Robotics Programming Laboratory

Bertrand Meyer
Jiwon Shin

Lecture 4:

Introduction to concurrency and
SCOOP
Multiprocessing, parallelism*

Many of today’s computations can take advantage of multiple processing units (through *multi-core* processors):

Terminology:

- **Multiprocessing**: the use of more than one processing unit in a system
- **Parallel execution**: processes running at the same time

* This slide and the next are from material developed by Sebastian Nanz as part of a jointly taught ETH course
Multitasking, concurrency

Even on systems with a single processing unit we may give the illusion of that several programs run at once

The OS switches between executing different tasks

Terminology:
- **Interleaving**: several tasks active, only one running at a time
- **Multitasking**: the OS runs interleaved executions
- **Concurrency**: multiprocessing, multitasking, or any combination
Reasons for using concurrency

1. Performance
   - Faster computation through multiprocessing

2. Convenience
   - Programs performing several actions at once (through multithreading)

3. Modeling
   - Adapting to the world’s built-in concurrency (networking, real-time, robotics, modeling)
The end of Moore's Law as we knew it

Source: Intel
What they say about concurrent programming

Intel, 2006:
- Multi-core processing is taking the industry on a fast-moving and exciting ride into profoundly new territory

Rick Rashid, head of Microsoft Research, 2007:
- Multicore processors represent one of the largest technology transitions in the computing industry today, with deep implications for how we develop software

Bill Gates:
- We have never had a problem like this.
  A breakthrough is needed.

Dave Patterson, UC Berkeley, 2007:
- Industry has basically thrown a Hail Mary. The whole industry is betting on parallel computing. They’ve thrown it, but the big problem is catching it
Heroic programmers can exploit vast amounts of parallelism...

However, none of those developments comes close to the ubiquitous support for programming parallel hardware that is required to ensure that IT’s effect on society over the next two decades will be as stunning as it has been over the last half-century
Listing 4.33: Variables for Tanenbaum’s solution

```python
state = ['thinking'] * 5
sem = [Semaphore(0) for i in range(5)]
mutex = Semaphore(1)
```

The initial value of `state` is a list of 5 copies of 'thinking'. `sem` is a list of 5 semaphores with the initial value 0. Here is the code:

Listing 4.34: Tanenbaum’s solution

```python
def get_fork(i):
    mutex.wait()
    state[i] = 'hungry'
test(i)
    mutex.signal()
    sem[i].wait()

def put_fork(i):
    mutex.wait()
    state[i] = 'thinking'
test(right(i))
test(left(i))
    mutex.signal()

def test(i):
    if state[i] == 'hungry' and
       state(left(i)) != 'eating' and
       state(right(i)) != 'eating':
        state[i] = 'eating'
    sem[i].signal()
```
Programming for heroes: dining philosophers

Listing 4.33: Variables for Tanenbaum’s solution

```python
1  state = ['thinking'] * 5
2  sem = [Semaphore(0) for i in range(5)]
3  mutex = Semaphore(1)
```

The initial value of `state` is a list of 5 copies of `'thinking'`. `sem` is a list of 5 semaphores with the initial value 0. Here is the code:

Listing 4.34: Tanenbaum’s solution

```python
1  def get_fork(i):
2      mutex.wait()
3      state[i] = 'hungry'
4      test(i)
5      mutex.signal()
6      sem[i].wait()
7
8  def put_fork(i):
9      mutex.wait()
10    state[i] = 'thinking'
11    test(right(i))
12    test(left(i))
13    mutex.signal()
14
15  def test(i):
16      if state[i] == 'hungry' and
17          state[left(i)] != 'eating' and
18          state[right(i)] != 'eating':
19          state[i] = 'eating'
20          sem[i].signal()
```
Bank transfer

transfer (source, target: ACCOUNT;
    amount: INTEGER)
    -- If enough funds, transfer amount from source to target.
    do
        if source.balance >= amount then
            source.withdraw (amount)
            target.deposit (amount)
        end
    end

transfer (Jane, Jill, 100) 1 100 0 100
transfer (Jane, Joan, 100) 2 -100 0 100
transfer (source, target: ACCOUNT; amount: INTEGER)

-- Transfer amount from source to target.

require
  source.balance >= amount

do
  source.withdraw (amount)
  target.deposit (amount)

ensure
  source.balance = old source.balance – amount
  target.balance = old target.balance + amount

end
The inability to reason from APIs

```plaintext
if acc1.balance >= 100 then transfer (acc1, acc2, 100) end

if acc1.balance >= 100 then transfer (acc1, acc3, 100) end

transfer (source, target: ACCOUNT;
  amount: INTEGER)

  -- Transfer amount from source to target.

  require
    source.balance >= amount

  do
    ...
  ensure
    source.balance = old source.balance – amount
    target.balance = old target.balance + amount
end

invariant balance >= 0
```
The core question

Can we bring concurrent programming to the same level of abstraction and convenience as sequential programming?
Four risks

Data race
- Incorrect concurrent access to shared data

Deadlock
- Computation cannot progress because of circular waiting

Starvation
- Execution favors certain processes over others, which never get executed

Priority inversion
- Locks cause a violation of priority rules
Thank you for calling Ecstatic Opera Company.
How can I help you?

(Joan) I need a single seat for next Tuesday’s performance of *Pique Dame*.

Let me check... You’re in luck! Just one left. Eighty dollars.

Great. I’ll go for it.

*Just a moment while I book it.*

Thanks.

*Sorry, there are no seats available for Tuesday.*
## Data race: scenario

<table>
<thead>
<tr>
<th>Time step</th>
<th>Active participant</th>
<th>Request or action</th>
<th>Answer or result</th>
<th>Available seats</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Theatre</td>
<td>Available seats?</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Jane</td>
<td>Seats left?</td>
<td>Yes</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Joan</td>
<td>Seats left?</td>
<td>Yes</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Joan (fast to react)</td>
<td>Please book!</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>Jane (slow to react)</td>
<td>Please book!</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>Jane’s agent (fast to act)</td>
<td>Try to book</td>
<td>Success</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>Joan’s agent (slow to act)</td>
<td>Try to book</td>
<td>Failure</td>
<td>0</td>
</tr>
</tbody>
</table>

Notation adapted from Mordechai Ben Ari, *Principles of Concurrent and Distributed Programming*
Data races (race conditions)

If processes (OS processes, threads) are completely independent, concurrency is easy

Usually, however, threads *interfere* with each other by accessing and modifying common resources, such as variables and objects

- Unwanted dependency of the computation’s result on nondeterministic interleaving is a *race condition* or *data race*
- Such errors can stay hidden for a long time and are difficult to find by testing
(Jane)

- I’d like to change my Tuesday evening seat for the matinee performance.
- Both shows are sold out, but I heard there was a customer who wanted to change the other way around. Matinee booking is handled by a different office, so let me call them and make the change.
- Thanks.
- (Ten minutes later.) “The number is still busy.”
# Deadlock: scenario

<table>
<thead>
<tr>
<th>Time step</th>
<th>Active participant</th>
<th>Request or action</th>
<th>Answer or result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Agent 1</td>
<td>Matinee available for exchange?</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>Agent 2</td>
<td>Evening available for exchange?</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>Agent 1</td>
<td>Start dialing call to agent 2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Agent 2</td>
<td>Start dialing call to agent 1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Agent 1</td>
<td>Finish dialing</td>
<td>Busy signal, because agent 2 is trying to call</td>
</tr>
<tr>
<td>6</td>
<td>Agent 2</td>
<td>Finish dialing</td>
<td>Busy signal, because agent 1 is trying to call</td>
</tr>
<tr>
<td>7</td>
<td>Agent 1 &amp; Agent 2</td>
<td>Repeat steps 3 to 6 forever as the result remains the same: busy signals</td>
<td></td>
</tr>
</tbody>
</table>
Starvation

Jane keeps calling, but agents always pick up someone else’s call
Priority inversion

Norm: normal customer
Frieda: member of “Friends of Ecstatic”: priority over Norm
Ben: benefactor (priority over both)

Bookings open at 9; Ben comes at 9:02, jumps to front of line
Cashier, handling Norm’s request, pushes Norm aside to take care of Ben; but Ben uses a credit card and the card machine is in use to check Norm’s card. So Ben, despite his elite status, has to wait.
In walks Frieda, ready to pay cash
Cashier interrupts Norm’s transaction again (card machine remains busy) and gets Frieda a ticket
Norm’s transaction resumes and, as soon as credit check finishes, is interrupted for Ben — too late, as Frieda walked away with the last ticket
## Priority inversion: scenario

<table>
<thead>
<tr>
<th>Time</th>
<th>Active participant</th>
<th>Request or action</th>
<th>Answer or result</th>
<th>Available seats</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:00</td>
<td>Theatre</td>
<td>Theater opens</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>9:01</td>
<td>Norm</td>
<td>Try to book</td>
<td>Start card check</td>
<td>1</td>
</tr>
<tr>
<td>9:02</td>
<td>Ben</td>
<td>Interrupt Norm</td>
<td>Success</td>
<td>1</td>
</tr>
<tr>
<td>9:03</td>
<td>Ben</td>
<td>Try to book</td>
<td>Card machine busy: wait</td>
<td>1</td>
</tr>
<tr>
<td>9:04</td>
<td>Norm</td>
<td>Resume transaction</td>
<td>Resume card check</td>
<td>1</td>
</tr>
<tr>
<td>9:05</td>
<td>Frieda</td>
<td>Interrupt Norm</td>
<td>Success</td>
<td>1</td>
</tr>
<tr>
<td>9:06</td>
<td>Frieda</td>
<td>Try to book</td>
<td>Success (last ticket)</td>
<td>0</td>
</tr>
<tr>
<td>9:07</td>
<td>Norm</td>
<td>Finish card check</td>
<td>Card went through (or not)</td>
<td>0</td>
</tr>
<tr>
<td>9:08</td>
<td>Ben</td>
<td>Interrupt Norm</td>
<td>Success</td>
<td>0</td>
</tr>
<tr>
<td>9:09</td>
<td>Ben</td>
<td>Try to book</td>
<td>Failure! All seats gone</td>
<td>0</td>
</tr>
</tbody>
</table>
Choices in the SCOOP model

Choice 1: object-oriented programming

- (Static) type and module structure: class
- (Dynamic) data structure: object
- Inheritance for (static) reuse and (dynamic) binding
Choice 2: processors

Computation is the responsibility of “processors”, each of which is a sequential execution mechanism

(such as a thread)
Choice 3: regions

Objects partitioned into regions
Operations on object in a given region are the responsibility of a processor, the region’s handler
Some regions, however, are passive: they do not have a handler
Consequence of choice 3

At any given time, at most one operation in progress on any given object

(In fact, on objects in any given region)

No intra-object concurrency
The execution of a call requested by a processor on objects in another region is asynchronous

Introduce distinction between:

- Routine/method *call*
- Routine *application*
The sequential view: O-O feature calls

\[ x.r(a) \]

Diagram:
- Client
  - previous
  - \( x.r(a) \)
  - next

- Supplier
  - \( r(u:A) \)
  - do
  - \(...\)
  - end

Processor
The concurrent form of call: asynchronous

Client

```
previous
x. r(a)
next
```

Client’s handler

Supplier

```
r(u : A)
do ...
end
```

Supplier’s handler
Choice 5

The application of a call has exclusive access to the needed object
Choice 6

The application of a call has exclusive access to the needed objects
A query is blocking (synchronous)

Based on distinction between two kinds of operation:

- **Command**: does something
  (in programming languages: procedures)

- **Query**: gives some information
  (in programming languages: functions, fields/attributes/instance variables)
Choice 6

The application of a call has exclusive access to the needed \textbf{objects}
transfer (source, target: ACCOUNT; amount: INTEGER)

-- Transfer amount from source to target.

require
  source.balance >= amount

do
  source.withdraw (amount)
  target.deposit (amount)

ensure
  source.balance = old source.balance – amount
  target.balance = old target.balance + amount

end
An operation on an object may have to wait until a condition is satisfied (expressed by a precondition)
Using preconditions for waiting

```plaintext
if acc1.balance >= 100 then transfer (acc1, acc2, 100) end
if acc1.balance >= 100 then transfer (acc1, acc3, 100) end
```

```plaintext
transfer (source, target: ACCOUNT;
  amount: INTEGER)

  -- Transfer amount from source to target.
require
  source.balance >= amount
do
  ...
ensure
  source.balance = old source.balance – amount
target.balance = old target.balance + amount
end
```
Hexapod robot

Hind legs have force sensors on feet and retraction limit switches
Hexapod locomotion

Alternating protraction and retraction of tripod pairs

- Begin protraction only if partner legs are down
- Depress legs only if partner legs have retracted
- Begin retraction when partner legs are up

Ganesh Ramanathan, Benjamin Morandi, IROS 2011
Hexapod coordination rules

**R1**: Protraction can start only if partner group on ground

**R2.1**: Protraction starts on completion of retraction

**R2.2**: Retraction starts on completion of protraction

**R3**: Retraction can start only when partner group raised

**R4**: Protraction can end only when partner group retracted

Sequential implementation

```
TripodLeg lead = tripodA;
TripodLeg lag = tripodB;

while (true)
{
    lead.Raise();
    lag.Retract();
    lead.Swing();
    lead.Drop();

    TripodLeg temp = lead;
    lead = lag;
    lag = temp;
}
```
private object m_protractionLock = new object;

private void ThreadProcWalk(object obj)
{
    TripodLeg leg = obj as TripodLeg;
    while (Thread.CurrentThread.ThreadState != ThreadState.Abandoned)
    {
        // Waiting for protraction lock
        lock (m_protractionLock)
        {
            // Waiting for partner leg drop
            leg.Pair.DroppedEvent.WaitOne();
            leg.Raise();
        }
        leg.Swing();

        // Waiting for partner retraction
        leg.Pair.RetractionEvent.WaitOne();
        leg.Drop();

        // Waiting for partner raise
        leg.Pair.RaisedEvent.WaitOne();
        leg.Retract();
    }
}
begin_protracition (partner, me: separate LEG_GROUP)

require

me.legs_retracted
partner.legs_down
not partner.protracition_pending

do

tripod.lift
me.set_protracition_pending
end
Hexapod coordination rules

**R1:** Protration can start only if partner group on ground

**R2.1:** Protration starts on completion of retraction

**R2.2:** Retraction starts on completion of protration

**R3:** Retraction can start only when partner group raised

**R4:** Protration can end only when partner group retracted

Using preconditions for exclusive access

\[
\text{transfer (source, target: separate ACCOUNT; amount: INTEGER)}
\]

-- If enough funds, transfer amount from source to target.
\[
do
\]
\[
\text{if source.balance} \geq \text{amount then}
\]
\[
\text{source.withdraw (amount)}
\]
\[
\text{target.deposit (amount)}
\]
\[
\text{end}
\]
\[
\text{end}
\]

\[
\text{transfer (Jane, Jill, 100)}
\]

\[
\text{transfer (Jane, Joan, 100)}
\]
class PHILOSOPHER create make feature
  left, right: separate FORK
make (u, v: separate FORK) do left:= u ; right := v end
live
do from until False loop
  think ; eat (left, right)
end
den

eat (l, r: separate FORK) do left.pick ; right.pick ; ... end
 think do ... end
end
To know more

SCOOP pages at

- [http://cme.ethz.ch/scoop/](http://cme.ethz.ch/scoop/)