Programming Language Support for Concurrency

Madhavan Mukund, S P Suresh

Programming Language Concepts

Lecture 14, 2 March 2023

Race conditions

- Concurrent update of a shared variable can lead to data inconsistency
  - Race condition
- Control behaviour of threads to regulate concurrent updates
  - Critical sections — sections of code where shared variables are updated
  - Mutual exclusion — at most one thread at a time can be in a critical section

  "lock" the code
- We can construct protocols that guarantee mutual exclusion to critical sections
  - Watch out for starvation and deadlock
- These protocols cleverly use regular variables
  - No assumptions about initial values, atomicity of updates
- Difficult to generalize such protocols to arbitrary situations
- Look to programming language for features that control synchronization

Test and set

- The fundamental issue preventing consistent concurrent updates of shared varuables is `test-and-set`

Test and set

- The fundamental issue preventing consistent concurrent updates of shared varuables is `test-and-set`
- To increment a counter, check its current value, then add 1

Test and set

- The fundamental issue preventing consistent concurrent updates of shared varuables is `test-and-set`
- To increment a counter, check its current value, then add 1
- If more than one thread does this in parallel, updates may overlap and get lost

Test and set

- The fundamental issue preventing consistent concurrent updates of shared varuables is `test-and-set`
- To increment a counter, check its current value, then add 1
- If more than one thread does this in parallel, updates may overlap and get lost
- Need to combine test and set into an atomic, indivisible step

Test and set

- The fundamental issue preventing consistent concurrent updates of shared varuables is `test-and-set`
- To increment a counter, check its current value, then add 1
- If more than one thread does this in parallel, updates may overlap and get lost
- Need to combine test and set into an atomic, indivisible step
- Cannot be guaranteed without adding this as a language primitive

Eg compare-and-set (x, v_1, v_2)

if $x == v_1$
  $x = v_2$

Semaphores

- Programming language support for mutual exclusion

DIJKSTRA

Semaphores

- Programming language support for mutual exclusion
- Dijkstra's semaphores
  - Integer variable with atomic test-and-set operation

Semaphores

- Programming language support for mutual exclusion
- Dijkstra's semaphores
  - Integer variable with atomic test-and-set operation
- A semaphore S supports two atomic operations
  - P(s) — from Dutch passeren, to pass
  - V(s) — from Dutch vrygeven, to release

Semaphores

- Programming language support for mutual exclusion
- Dijkstra's semaphores
  - Integer variable with atomic test-and-set operation
- A semaphore `S` supports two atomic operations
  - `P(s)` — from Dutch passeren, to pass
  - `V(s)` — from Dutch vrygeven, to release

- `P(S)` atomically executes the following

```
if (S > 0)
  decrement S;
else
  wait for S to become positive;
```

Semaphores

- Programming language support for mutual exclusion
- Dijkstra's semaphores
  - Integer variable with atomic test-and-set operation
- A semaphore `S` supports two atomic operations
  - `P(s)` — from Dutch passeren, to pass
  - `V(s)` — from Dutch vrygeven, to release

- `P(S)` atomically executes the following

```
if (S > 0)
  decrement S;
else
  wait for S to become positive;
```

- `V(S)` atomically executes the following

```
if (there are threads waiting
    for S to become positive)
  wake one of them up;
    //choice is nondeterministic
else
  increment S;
```

Using semaphores

- Mutual exclusion using semaphores

Thread 1

```
...
P(S);
// Enter critical section
   ...
// Leave critical section
V(S);
...
```

Thread 2

```
...
P(S);
// Enter critical section
   ...
// Leave critical section
V(S);
...
```

Different locks for diff situation

replaces complicated setting of protocol variable + busy wait

Using semaphores

- Mutual exclusion using semaphores

```
Thread 1                         Thread 2
...                              ...
P(S);                            P(S);
// Enter critical section        // Enter critical section
   ...                              ...
// Leave critical section        // Leave critical section
V(S);                            V(S);
...                              ...
```

- Semaphores guarantee
  - Mutual exclusion
  - Freedom from starvation — can happen due to poor scheduling
  - Freedom from deadlock

Problems with semaphores

- Too low level

Problems with semaphores

- Too low level
- No clear relationship between a semaphore and the critical region that it protects

Problems with semaphores

- Too low level
- No clear relationship between a semaphore and the critical region that it protects
- All threads must cooperate to correctly reset semaphore

Problems with semaphores

- Too low level
- No clear relationship between a semaphore and the critical region that it protects
- All threads must cooperate to correctly reset semaphore
- Cannot enforce that each `P(S)` has a matching `V(S)`

P(s)
if —
V(s)
else
V(s)

Problems with semaphores

- Too low level
- No clear relationship between a semaphore and the critical region that it protects
- All threads must cooperate to correctly reset semaphore
- Cannot enforce that each P(S) has a matching V(S)
- Can even execute V(S) without having done P(S)

Counting semaphores   k out of n access

Monitors

- Attach synchronization control to the data that is being protected

Monitors

- Attach synchronization control to the data that is being protected
- Monitors — Per Brinch Hansen and CAR Hoare

Monitors

- Attach synchronization control to the data that is being protected
- Monitors — Per Brinch Hansen and CAR Hoare
- Monitor is like a class in an OO language
  - Data definition — to which access is restricted across threads
  - Collections of functions operating on this data — all are implicitly mutually exclusive

```
monitor bank_account{
  double accounts[100];

  boolean transfer (double amount,
                        int source,
                        int target){
    if (accounts[source] < amount){
      return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
  }

  double audit(){
    // compute balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++){
      balance += accounts[i];
    }
    return balance;
  }
}
```

Data

f1

f2

Monitors

- Attach synchronization control to the data that is being protected
- Monitors — Per Brinch Hansen and CAR Hoare
- Monitor is like a class in an OO language
  - Data definition — to which access is restricted across threads
  - Collections of functions operating on this data — all are implicitly mutually exclusive
- Monitor guarantees mutual exclusion — if one function is active, any other function will have to wait for it to finish

```
monitor bank_account{
  double accounts[100];

  boolean transfer (double amount,
                        int source,
                        int target){
    if (accounts[source] < amount){
      return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
  }

  double audit(){
    // compute balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++){
      balance += accounts[i];
    }
    return balance;
  }
}
```

Monitors: external queue

- Monitor ensures `transfer` and `audit` are mutually exclusive

```
monitor bank_account{
  double accounts[100];

  boolean transfer (double amount,
                        int source,
                        int target){
    if (accounts[source] < amount){
      return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
  }

  double audit(){
    // compute balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++){
      balance += accounts[i];
    }
    return balance;
  }
}
```

Monitors: external queue

- Monitor ensures `transfer` and `audit` are mutually exclusive
- If `Thread 1` is executing `transfer` and `Thread 2` invokes `audit`, it must wait

```
monitor bank_account{
  double accounts[100];

  boolean transfer (double amount,
                    int source,
                    int target){
    if (accounts[source] < amount){
      return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
  }

  double audit(){
    // compute balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++){
      balance += accounts[i];
    }
    return balance;
  }
}
```

Monitors: external queue

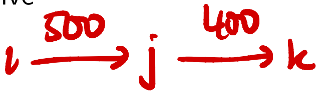
- Monitor ensures `transfer` and `audit` are mutually exclusive
- If `Thread 1` is executing `transfer` and `Thread 2` invokes `audit`, it must wait
- Implicit queue associated with each monitor
  - Contains all processes waiting for access
  - In practice, this may be just a set, not a queue

```
monitor bank_account{
  double accounts[100];

  boolean transfer (double amount,
                    int source,
                    int target){
    if (accounts[source] < amount){
      return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
  }

  double audit(){
    // compute balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++){
      balance += accounts[i];
    }
    return balance;
  }
}
```

Making monitors more flexible

- Our definition of monitors may be too restrictive

```
transfer(500.00,i,j);
transfer(400.00,j,k);
```



Making monitors more flexible

- Our definition of monitors may be too restrictive

```
transfer(500.00,i,j);
transfer(400.00,j,k);
```

- This should always succeed if `accounts[i] > 500`

Making monitors more flexible

- Our definition of monitors may be too restrictive

```
transfer(500.00,i,j);
transfer(400.00,j,k);
```

- This should always succeed if `accounts[i] > 500`
- If these calls are reordered and `accounts[j] < 400` initially, this will fail

Making monitors more flexible

- Our definition of monitors may be too restrictive

```
transfer(500.00,i,j);
transfer(400.00,j,k);
```

- This should always succeed if `accounts[i] > 500`
- If these calls are reordered and `accounts[j] < 400` initially, this will fail
- A possible fix — let an account wait for pending inflows

```
boolean transfer (double amount, int source, int target){
  if (accounts[source] < amount){
    // wait for another transaction to transfer money
    // into accounts[source]
  }
  accounts[source] -= amount;
  accounts[target] += amount;
  return true;
}
```

] How will this happen?

Monitors — wait()

```
boolean transfer (double amount, int source, int target){
  if (accounts[source] < amount){
    // wait for another transaction to transfer money
    // into accounts[source]
  }
  accounts[source] -= amount;
  accounts[target] += amount;
  return true;
}
```

- All other processes are blocked out while this process waits!

Monitors — wait()

```
boolean transfer (double amount, int source, int target){
  if (accounts[source] < amount){
    // wait for another transaction to transfer money
    // into accounts[source]
  }
  accounts[source] -= amount;
  accounts[target] += amount;
  return true;
}
```

- All other processes are blocked out while this process waits!
- Need a mechanism for a thread to suspend itself and give up the monitor

Monitors — wait()

```
boolean transfer (double amount, int source, int target){
  if (accounts[source] < amount){
    // wait for another transaction to transfer money
    // into accounts[source]
  }
  accounts[source] -= amount;
  accounts[target] += amount;
  return true;
}
```

- All other processes are blocked out while this process waits!
- Need a mechanism for a thread to suspend itself and give up the monitor
- A suspended process is waiting for monitor to change its state

Monitors — wait()

```
boolean transfer (double amount, int source, int target){
  if (accounts[source] < amount){
    // wait for another transaction to transfer money
    // into accounts[source]
  }
  accounts[source] -= amount;
  accounts[target] += amount;
  return true;
}
```

- All other processes are blocked out while this process waits!
- Need a mechanism for a thread to suspend itself and give up the monitor
- A suspended process is waiting for monitor to change its state
- Have a separate internal queue, as opposed to external queue where initially blocked threads wait

Monitors — wait()

```
boolean transfer (double amount, int source, int target){
  if (accounts[source] < amount){
    // wait for another transaction to transfer money
    // into accounts[source]
  }
  accounts[source] -= amount;
  accounts[target] += amount;
  return true;
}
```

- All other processes are blocked out while this process waits!
- Need a mechanism for a thread to suspend itself and give up the monitor
- A suspended process is waiting for monitor to change its state
- Have a separate internal queue, as opposed to external queue where initially blocked threads wait
- Dual operation to notify and wake up suspended processes

Monitors — notify()

```
boolean transfer (double amount, int source, int target){
  if (accounts[source] < amount){  wait();  }
  accounts[source] -= amount;
  accounts[target] += amount;
  notify();
  return true;
}
```

Monitors — notify()

```
boolean transfer (double amount, int source, int target){
  if (accounts[source] < amount){  wait();  }
  accounts[source] -= amount;
  accounts[target] += amount;
  notify();
  return true;
}
```

- What happens when a process executes `notify()`?

Monitors — notify()

```
boolean transfer (double amount, int source, int target){
  if (accounts[source] < amount){  wait();  }
  accounts[source] -= amount;
  accounts[target] += amount;
  notify();
  return true;
}
```

- What happens when a process executes `notify()`?
- Signal and exit — notifying process immediately exits the monitor
  - `notify()` must be the last instruction

Monitors — notify()

```
boolean transfer (double amount, int source, int target){
  if (accounts[source] < amount){  wait();  }
  accounts[source] -= amount;
  accounts[target] += amount;
  notify();
  return true;
}
```

- What happens when a process executes `notify()`?
- Signal and exit — notifying process immediately exits the monitor
  - `notify()` must be the last instruction
- Signal and wait — notifying process swaps roles and goes into the internal queue of the monitor

Monitors — notify()

```
boolean transfer (double amount, int source, int target){
  if (accounts[source] < amount){  wait();  }
  accounts[source] -= amount;
  accounts[target] += amount;
  notify();
  return true;
}
```

- What happens when a process executes `notify()`?
- Signal and exit — notifying process immediately exits the monitor
  - `notify()` must be the last instruction
- Signal and wait — notifying process swaps roles and goes into the internal queue of the monitor
- Signal and continue — notifying process keeps control till it completes and then one of the notified processes steps in

Normal – Java

Monitors — `wait()` and `notify()`

- Should check the `wait()` condition again on wake up
  - Change of state may not be sufficient to continue — e.g., not enough inflow into the account to allow transfer

Monitors — `wait()` and `notify()`

- Should check the `wait()` condition again on wake up
  - Change of state may not be sufficient to continue — e.g., not enough inflow into the account to allow transfer
- A thread can be again interleaved between notification and running
  - At wake-up, the state was fine, but it has changed again due to some other concurrent action

Monitors — wait() and notify()

- Should check the wait() condition again on wake up
  - Change of state may not be sufficient to continue — e.g., not enough inflow into the account to allow transfer
- A thread can be again interleaved between notification and running
  - At wake-up, the state was fine, but it has changed again due to some other concurrent action
- wait() should be in a while, not in an if

```
boolean transfer (double amount, int source, int target){
  while (accounts[source] < amount){  wait();  }
  accounts[source] -= amount;
  accounts[target] += amount;
  notify();
  return true;
}
```

Condition variables

- After `transfer`, `notify()` is only useful for threads waiting for target account of transfer to change state

Condition variables

- After `transfer`, `notify()` is only useful for threads waiting for target account of transfer to change state
- Makes sense to have more than one internal queue

Condition variables

- After `transfer`, `notify()` is only useful for threads waiting for target account of transfer to change state
- Makes sense to have more than one internal queue
- Monitor can have condition variables to describe internal queues

```
monitor bank_account{
  double accounts[100];
  queue q[100];  // one internal queue
                 // for each account
  boolean transfer (double amount,
                    int source,
                    int target){
    while (accounts[source] < amount){
      q[source].wait();  // wait in the queue
                         // associated with source
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    q[target].notify();  // notify the queue
                         // associated with target
    return true;
  }

  // compute the balance across all accounts
  double audit(){ ...}
}
```

Summary

- Test-and-set is at the heart of most race conditions
- Need a high level primitive for atomic test-and-set in the programming language
- Semaphores provide one such solution
- Solutions based on test-and-set are low level and prone to programming errors
- Monitors are like abstract datatypes for concurrent programming
  - Encapsulate data and methods to manipulate data
  - Methods are implicitly atomic, regulate concurrent access
  - Each object has an implicit external queue of processes waiting to execute a method
- `wait()` and `notify()` allow more flexible operation
- Can have multiple internal queues controlled by condition variables