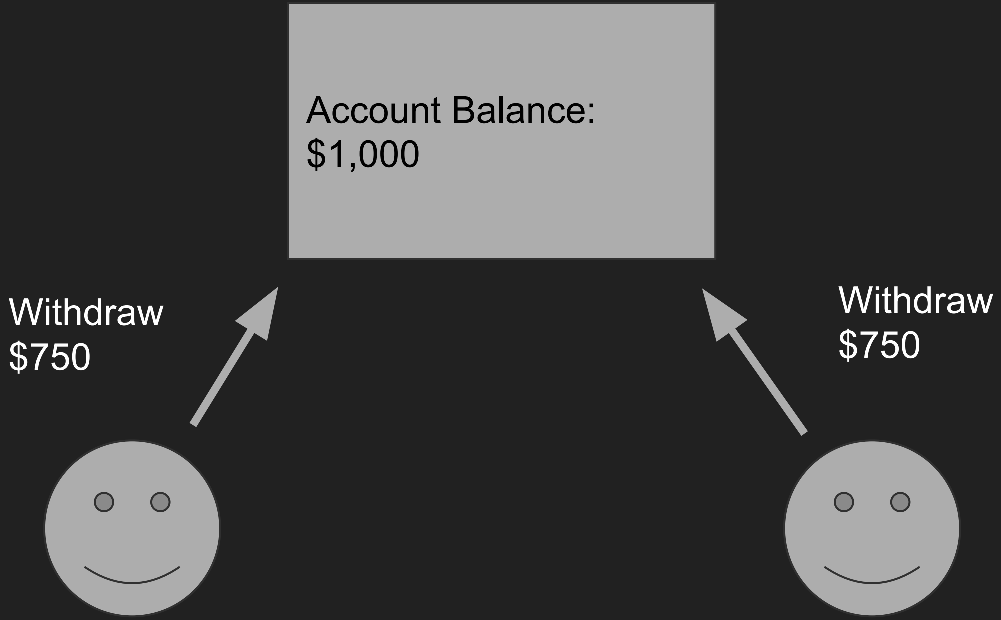
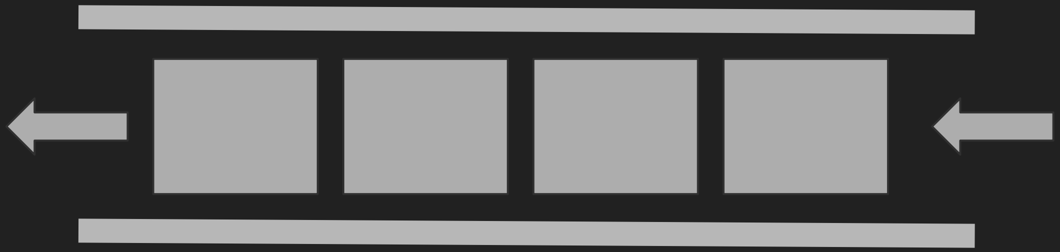
Concurrency

Michael Whitmeyer and Milin Kodnongbua

Concurrency Problem



Concurrency Problem



Solutions

- Message Passing
  - Each thread has their own copy of data and use messages to synchronize changes.
- Shared Memory
  - One copy of shared data. Only one thread is allowed to modify it at a time.
  - Several Ways:
    - No preemption: thread runs through completion without interleaving
    - Atomic transactions
    - locks/semaphores
    - Monitors

Main Goal: Local Concurrent Programming

- Concurrency between lightweight processes (today's threads) within the same application.
- Implement concurrent support in Mesa language using **monitors**.

Producer-consumer: Naive

```
class ThreadSafeQueue {
  Queue q;
  Lock lock;

  void PushRequest(Request rst) {
    lock.Acquire();
    q.Push(rst);
    lock.Release();
  }

  Request GetRequest() {
    lock.Acquire();
    while (q.Empty()) {
      lock.Release();
      sleep(1);
      lock.Acquire();
    }
    Request rst = q.Pop();
    lock.Release();
    return rst;
  }
};
```

Producer-consumer: with Monitor

```
class ThreadSafeQueue {
  Queue q;
  Lock lock;

  void PushRequest(Request rst) {
    lock.Acquire();
    q.Push(rst);
    lock.Release();
  }

  Request GetRequest() {
    lock.Acquire();
    while (q.Empty()) {
      lock.Release();
      sleep(1);
      lock.Acquire();
    }
    Request rst = q.Pop();
    lock.Release();
    return rst;
  }
};
```

```
monitor ThreadSafeQueue {
  Queue q;
  ConditionVariable qChanged;

  entry void PushRequest(Request rst) {

    q.Push(rst);
    qChanged.Notify();
  }

  entry Request GetRequest() {

    while (q.Empty()) {
      qChanged.Wait();


    }
    Request rst = q.Pop();

    return rst;
  }
};
```

Monitors

- Similar to thread-safe classes in Java
- Language construct that contains
  - Synchronization (a lock and condition variables)
  - Shared data,
  - Methods that perform accesses
- Three types of procedure
  - **entry**: acquires and release lock, public
  - **internal**: no locking, private
  - **external**: no locking, public

```
monitor ThreadSafeQueue {
  Queue q;
  ConditionVariable qChanged;

  entry void PushRequest(Request rst) {

    q.Push(rst);
    qChanged.Notify();
  }

  entry Request GetRequest() {

    while (q.Empty()) {
      qChanged.Wait();


    }
    Request rst = q.Pop();

    return rst;
  }
};
```

Condition Variables

- Variables that can communicate status between threads.
- wait() blocks the execution until someone calls notify()
- Helps programmers think about conditions that need to be met before proceeding.

```
monitor ThreadSafeQueue {
  Queue q;
  ConditionVariable qChanged;

  entry void PushRequest(Request rst) {

    q.Push(rst);
    qChanged.Notify();
  }

  entry Request GetRequest() {

    while (q.Empty()) {
      qChanged.Wait();


    }
    Request rst = q.Pop();

    return rst;
  }
};
```

Implementation

A thread can only belong to one of the four states at a time

- Ready
- Monitor Lock: wait to acquire a lock on a monitor
- Condition Variable: wait for a condition variable to be notified
- Fault: unable to run (e.g., exceptions or errors)

These can be implemented using queues for each state. Each monitor and condition variable will have its own queue.

Discussion Question 1

Do we actually need the “while” here?

Could it be replaced with an “if”?

```
monitor ThreadSafeQueue {
  Queue q;
  ConditionVariable qChanged;

  entry void PushRequest(Request rst) {

    q.Push(rst);
    qChanged.Notify();
  }

  entry Request GetRequest() {

    while (q.Empty()) {
      qChanged.Wait();


    }
    Request rst = q.Pop();

    return rst;
  }
};
```

Discussion Question 2

What’s the problem with this code?

```
monitor A {
  entry void Foo(B b) {
    b.Run(*this);
  }
  entry void Bar() {

  }
};


monitor B {
  entry void Run(A a) {
    a.Bar();
  }
};
```

Discussion 3: Priority Inversion

- Suppose **H** (high priority) and **L** (low priority) share resource **R**.
  - Good design ⇒ **L** doesn't hold **R** too long
- But suppose **M** (medium priority) becomes runnable
  - **M** holds **R** ⇒ "priority inversion"
- Solutions?
  - Only **two priorities**: "preemptible" (L) and "interrupts disabled" (H).
    - No deadlocks or priority inversions possible
  - "**Priority ceiling**": If **M** tries to preempt **L** then **L**'s priority gets bumped up to **H**'s.
  - "**Priority inheritance**": If **H** is waiting on **L** then **L** automatically takes **H**'s priority.
  - "**Random Boosting**": ready tasks holding locks are randomly boosted in priority.
    - Windows uses this!

Discussion from Ed

1. Compare and contrast Mesa monitors with a different concurrent programming paradigm. For example, do they have equivalent expressive power (can one be implemented on top of the other)?

- Message passing potentially simpler to analyze/implement
- Some languages use both shared memory and message passing

Discussion from Ed

2. At what level(s) of computer system architecture (e.g., hardware, OS, programming language, user library, etc.) do you think concurrency control mechanisms should be implemented, and why?

- Hardware support is required;
- OS can help with IPC;
    - OS ⇒ processes, threads
- Programing language can help programmers write safer code
    - PLs ⇒ channels (message passing), “promises/futures” (proxy for currently unknown variable)

Discussion from Ed

3. Give an example of a design decision from Mesa/Pilot that was not adopted in modern languages/OSes, and why you think this choice was made differently in later systems.

- Having no way to stop runaway processes
- Having `entry` as part of language construct.
    - Most modern languages provide building blocks for concurrent control but they don’t have a fixed pattern since the use case can vary.

Discussion from Ed

4. Does it make sense to consider monitors for concurrency control in a distributed system, rather than a shared-memory multithreaded environment? If so, how and why? If not, why not?

- Memory an issue. Spread across multiple machines?
- Multiple threads waiting on same lock

Discussion from Ed

5. Do you think any of the lessons, tradeoffs, or tensions of monitors described in paper change on modern multicore machines (e.g., imagine a 100-way multicore)?

- Memory bandwidth will be a bottleneck using a shared memory paradigm.
- Most processes will end up waiting for the lock.