Cooperation requires Synchronization

Example:
Two threads share account balance in memory
Each runs common code, deposit()
void deposit (int amount) {
    balance = balance + amount;
}
Compile to sequence of assembly instructions
load R1, balance
add R1, amount
store R1, balance
Which variables are shared? Which private?

Definitions
Race condition: Result depends upon ordering of execution
  • Non-deterministic bug, very difficult to find
Critical section: Required Properties
  • Mutual exclusion
    - Only one thread in critical section at a time
  • Progress (deadlock-free)
    - If several simultaneous requests, must allow one to proceed
    - Must not depend on threads outside critical section
  • Bounded (starvation-free)
    - Must eventually allow each waiting thread to enter
Desirable Properties
  • Efficient
    - Don’t consume substantial resources while waiting
    - Do not busy wait (i.e., spin wait)
  • Fair
    - Don’t make some processes wait longer than others

Concurrent Execution
What happens if 2 threads deposit concurrently?
Assume any interleaving of instructions is possible
Make no assumptions about scheduler
Initial balance: $100
Thread 1:deposit(10)
    Load R1, balance
    Add R1, amount
    Store R1, balance
Thread 2:deposit(20)
    Load R1, balance
    Add R1, amount
    Store R1, balance
What is the final balance?

Synchronization
Questions answered in this lecture:
Why is synchronization necessary?
What are race conditions, critical sections, and atomic operations?
How to protect critical sections with atomic loads and stores?
Implementing Critical Sections

To implement, need atomic operations

Atomic operation: No other instructions can be interleaved

Examples of atomic operations
- Loads and stores of words — Today's topic
  - Load r1, B
  - Store r1, A
- Code between interrupts on uniprocessors
  - Disable timer interrupts, don't do any I/O
- Special hw instructions
  - Test&Set
  - Compare&Swap

Critical Section: Attempt #1

Code uses a single shared lock variable

Boolean lock = false; // shared variable
Void deposit(int amount) {
  while (lock) /* wait */ ;
  lock = true;
  balance += amount; // critical section
  lock = false;
}

Why doesn't this work? Which principle is violated?

Attempt #2

Each thread has its own lock; lock indexed by tid (0, 1)

Boolean lock[2] = {false, false}; // shared
Void deposit(int amount) {
  lock[tid] = true;
  while (lock[(1-tid)]) /* wait */ ;
  balance += amount; // critical section
  lock[tid] = false;
}

Why doesn't this work? Which principle is violated?

Attempt #3

Turn variable determines which thread can enter

Int turn = 0; // shared
Void deposit(int amount) {
  while (turn == 1-tid) /* wait */ ;
  balance += amount; // critical section
  turn = 1-tid;
}

Why doesn't this work? Which principle is violated?
Peterson’s Algorithm: Solution for Two Threads

Combine approaches 2 and 3: Separate locks and turn variable
Int turn = 0; // shared
Boolean lock[2] = {false, false};
Void deposit(int amount) {
    lock[tid] = true;
    turn = 1-tid;
    while (lock[1-tid] && turn == 1-tid) /* wait */;
    balance += amount; // critical section
    lock[tid] = false;
}

Peterson’s Algorithm: Intuition

Mutual exclusion: Enter critical section if and only if
• Other thread does not want to enter
• Other thread wants to enter, but your turn
Progress: Both threads cannot wait forever at while() loop
• Completes if other process does not want to enter
• Other process (matching turn) will eventually finish
Bounded waiting
• Each process waits at most one critical section

Lamport’s Bakery Algorithm for N Threads

Bakery algorithm intuition
Each thread picks next highest ticket (may have ties)
Enter critical section when have lowest ticket
Choosing[tid] = true;
Number[tid] = Max(number[0]..number[n-1]) + 1;
Choosing[tid] = false;
For (j = 0; j < n; j++) {
    while (choosing[j]);
    while (number[j] && ((number[j],j) < (number[tid],tid)));}
Balance += amount;
Number[tid] = 0;