Programming Language Concepts: Lecture 11

Madhavan Mukund

Chennai Mathematical Institute

madhavan@cmi.ac.in

http://www.cmi.ac.in/~madhavan/courses/pl2009

PLC 2009, Lecture 11, 02 March 2009

Concurrent programming

- Multiprocessing
  - Single processor executes several computations “in parallel”
  - Time-slicing to share access

Concurrent programming

- Multiprocessing
  - Single processor executes several computations "in parallel"
  - Time-slicing to share access
- Logically parallel actions within a single application
  - Clicking `Stop` terminates a download in a browser
  - User-interface is running in parallel with network access

Concurrent programming

- Multiprocessing
  - Single processor executes several computations “in parallel”
  - Time-slicing to share access
- Logically parallel actions within a single application
  - Clicking `Stop` terminates a download in a browser
  - User-interface is running in parallel with network access
- Process
  - Private set of local variables
  - Time-slicing involves saving the state of one process and loading the suspended state of another

Concurrent programming

- Multiprocessing
  - Single processor executes several computations “in parallel”
  - Time-slicing to share access
- Logically parallel actions within a single application
  - Clicking `Stop` terminates a download in a browser
  - User-interface is running in parallel with network access
- Process
  - Private set of local variables
  - Time-slicing involves saving the state of one process and loading the suspended state of another
- Threads
  - Operated on same local variables
  - Communicate via “shared memory”
  - Context switches are easier

Concurrent programming

- Multiprocessing
  - Single processor executes several computations "in parallel"
  - Time-slicing to share access
- Logically parallel actions within a single application
  - Clicking `Stop` terminates a download in a browser
  - User-interface is running in parallel with network access
- Process
  - Private set of local variables
  - Time-slicing involves saving the state of one process and loading the suspended state of another
- Threads
  - Operated on same local variables
  - Communicate via "shared memory"
  - Context switches are easier
- Henceforth, we use process and thread interchangeably

Shared variables

- Browser example: download thread and user-interface thread run in parallel
  - Shared boolean variable `terminate` indicates whether download should be interrupted
  - `terminate` is initially false
  - Clicking `Stop` sets it to true
  - Download thread checks the value of this variable periodically and aborts if it is set to true

Shared variables

- Browser example: download thread and user-interface thread run in parallel
  - Shared boolean variable `terminate` indicates whether download should be interrupted
  - `terminate` is initially false
  - Clicking `Stop` sets it to true
  - Download thread checks the value of this variable periodically and aborts if it is set to true
- Watch out for race conditions
  - Shared variables must be updated consistently

Race conditions

- Two threads increment a shared variable `n`

Thread 1	Thread 2
`...`	`...`
`m = n;`	`k = n;`
`m++;`	`k++;`
`n = m;`	`n = k;`
`...`	`...`

Race conditions

- Two threads increment a shared variable `n`

```
Thread 1            Thread 2

...                 ...
m = n;              k = n;
m++;                k++;
n = m;              n = k;
...                 ...
```

- Expect `n` to increase by 2 . . .

Race conditions

- Two threads increment a shared variable `n`

```
Thread 1          Thread 2

...               ...
m = n;            k = n;
m++;              k++;
n = m;            n = k;
...               ...
```

- Expect `n` to increase by 2 ...
- ... but, time-slicing may order execution as follows

```
Thread 1: m = n;
Thread 1: m++;
Thread 2: k = n;   // k gets the original value of n
Thread 2: k++;
Thread 1: n = m;
Thread 2: n = k;   // Same value as that set by Thread 1
```

Race conditions . . .

- Array `double accounts[100]` describes 100 bank accounts
- Two functions that operate on `accounts`

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

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

Race conditions . . .

- What are the possibilities when we execute the following?

```
Thread 1                            Thread 2
...                                 ...
status = transfer(500.00,7,8);      print (audit());
...                                 ...
```

Race conditions ...

- What are the possibilities when we execute the following?

```
Thread 1                            Thread 2
...                                 ...
status = transfer(500.00,7,8);      print (audit());
...                                 ...
```

- `audit()` can report an overall total that is 500 more or less than the actual assets
  - Depends on how actions of `transfer` are interleaved with actions of `audit`

Race conditions . . .

- What are the possibilities when we execute the following?

```
Thread 1                              Thread 2
...                                   ...
status = transfer(500.00,7,8);        print (audit());
...                                   ...
```

- `audit()` can report an overall total that is 500 more or less than the actual assets
  - Depends on how actions of `transfer` are interleaved with actions of `audit`
- Can avoid this by insisting that `transfer` and `audit` do not interleave

Race conditions . . .

- What are the possibilities when we execute the following?

```
Thread 1                          Thread 2
...                               ...
status = transfer(500.00,7,8);    print (audit());
...                               ...
```

- `audit()` can report an overall total that is 500 more or less than the actual assets
  - Depends on how actions of `transfer` are interleaved with actions of `audit`
- Can avoid this by insisting that `transfer` and `audit` do not interleave
- Should never have simultaneously have current control point of `Thread 1` within `transfer` and `Thread 2` within `audit`

Race conditions . . .

- What are the possibilities when we execute the following?

```
Thread 1                              Thread 2
...                                   ...
status = transfer(500.00,7,8);        print (audit());
...                                   ...
```

- `audit()` can report an overall total that is 500 more or less than the actual assets
  - Depends on how actions of `transfer` are interleaved with actions of `audit`
- Can avoid this by insisting that `transfer` and `audit` do not interleave
- Should never have simultaneously have current control point of `Thread 1` within `transfer` and `Thread 2` within `audit`
- Mutually exclusive access to critical regions of code

Mutual exclusion for two processes

- First attempt

```
Thread 1

...
while (turn != 1){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
turn = 2;
...
```

```
Thread 2

...
while (turn != 2){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
turn = 1;
...
```

- No assumption about initial value of `turn`!

Mutual exclusion for two processes

- First attempt

```
Thread 1

...
while (turn != 1){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
turn = 2;
...
```

```
Thread 2

...
while (turn != 2){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
turn = 1;
...
```

- No assumption about initial value of `turn`!
- Mutually exclusive access is guaranteed . . .

Mutual exclusion for two processes

- First attempt

```
Thread 1

...
while (turn != 1){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
turn = 2;
...
```

```
Thread 2

...
while (turn != 2){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
turn = 1;
...
```

- No assumption about initial value of `turn`!
- Mutually exclusive access is guaranteed ...
- ...but one thread is locked out permanently if other thread shuts down
  Starvation!

Mutual exclusion for two processes . . .

- Second attempt

```
Thread 1

...
request_1 = true;
while (request_2){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_1 = false;
...
```

```
Thread 2

...
request_2 = true;
while (request_1)
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_2 = false;
...
```

Mutual exclusion for two processes . . .

- Second attempt

```
Thread 1

...
request_1 = true;
while (request_2){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_1 = false;
...
```

```
Thread 2

...
request_2 = true;
while (request_1)
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_2 = false;
...
```

- Mutually exclusive access is guaranteed . . .

Mutual exclusion for two processes . . .

- Second attempt

```
Thread 1

...
request_1 = true;
while (request_2){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_1 = false;
...
```

```
Thread 2

...
request_2 = true;
while (request_1)
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_2 = false;
...
```

- Mutually exclusive access is guaranteed . . .
- . . . but if both threads try simultaneously, they block each other
  Deadlock!

Peterson's algorithm

Thread 1

```
...
request_1 = true;
turn = 2;
while (request_2 &&
       turn != 1){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_1 = false;
...
```

Thread 2

```
...
request_2 = true;
turn = 1;
while (request_1 &&
       turn != 2){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_2 = false;
...
```

Peterson's algorithm

```
Thread 1

...
request_1 = true;
turn = 2;
while (request_2 &&
       turn != 1){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_1 = false;
...
```

```
Thread 2

...
request_2 = true;
turn = 1;
while (request_1 &&
       turn != 2){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_2 = false;
...
```

- If both try simultaneously, `turn` decides who goes through
- If only one is alive, `request` for that process is stuck at false and `turn` is irrelevant

Beyond two processes

- Generalizing Peterson's solution to more than two processes is not trivial
- For n process mutual exclusion other solutions exist
  - e.g., Lamport's Bakery Algorithm
- Need specific clever solutions for different situations
- Need to argue correctness in each case

Programming language support

- Add programming language support for mutual exclusion

Programming language support

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

Programming language support

- Add 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

Programming language support

- Add 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;
```

Programming language support

- Add 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);
...
```

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);
...
```

- Semaphores guarantee
  - Mutual exclusion
  - Freedom from starvation
  - Freedom from deadlock

Problem with semaphores

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

Problem 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

Problem 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)`

Problem 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)`

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

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

Monitors . . .

```
monitor bank_account{

  double accounts[100];

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

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

Monitors . . .

- 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

Monitors ...

- 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

Monitors . . .

- 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

```
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;
}
```

Monitors . . .

```
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;
}
```

Monitors . . .

```
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 ...

```
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 . . .

```
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 wake up suspended processes

Monitors . . .

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

Monitors ...

```
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 . . .

```
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 ...

```
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 . . .

```
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

Monitors . . .

- A thread can be again interleaved between notification and running

Monitors . . .

- A thread can be again interleaved between notification and running
- Should check the `wait()` condition again on wake up

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

- Note: `wait()` is in a `while`, not in an `if`

Monitors ...

- 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

Monitors ...

```
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 total balance across all accounts
  double audit(){ ...}
}
```