Lecture 3: 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 only atomic loads and stores
  - How to implement locks?
- Readings for this topic
  - Silberschatz/Galvin: Chapter 6

Independent vs. Cooperating Processes

- Independent process: Not affected by rest of universe
  - Properties
    - No shared state between process (memory, filesystem)
    - Deterministic: Input alone determines results
    - Stop and restart without adverse effects
  - Example: Isolated program that calculates factorial sequence
- Cooperative: Non-independent (not necessarily “cooperative”)
  - Properties
    - Some shared state with other processes
    - Non-deterministic: May have different results each time
    - May be irreproducible: Difficult to debug and test
    - Timing affects results
  - Example: Two programs sharing single terminal
- Deal with cooperative processes in this course

Why support Cooperative Processes

- Share resources
  - Many users share one computer: Increase throughput and utilization
  - Many users share same data
- Construct system in modular fashion
  - Each process performs simple functions
  - Unix example: Connect output and input with pipes
    ps -edaf | grep "java" | wc
- Do things faster
  - Slow device: One process waits for device while other computes
  - Defer work: One process performs non-critical work when idle
  - Parallelism: Divide into smaller jobs and execute on multiprocessor

Cooperating Requires Synchronization

- Two processes share account balance
  void deposit (int amount) {
    balance += amount;
  }
  balance: $100
- Implemented as sequence of assembly instructions
  load R1, balance
  add R1, amount
  store R1, balance
- What happens if two processes make deposit at same time?
  Process 0: deposit (10)  Process 1: deposit (20)
  load R1, balance
  add R1, amount
  store R1, balance
- What is the final balance?
- Race condition: Result depends on ordering of interleaving
Avoiding Race Conditions

- Critical section: Only one process can execute at a time

  ```
  void deposit (int amount) {
      balance += amount;
  }
  ```
  balance: $100

- To implement critical sections, need atomic operations
  - Atomic operations: No other instructions can be interleaved

  Examples of atomic operations
  - Loads and stores of words
    ```
    load r1, B
    store r1, A
    ```
  - Code between interrupts on uniprocessors
    Enable and disable timer interrupts
  - Special instructions
    ```
    Test&Set
    Compare&Swap instruction
    ```

Critical Sections

- Requirements
  - Mutual Exclusion: Only one process in critical section at a time
  - Progress (Deadlock-free)
    If several simultaneous requests, must allow one process to proceed
    Must not depend on processes outside critical section
  - Bounded (Starvation-free)
    Must eventually allow waiting process to proceed

- Desirable Properties
  - Efficient: Don’t consume substantial resources while waiting
    Don’t busy wait (spin wait)
    Better to relinquish processor and let another process run
  - Fair: Don’t make some processes wait longer than others
  - Simple: Should be easy for programmer to reason about and use

Critical Section: Attempt #1

- Previous example with no mutual exclusion

  ```
  void deposit (int amount) {
      balance += amount;
  }
  ```
  balance: $100

- Prevent process from entering when one already executing

  ```
  boolean lock = false;
  void deposit (int amount) {
      while (lock) /* wait */;
      lock = true;
      balance += amount;
      lock = false;
  }
  ```

- Why doesn’t this work?

Critical Section: Attempt #2

- Swap order of setting lock and checking lock

  ```
  boolean lock[] = {false, false};
  void deposit (int amount) {
      lock[pid] = true;
      while (lock[1-pid]) /* wait */;
      lock[pid] = false;
      balance += amount;
      lock[pid] = false;
  }
  ```

- Why doesn’t this work?
Critical Section: Attempt #3

- Simple test determines which process can enter
  
  ```java
  void deposit (int amount) {
      while (turn == 1-pid) /* wait */;
      balance += amount;
      turn = 1-pid;
  }
  ```

- Why doesn't this work?

Critical Section: Solution

- Combine approaches: Separate locks and turn variable
  
  ```java
  boolean lock[] = {false, false};
  void deposit (int amount) {
      lock[pid] = true;
      turn = 1 - pid;
      while (lock[1-pid] && turn == 1 - pid) /* wait */;
      balance += amount;
      lock[pid] = false;
  }
  ```

- Mutual exclusion: Enter critical section if and only if
  - Other process does not want to enter
  - Other process wants to enter, but your turn

- Progress: All processes cannot wait forever at while() loop
  - Complete if other process does not want to enter
  - One process (matching turn) will complete

- Bounded Waiting: Process waits at most one critical section

Synchronization Layering

- Build higher-level synchronization operations in OS
  - Operations that ensure correct ordering of instructions in processes

  - Build them once and get them right

Locks

- `lock.acquire()`
  - Acquire exclusive access to lock
  - Wait if lock is not available

- `lock.release()`
  - Release exclusive access to lock

- Modify previous example to use locks

  ```java
  Lock lock = new Lock();
  lock.acquire();
  balance += amount;
  lock.release();
  ```
**Implementing Locks: Version 1**

- Build locks from atomic loads and stores
  - Use previous solution
    ```java
class Lock {
    private boolean lock[] = {false, false};
    private int turn = 0;
    public void acquire() {
        lock[pid] = true;
        turn = 1 - pid;
        while (lock[1-pid] && turn == 1 - pid) /* wait */;
    }
    public void release() {
        lock[pid] = false;
    }
}
```
- Disadvantages?

**Implementing Locks: Version 2**

- Turn off interrupts for critical section
  - Prevent dispatcher from running another process
  - Code executes atomically
    ```java
    lock.acquire () <--> disableInterrupts();
    lock.release () <--> enableInterrupts();
    ``
- Disadvantages?
  - OS should carefully manage when interrupts are disabled

**Implementing Locks: Version 3**

- Problem with Attempt 1: Testing of lock and setting not atomic
  ```java
  boolean lock = false;
  while (lock) /* wait */;
  lock = true;
  ```
- Special instruction: TestAndSet address
  - Returns previous value of address
  - Sets value at address
- New solution
  ```java
  private boolean lock = false; // true if process in CS
  public void acquire() {
      while (TestAndSet(lock, true));
  }
  public void release() {
      lock = false;
  }
  ```
- Disadvantages?

**Spin-Wait or Block (Sleep)**

- Uniprocessor
  - Waiting process is scheduled --> process holding lock isn't
    - Conclusion: Waiting process should relinquish processor
  - Block: Associate queue with each lock --> Helps fairness
- Multiprocessor
  - Waiting process is scheduled --> ???
  - Correct action depends on how long before lock is released
    - Lock released “quickly” --> Spin-Wait
    - Lock released “slowly” --> Block
    - “Quick” or “Slow” is relative to context-switch cost
- Theoretical result: Two-Phase Waiting
  - Spin for length of context-switch
  - If lock not available, then block
  - Performance always within factor of two of optimal
Implementing Locks: Version 4

- Add process to list when cannot obtain lock

```java
class Lock {
    private boolean lock = false; // true if process in CS
    private boolean guard = false;
    private queue q = new queue();

    public void acquire() {
        while (TestAndSet(guard, true));
        if (lock == true) {
            Add self to q;
            guard = false;
            Call dispatcher;
        } else {
            lock = true;
            guard = false;
        }
    }

    public void release() {
        while (TestAndSet(guard, true));
        if (q == empty) lock = false;
        else Remove and wake first process on q;
    }
}
```