COMP 322: Fundamentals of Parallel Programming

Lecture 31: More on Actors

Shams Imam (guest lecturer)
Department of Computer Science, Rice University
shams@rice.edu

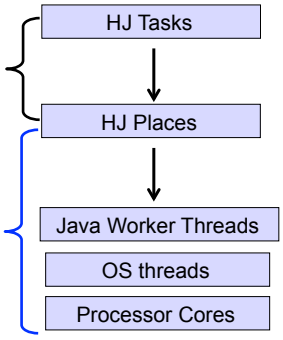
https://wiki.rice.edu/confluence/display/PARPROG/COMP322

Places in HJ

HJ programmer defines mapping from HJ tasks to set of places

HJ runtime defines mapping from places to one or more worker Java threads per place

The option "-places p:w" when executing an HJ program can be used to specify
p, the number of places
w, the number of worker threads per place



Places in HJ (Recap)

here = place at which current task is executing

place.MAX_PLACES = total number of places (runtime constant)
Specified by value of p in runtime option, -places p:w

place.factory.place(i) = place corresponding to index i

<place-expr>.toString() returns a string of the form "place(id=0)"

<place-expr>.id returns the id of the place as an int

async at(P) S

- Creates new task to execute statement S at place P
- async S is equivalent to async at(here) S
- Main program task starts at place.factory.place(0)

Note that here in a child task refers to the place P at which the child task is executing, not the place where the parent task is executing

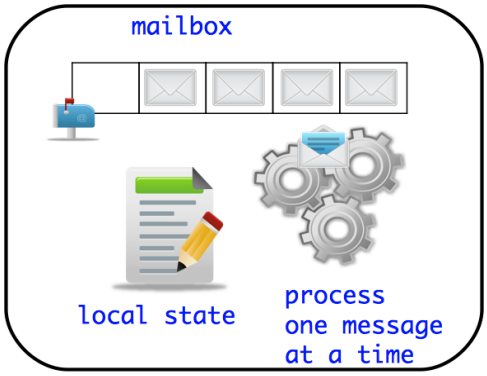
What is an Actor?

- Actors are computational entities
  - They encapsulate some local state
    - No shared mutable state :)
      - Other actors cannot directly change this state
  - They are passive and lazy
    - Only respond if messages are sent to them
      - Usually messages come from other actors
    - Only process one message at a time
      - Store pending messages in a mailbox
    - Mutate local state only while processing a message
    - Mutating local state can result in actor responding differently to subsequent messages

Using Actors in HJ

- Create your custom class which extends hj.lang.Actor<Object> and implement the void process() method
- ```
  class MyActor extends Actor<Object> {
    protected void process(Object message) {
      System.out.println("Processing " + message);
  } }
  ```
- Instantiate and start your actor
  ```
  Actor<Object> anActor = new MyActor();
  anActor.start()
  ```
- Send messages to the actor
  ```
  anActor.send(aMessage); //aMessage can be any object in general
  ```
- Use a special message to terminate an actor
  ```
  protected void process(Object message) {
    if (message.someCondition()) exit();
  }
  ```
- Actor execution implemented as async tasks in HJ

Anatomy of an Actor



- Message are processed asynchronously (i.e. async tasks)
- Can we obtain data locality benefits from local state?
- Where is the message processed?
  ```

Adding support for places in HJ actors

- Basic approach: include an optional place parameter in the start() method

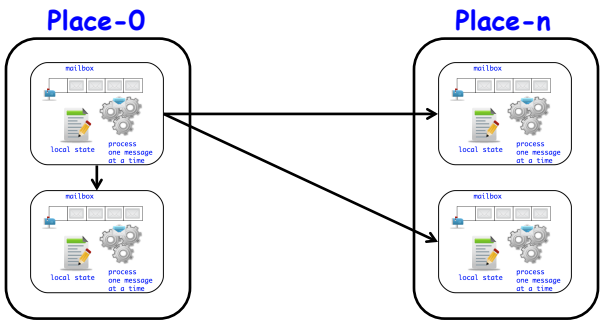
```
Actor<Object> anActor = new MyActor();
anActor.start(p); // Start actor at place p
```

- Example:

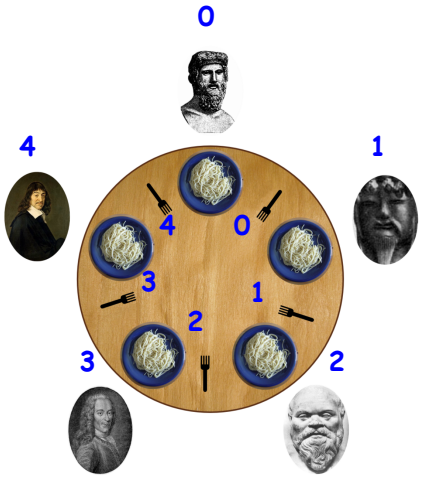
```
SievePlaceActor nextActor = new SievePlaceActor(...);
// Start actor at next place, relative to current place
nextActor.start(here.next());
// This ensures locality with respect to local primes stored
```


Actor and Places

- Places act as containers for Actors
- Actors from different places can send each other messages
- Actor always processes the message in a specified place
  - Easier to achieve data locality via local state



The Dining Philosophers Problem



Constraints

- Five philosophers either eat or think
- They must have two forks to eat (don't ask why)
- Can only use forks on either side of their plate
- No talking permitted

Goals

- Progress guarantees
  - Deadlock freedom
  - Livelock freedom
  - Starvation freedom
  - Bounded wait
- Maximize concurrency when eating

Dining philosophers using Actors

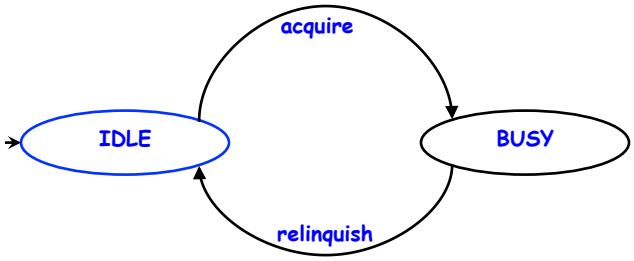
- Basic approach: use actors to model state transition diagrams
- Philosophers and Forks treated as actors
- Each has their own state transition diagram
- Based on solution proposed by: http://www.dalnefre.com/wp/2010/08/dining-philosophers-in-humus/
- HJ solution available for download at the course wiki

Philosophers – HJ solution

```
// use Behavior to represent a state in the transition diagram
interface Behavior {
  Behavior process(Object theMsg);
}
class BehavingActor extends Actor<Object> {
  protected Behavior currentBehavior;
  protected final void process(Object theMsg) {
    if (currentBehavior == null) {
      throw new IllegalStateException("Current behavior is null!");
    }
    // update the state in the transition diagram
    currentBehavior = currentBehavior.process(theMsg);
} }
class Fork extends BehavingActor {
  public Fork (int id) {...}
  ...
}
class Philosopher extends BehavingActor {
  public Philosopher(int id, final Fork left, final Fork right) {...}
  ...
}
```


Forks – state transition diagram

- Basic thinking/eating cycle

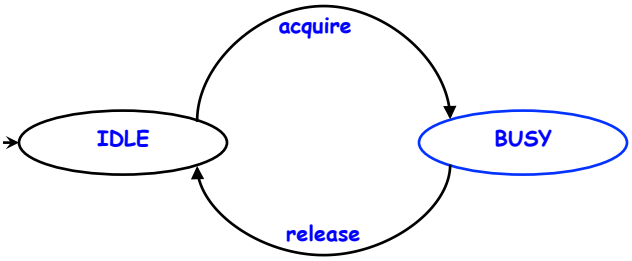


Forks – HJ solution

```
1.  class Fork extends BehavingActor {
2.    ...
3.    private final Behavior IdleBehavior = new Behavior() {
4.      public Behavior process(Object theMsg) {
5.        final Actor<Object> source = ((Message) theMsg).source;
6.        if (theMsg instanceof Acquire) {
7.          // acquire an available fork
8.          source.send(new Allow(Fork.this));
9.          return BusyBehavior;
10.       } else if (theMsg instanceof Relinquish) {
11.         return IdleBehavior;
12.       } else {
13.         throw new UnsupportedOperationException();
14.       }
15.     }
16.   };
17. }
```


Forks – state transition diagram

- Basic thinking/eating cycle



Forks – HJ solution

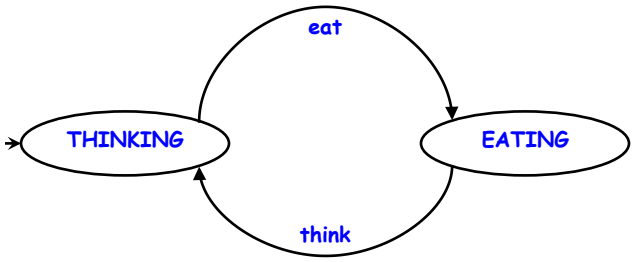
```
1.  class Fork extends BehavingActor {
2.    ...
3.    private final Behavior BusyBehavior = new Behavior() {
4.      public Behavior process(Object theMsg) {
5.        final Actor<Object> source = ((Message) theMsg).source;
6.        if (theMsg instanceof Acquire) {
7.          // fork is busy, deny request
8.          source.send(new Deny(Fork.this));
9.          return BusyBehavior;
10.       } else if (theMsg instanceof Release) {
11.         // fork is now idle
12.         return IdleBehavior;
13.       } else if (theMsg instanceof Relinquish) {
14.         // track philosopher relinquishing
15.         return BusyBehavior;
16.       } else {
17.         throw new UnsupportedOperationException();
18. } } }; }
```


Forks – State Transition Table

	Acquire	Relinquish	Release
IDLE	BUSY	EXIT	ERROR
BUSY	BUSY	EXIT	IDLE

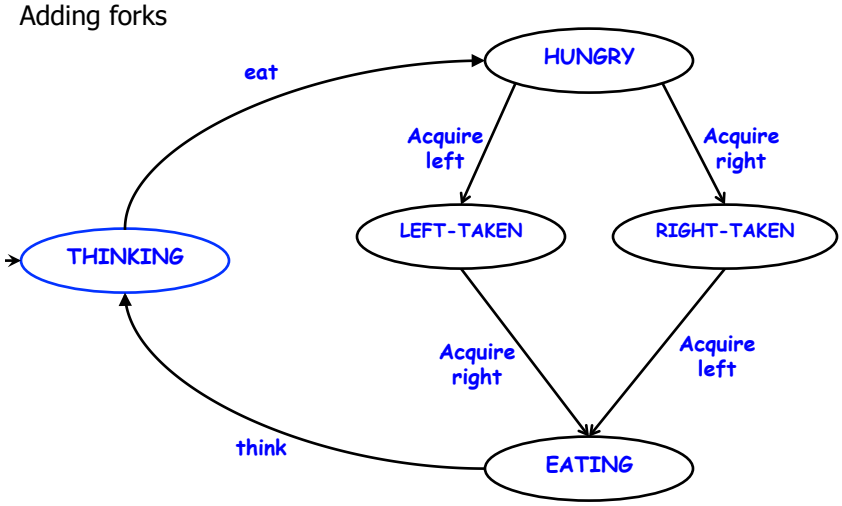
Philosophers – state transition diagram

- Basic thinking/eating cycle



Philosophers – state transition diagram

- Adding forks

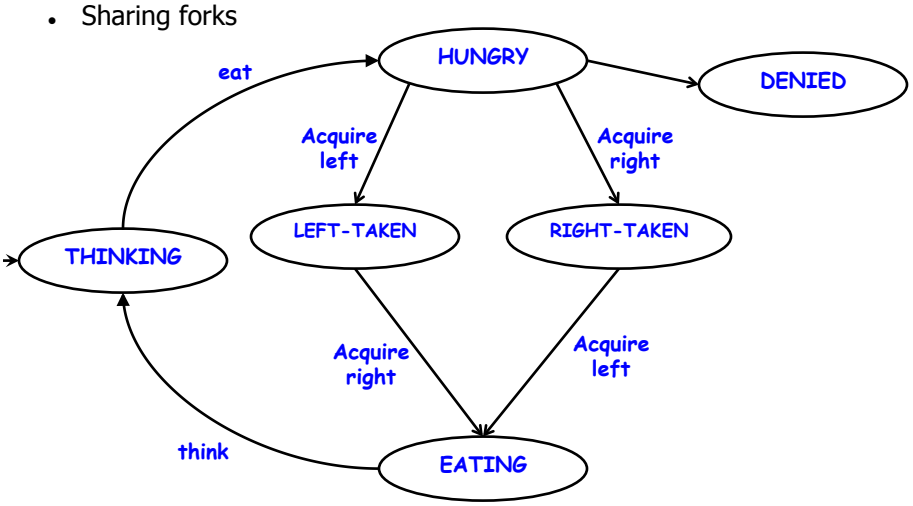


Philosophers – HJ solution

```
1.  class Philosopher extends BehavingActor {
2.    ...
3.    private final Behavior ThinkingBehavior = new Behavior() {
4.      public Behavior process(Object theMsg) {
5.        if (theMsg instanceof Eat) {
6.          // request forks and switch to hungry
7.          final Actor<Object> source = ((Message) theMsg).source;
8.          println(Philosopher.this + " becomes hungry, cause=" + source);
9.          left.send(new Acquire(Philosopher.this);
10.         right.send(new Acquire(Philosopher.this);
11.         return HungryBehavior;
12.       } else {
13.         throw new UnsupportedOperationException();
14.   } } }; }
```


Philosophers – state transition diagram

- Sharing forks

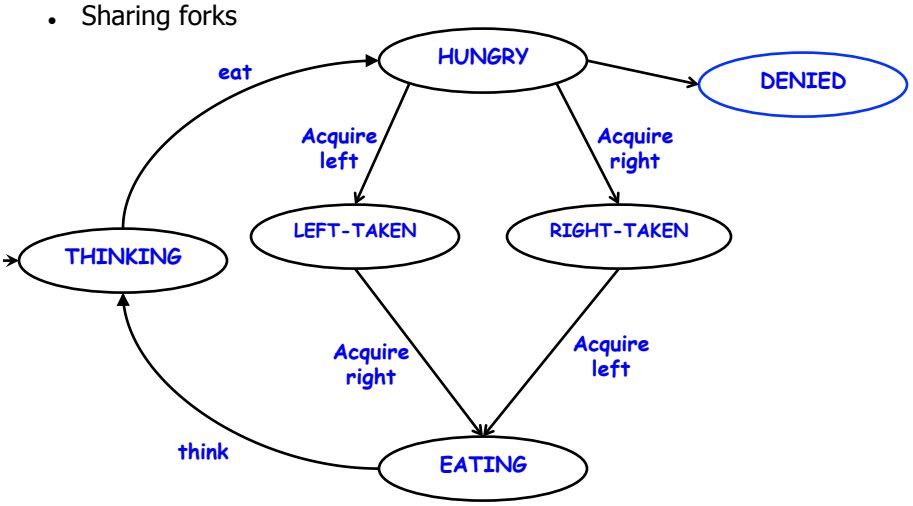


Philosophers – HJ solution

```
1.  class Philosopher extends BehavingActor {
2.    ...
3.    private final Behavior HungryBehavior = new Behavior() {
4.      public Behavior process(Object theMsg) {
5.        final Actor<Object> source = ((Message) theMsg).source;
6.        if (theMsg instanceof Deny) {
7.          // one of the forks unavailable
8.          return DeniedBehavior;
9.        } else if (theMsg instanceof Allow) {
10.         if (source == left) { // wait on the right fork
11.           return RightWaitingBehavior;
12.         } else { // wait on the left fork
13.           return LeftWaitingBehavior;
14.         }
15.       } else {
16.         throw new UnsupportedOperationException();
17. } } }; }
```


Philosophers – state transition diagram

- Sharing forks

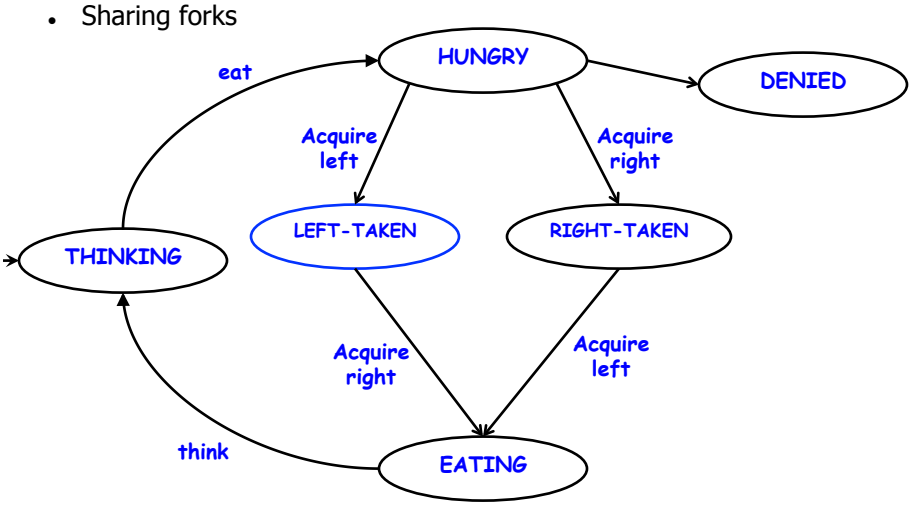


Philosophers – HJ solution

```
1.  class Philosopher extends BehavingActor {
2.    ...
3.    private final Behavior DeniedBehavior = new Behavior() {
4.      public Behavior process(Object theMsg) {
5.        final Actor<Object> source = ((Message) theMsg).source;
6.        if (theMsg instanceof Allow) {
7.          // TODO in class
8.        } else if (theMsg instanceof Deny) {
9.          // try eating in future, resume thinking
10.         Philosopher.this.send(new Eat(Philosopher.this));
11.         return ThinkingBehavior;
12.       } else {
13.         throw new UnsupportedOperationException();
14.   } } }; }
```


Philosophers – state transition diagram

- Sharing forks



Philosophers – HJ solution

```
1.  class Philosopher extends BehavingActor {
2.    ...
3.    private final Behavior RightWaitingBehavior = new Behavior() {
4.      public Behavior process(Object theMsg) {
5.        final Actor<Object> source = ((Message) theMsg).source;
6.        if (theMsg instanceof Allow) {
7.          // both forks available, ready to eat
8.          Philosopher.this.send(new Think(Philosopher.this));
9.          return EatingBehavior;
10.       } else if (theMsg instanceof Deny) {
11.         // release current fork, try eating in future, resume thinking
12.         releaseFork(left);
13.         Philosopher.this.send(new Eat(Philosopher.this));
14.         return ThinkingBehavior;
15.       } else {
16.         throw new UnsupportedOperationException();
17. } } }; }
```


Philosophers – HJ solution

```
1.  class Philosopher extends BehavingActor {
2.    ...
3.    private final Behavior EatingBehavior = new Behavior() {
4.      public Behavior process(Object theMsg) {
5.        final Actor<Object> source = ((Message) theMsg).source;
6.        if (theMsg instanceof Think) {
7.          // release forks
8.          releaseFork(left);
9.          releaseFork(right);
10.         Philosopher.this.send(new Eat(Philosopher.this));
11.         return ThinkingBehavior;
12.       } else {
13.         throw new UnsupportedOperationException();
14. } } }; }
```

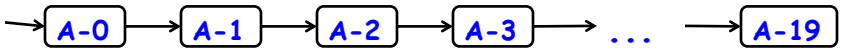

Philosophers – State Transition Table

	Eat	Think	Deny-Fork	Allow-Left	Allow-Right
THINKING	HUNGRY	ERROR	ERROR	ERROR	ERROR
HUNGRY	ERROR	ERROR	DENIED	RIGHT-WAIT	LEFT-WAIT
DENIED	ERROR	ERROR	THINKING	THINKING	THINKING
LEFT-WAIT	ERROR	ERROR	THINKING	EATING	ERROR
RIGHT-WAIT	ERROR	ERROR	THINKING	ERROR	EATING
EATING	ERROR	THINKING	ERROR	ERROR	ERROR

	Deadlock	Livelock	Starvation	Non-concurrency
Solution 1: synchronized	Yes	No	Yes	Yes
Solution 2: tryLock/unLock	No	Yes	Yes	Yes
Solution 3: isolated	No	No	Yes	Yes
Solution 4: object-based isolation	No	No	Yes	No
Solution 5: semaphores	No	No	No	No
Solution 6: actors	No	Yes	Yes	No

Worksheet #31: actors and places

Name 1: _____ Name 2: _____



Consider a pipeline of actors where an item is produced in each actor and then transferred between actors using messages. Would a block or cyclic assignment of actors to places have better data locality?

- Example with 4 places:
  - Block Distribution:
    Place 0: A-0…A-4;
    Place 1: A-5…A-9, …
  - Cyclic Distribution:
    Place 0: A-0, A-5, A-10, A-15;
    Place-1: A-1, A-6, A-11, A16, …

BACKUP SLIDES START HERE