unit

Very basic till about 1999
Specialized device to accelerate display
Then started changing into a full processor

2000-...: Frontier times
Parallelism

CPU: Central Processing Unit
GPU: Graphics Processing Unit
GPU-type computation offers higher GFlops

(Source: Sam Naffziger, AMD)
GPUs: Faster than Moore’s Law
Moore’s Law is for Wimps?!

One-pixel polygons (~10M polygons @ 30Hz)

Slope ~2.4x/year
(Moore’s Law ~ 1.7x/year)

Graph courtesy of Professor John Poulton (from Eric Haines)
Programmable Hardware

• Started in 1999
• Flexible, programmable
  – Vertex, Geometry, Fragment Shaders
• And much faster, of course
  • 1999 GeForce256: 0.35 Gigapixel peak fill rate
  • 2001 GeForce3: 0.8 Gigapixel peak fill rate
  • 2003 GeForceFX Ultra: 2.0 Gigapixel peak fill rate
  • ATI Radeon 9800 Pro: 3.0 Gigapixel peak fill rate
  • 2006 NV60: ... Gigapixel peak fill rate
  • 2009 GeForce GTX 285: 10 Gigapixel peak fill rate
  • 2011
    – GeForce GTC 590: 56 Gigapixel peak fill rate
    – Radeon HD 6990: 2x26.5
  • 2012
    – GeForce GTC 690: 62 Gigapixel/s peak fill rate
Evolution of GPU

- DirectX 5: Riva 128
- DirectX 6: Multi-texturing, Riva TNT
- DirectX 7: T&L, Texture Stage, State
- DirectX 8: SM 1.x, GeForce 3
- DirectX 9: SM 2.0, GeForce FX
- DirectX 9.0c: SM 3.0, GeForce 6
- DirectX 10: OpenGL 4.0
- DirectX 10.1: OpenGL 4.2
- DirectX 11
- DirectX 11.1

Games:
- Quake 3
- Giants
- Halo
- Far Cry
- UE3
- BattleForge
Around 2000

Fixed function pipeline
Around 2005
Programmable vertex and pixel processors
Post 2006: Unified Architecture
Why?

- Parallelism: thousands of cores
- Pipelining
- Hardware multithreading
- Not multiscale caching
  - Streaming caches
- Throughput, not latency
## Flynn’s Taxonomy

<table>
<thead>
<tr>
<th>Single Instruction</th>
<th>Multiple Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Data</td>
<td>Single Data</td>
</tr>
<tr>
<td>(SISD)</td>
<td>(MISD)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Single Instruction</td>
<td>Multiple Instruction</td>
</tr>
<tr>
<td>Multiple Data</td>
<td>Multiple Data</td>
</tr>
<tr>
<td>(SIMD)</td>
<td>(MIMD)</td>
</tr>
</tbody>
</table>
MIMD array of SIMD procs
Grids, Blocks, and Threads

Shuang Zhao, Cornell University, 2014
CUDA Memory

- Grid
  - Block (0, 0)
    - Shared Memory
    - Registers
    - Thread (0, 0)
    - Thread (1, 0)
  - Block (1, 0)
    - Shared Memory
    - Registers
    - Thread (0, 0)
    - Thread (1, 0)

- Host
- Global Memory
- Constant Memory

Faster, per-block
Fastest, per-thread
Slower, global
Read-only, cached
Heterogeneous Computing

Host: the CPU and its memory

Device: the GPU and its memory

Shuang Zhao, Cornell University, 2014
Programming using CUDA

Compute Unified Device Architecture

do_something_on_host();
kernel<<<nBlk, nTid>>>(args);
cudaDeviceSynchronize();
do_something_else_on_host();

Highly parallel

Shuang Zhao, Cornell University, 2014
Hardware Thread Organization

Threads in a block are partitioned into warps

- All threads in a warp execute in a Single Instruction Multiple Data, or SIMD, fashion
- All paths of conditional branches will be taken
- Warp size varies, many graphics cards have 32

NO guaranteed execution ordering between warps
Branch Divergence

Threads in one warp execute very different branches
Significantly harms the performance!

Simple solution:

• Reordering the threads so that all threads in each block are more likely to take the same branch
• Not always possible

Shuang Zhao, Cornell University, 2014
single-threaded free lunch

Welcome to the jungle

cloud-core

The free lunch is so over

multi-core

hetero-core

20??
Exit Moore