Announcements

MP4 due tonight

Midterm
  • Next Tuesday, 7-9 p.m.
  • Study guide and practice exam released Wednesday

PPT?
Do threads conflict in practice?

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

#define NUM_THREADS 2
#define ITERATIONS_PER_THREAD 5000000

int cnt = 0;

void * worker( void *ptr )
{
    int i;
    for (i = 0; i < ITERATIONS_PER_THREAD; i++)
        cnt++;
}
```
Do threads conflict in practice?

```c
int main(void)
{
    pthread_t threads[NUM_THREADS];
    int i, result;

    /* Start threads */
    for (i = 0; i < NUM_THREADS; i++) {
        result = pthread_create(&threads[i], NULL, worker, NULL);
        assert(result == 0);
    }

    /* Wait for threads to finish */
    for (i = 0; i < NUM_THREADS; i++) {
        result = pthread_join(threads[i], NULL);
        assert(result == 0);
    }

    printf("Final value: %d (%.2f%%)\n", cnt,
           100.0 * cnt / (NUM_THREADS * (double)ITERATIONS_PER_THREAD));
}
```
Do threads conflict in practice?

If everything worked...

```bash
$ ./20-counter
Final value: 100000
```

Q: What are the minimum and maximum final value?

Q: What value do you expect in practice?
Assembly Code for Counter Loop

C code for counter loop for thread i

```c
for (i=0; i < 50000; i++)
cnt++;
```

Corresponding assembly code

```
movl (%rdi),%ecx
movl $0,%edx
cmpl %ecx,%edx
jge .L13

.L11:
movl cnt(%rip),%eax
incl %eax
movl %eax,cnt(%rip)
incl %edx
cmpl %ecx,%edx
jl .L11

.L13:
```

Head (H_i)

Load cnt (L_i)
Update cnt (U_i)
Store cnt (S_i)

Critical section: reading or writing shared variable

Tail (T_i)
**Key idea:** In general, any sequentially consistent interleaving is possible, but some give an unexpected result!

- $I_i$ denotes that thread $i$ executes instruction $I$
- $%eax_i$ is the content of $%eax$ in thread $i$’s context

### Table

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>$%eax_1$</th>
<th>$%eax_2$</th>
<th>cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>L</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>U</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>S</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>H</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>L</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>U</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>S</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>T</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>T</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>2</td>
</tr>
</tbody>
</table>

**OK!**
Concurrent execution (example 2)

Incorrect ordering: two threads increment the counter, but the result is 1 instead of 2

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>%eax₁</th>
<th>%eax₂</th>
<th>cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td></td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>L</td>
<td></td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>U</td>
<td></td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>H</td>
<td></td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>L</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>T</td>
<td></td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>U</td>
<td></td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>S</td>
<td></td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>T</td>
<td></td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
</tbody>
</table>

Thread 1 critical section
Thread 2 critical section

Oops!
Progress Graphs

A progress graph depicts the discrete execution state space of concurrent threads.

Each axis corresponds to the sequential order of instructions in a thread.

Each point corresponds to a possible execution state (Inst₁, Inst₂).

E.g., (L₁, S₂) denotes state where:
thread 1 has completed L₁ and thread 2 has completed S₂.
A trajectory is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:
H₁, L₁, U₁, H₂, L₂, S₁, T₁, U₂, S₂, T₂
Critical Sections and Unsafe Regions

L, U, and S form a critical section with respect to the shared variable cnt.

Instructions in critical sections (wrt to some shared variable) should not be interleaved.

Sets of states where such interleaving occurs form unsafe regions.
Critical Sections and Unsafe Regions

A trajectory is \textit{safe} if and only if it does not enter any unsafe region.

**Claim:** A trajectory is correct (w.r.t. variable \textit{cnt}) iff it is safe.
Enforcing mutual exclusion

How can we guarantee a safe trajectory?

Answer: We must synchronize the execution of the threads so that they never have an unsafe trajectory.
  • i.e., need to guarantee mutually exclusive access to critical regions
  • provides a sufficient condition for correctness

Classic solution
  • Semaphores (Edsger Dijkstra) (pthreads)

Other approaches
  • Mutexes, and condition variables (pthreads)
  • Locks and rwlocks (pthreads)
  • Monitors (Java)
Semaphores

photo: Les Meloures / wikimedia
Semaphores

A non-negative global integer synchronization variable

Manipulated by \textit{wait} and \textit{post} operations:

- \textit{wait}(s): [ \textbf{while} (s == 0) \textbf{wait}(); s--; ]
  - Also \textit{P}(s), Dutch for "Proberen" (test)
- \textit{post}(s): [ s++; ]
  - Also \textit{V}(s), Dutch for "Verhogen" (increment)

OS kernel guarantees that operations between brackets [ ] are executed indivisibly

- i.e., s-- can’t be broken into load/update/store
- Result: only one \textit{wait} or \textit{post} operation at a time can modify s
- When \textbf{while} loop in \textit{wait} terminates, only that \textit{wait} can decrement s

Semaphore invariant: (s >= 0)
C Semaphore Operations

pthread functions:

```c
#include <semaphore.h>

int sem_init(sem_t *sem, 0, unsigned int val);} /* s = val */

int sem_wait(sem_t *s);
int sem_post(sem_t *s);
```
Back to the counter...

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

#define NUM_THREADS 2
#define ITERATIONS_PER_THREAD 50000

int cnt = 0;

void * worker( void *ptr )
{
    int i;
    for (i = 0; i < ITERATIONS_PER_THREAD; i++)
        cnt++;
}
```

How can we fix this using semaphores?
Semaphores for mutual exclusion

Basic idea

• Associate a unique semaphore \textit{mutex}, initially 1, with each shared variable (or related set of shared variables)
• Surround corresponding critical sections with \textit{wait}(mutex) and \textit{post}(mutex) operations.

Terminology

• **Binary semaphore**: semaphore whose value is always 0 or 1
• **Mutex**: binary semaphore used for mutual exclusion
  ▪ \textit{wait} operation: “locking” the mutex
  ▪ \textit{post} operation: “unlocking” or “releasing” the mutex
  ▪ “Holding” a mutex: locked and not yet unlocked
• **Counting semaphore**: used to count a set of available resources
goodcounter.c: good synchronization

#include <semaphore.h>  
...

int cnt = 0;
sem_t cnt_mutex;

int main(void)
{
    ...
    /* Initialize mutex */
    sem_init(&cnt_mutex, 0, 1);
    ...
}

void * worker( void *ptr )
{
    int i;
    for (i = 0; i < ITERATIONS_PER_THREAD; i++) {
        sem_wait(&cnt_mutex);
        cnt++;
        sem_post(&cnt_mutex);
    }
}
Why Mutexes Work

Provide mutually exclusive access to shared variable by surrounding critical section with *wait* and *post* operations on semaphore $s$ (initially set to 1)

Semaphore invariant creates a **forbidden region** that encloses the unsafe region that must not be entered by any trajectory.
Discussion

Mutual exclusion changes scheduling between threads
  • Previously: Schedule could be anything
  • With mutual exclusion: Schedule is constrained

Q: Since scheduling is constrained, which thread goes first, Thread 1 or Thread 2?

A: We still have no clue
  • mutex only ensures two threads aren’t in critical section at one time
  • otherwise scheduling is still arbitrary
  • and that’s fine with us
```c
int main(void)
{
    ...
    /* Initialize mutex */
    result = sem_init(&cnt_mutex, 0, 1);
    if (result < 0)
        exit(-1);

    ...

    /* Clean up the semaphore that we're done with */
    result = sem_destroy(&cnt_mutex);
    assert(result == 0);
}
```

Better synchronization!

Check for errors on each call

Clean up resources
Why bother checking for errors?

Without error handling, your code might:

• Crash rather than exiting gracefully
• Keep working for a while, crash later
• Sometimes fail randomly, but usually work fine
  ▪ Hard to reproduce: even harder to debug
• Fail when it might have recovered from the error cleanly!

At a minimum, error handling converts a messy failure into a clean failure

• Program terminates, but you know what caused it to terminate
Some errors are recoverable

```c
void * worker( void *ptr )
{
    int i;
    for (i = 0; i < ITERATIONS_PER_THREAD; i++) {
        while (sem_wait(&cnt_mutex) < 0)
            if (errno != EINTR)
                exit(-1);
        cnt++;
        if (sem_post(&cnt_mutex) < 0)
            exit(-1);
    }
}
```
Much more in the Director’s Cut

Options
• Named semaphores
• Semaphores shared between processes

Other functions / variants
• sem_trywait
• sem_timedwait
• semctl

Other mutual exclusion functions
• pthread_mutex_init
• PTHREAD_MUTEX_INITIALIZER
• pthread_mutex_lock / trylock / unlock
• pthread_mutex_destroy
• ...

Summary

Programmers need a clear model of how variables are shared by threads
  • Cannot reason about all possible interleavings of threads

Variables shared by multiple threads must be protected to ensure mutually exclusive access

Semaphores are a fundamental mechanism for enforcing mutual exclusion
Summary

This cat did not check for exceptional cases

This cat did.