Parallel Programming

Communication and Synchronization

- A collection of threads sharing one address space uses semaphores or monitors to communicate and synchronize
- A collection of threads in different address spaces uses message passing to communicate and synchronize
- Threads sharing one address space can also use message passing to communicate and synchronize

Message Passing

- Threads perform message passing using the following two primitive operations:
  send(to, message)
  receive(from, message)
where to and from specify source and destination addressing and message is a reference to the message

Direct & Indirect Addressing

- Messages may directly send to or receive from a thread
  send(thread-id, message)
  receive(thread-id, message)
- Messages may indirectly send to or receive from a medium called a channel, port, or mailbox
  channel channel-id;
  send(channel-id, message)
  message = receive(channel-id)
Direct & Indirect Addressing

- *Direct* addressing supports only *one-to-one* communication
- *Indirect* addressing can support *one-to-one*, *many-to-one*, *one-to-many*, and *many-to-many* communication

Blocking & Nonblocking

- *Blocking send*: the sender blocks if the receiver is not ready to receive
- *Nonblocking send*: the sender continues immediately after the message is queued
- *Blocking receive*: the receiver blocks if there is no message available
- *Nonblocking*: the receiver returns with a failure indicator if there is no message available

Buffered & Nonbuffered

- Usually *blocking receive* is used
- Messages may be *buffered* or *nonbuffered*
- *Nonbuffered* channels imply *blocking send*
- *Buffered* channels imply *nonblocking send* although bounded buffers are usually used

Synchronous & Asynchronous

- *Asynchronous* message passing means *nonblocking send* and *blocking receive*
- *Synchronous* message passing means *blocking send* and *blocking receive*
- *Synchronous* message passing is also called *rendezvous*
Remote Procedure Calls

- In client-server applications, usually two *rendezvous* are used
  - send(request) → receive(request)
  - waiter → do requested service → receive(results) → send(results)
- This pair of *rendezvous* is often structurally implemented as a *remote procedure* and a *remote procedure call*

Message Passing Interface

```java
public interface MessagePassing {
    public abstract void send(Object m) throws NotImplementedException;
    public abstract void send(int m) throws NotImplementedException;
    public abstract void send(double m) throws NotImplementedException;
}
```

Message Passing Root Class

```java
public abstract class MessagePassingRoot extends MyObject {
    public MessagePassingRoot() { super(); }
    public MessagePassingRoot(String name) { super(name); }

    public abstract void send(Object m) throws NotImplementedException;
    public void send(int m) { this.send(new Integer(m)); }
    public void send(double m) { this.send(new Double(m)); }
}
```

Message Passing Root Class

```java
public abstract Object receive() throws NotImplementedException;
public int receiveInt() throws ClassCastException {
    return ((Integer) this.receive()).intValue();
}
public double receiveDouble() throws ClassCastException {
    return ((Double) this.receive()).doubleValue();
}
public void close() throws NotImplementedException {
    throw new NotImplementedException();
}
```
Parallel Programming

Synchronous Message Passing

```java
public final class SyncMessagePassing extends MessagePassingRoot {
    private Object theMessage = null;
    private final Object sending = new Object();
    private final Object receiving = new Object();
    private final BinarySemaphore senderIn = new BinarySemaphore(0);
    private final BinarySemaphore receiverIn = new BinarySemaphore(0);

    public SyncMessagePassing() { super(); }
    // methods
}
```

Synchronous Message Passing

```java
public final void send(Object m) {
    if (m == null) {
        throw new NullPointerException();
    }
    synchronized (sending) {
        theMessage = m;
        V(senderIn);
        P(receiverIn);
    }
}
```

Synchronous Message Passing

```java
public final Object receive() {
    Object receivedMessage = null;
    synchronized (receiving) {
        P(senderIn);
        receivedMessage = theMessage;
        V(receiverIn);
    }
    return receivedMessage;
}
```

Asynchronous Message Passing

```java
public final class AsyncMessagePassing extends MessagePassingRoot {
    private int numMessages = 0;
    private final Vector messages = new Vector();

    public AsyncMessagePassing() { super(); }

    public final synchronized int numMessages() {
        return numMessages;
    }
    // methods
}
```
Asynchronous Message Passing

```java
public final synchronized void send(Object m) {
    if (m == null) {
        throw new NullPointerException();
    }
    numMessages++;
    messages.addElement(m); // at end
    if (numMessages <= 0) notify();
}
```

Asynchronous Message Passing

```java
public final synchronized Object receive() {
    Object receivedMessage = null;
    numMessages--;
    if (numMessages < 0) {
        try { wait(); }
        catch (InterruptedException e) { }
    }
    receivedMessage = messages.firstElement();
    messages.removeElementAt(0);
    return receivedMessage;
}
```

MyObject

```java
// syntactic sugar for message passing
protected static final void send(MessagePassing mp, Object o) {
    mp.send(o);
}
```

```java
protected static final void send(MessagePassing mp, int o) {
    mp.send(o);
}
```

```java
protected static final void send(MessagePassing mp, double o) {
    mp.send(o);
}
```

MyObject

```java
protected static final Object receive(MessagePassing mp) {
    return mp.receive();
}
```

```java
protected static final int receiveInt(MessagePassing mp) {
    return mp.receiveInt();
}
```

```java
protected static final double receiveDouble(MessagePassing mp) {
    return mp.receiveDouble();
}
```
Parallel Programming

An Example

class SendReceive extends MyObject {
    public static void main(String[] args) {
        AsyncMessagePassing gs = new AsyncMessagePassing();
        SyncMessagePassing ca = new SyncMessagePassing();
        Thread gsThread = new Thread(new GenerateProblem(gs, ca));
        Thread caThread = new Thread(new ComputeAnswer(gs, ca));
        gsThread.start();
        caThread.start();
        try {
            gsThread.join();
            caThread.join();
        } catch (InterruptedException e) {
            System.exit(0);
        }
    }
}

An Example

class Problem {
    public int x, y;
    public Problem(int x, int y) { this.x = x; this.y = y; }
}

class GenerateProblem extends MyObject implements Runnable {
    private MessagePassing gs = null, ca = null;
    public GenerateProblem(MessagePassing gs, MessagePassing ca) {
        this.gs = gs;
        this.ca = ca;
    }
    public void run() {
        nap(1000);
        send(gs, new Problem(1, 2));
        nap(4000);
        int z = (Integer) receive(ca).intValue();
    }
}

An Example

class ComputeAnswer extends MyObject implements Runnable {
    private MessagePassing gs = null, ca = null;
    public ComputeAnswer(MessagePassing gs, MessagePassing ca) {
        this.gs = gs;
        this.ca = ca;
    }
    public void run() {
        nap(2000);
        Problem p = (Problem) receive(gs);
        nap(1000);
        send(ca, new Integer(p.x + p.y));
    }
}

Bounded Buffer Message Passing

class BoundingBoxMessagePassing extends MessagePassingRoot {
    private final int numSlots = 0, putIn = 0, takeOut = 0;
    private Object[] buffer = null, mutexS = null, mutexR = null;
    private CountingSemaphore elements = null;
    private CountingSemaphore spaces = null;
    public BoundingBoxMessagePassing(int numSlots) {
        super("BoundingBoxMessagePassing, numSlots=" + numSlots);
        if (numSlots <= 0) throw new IllegalArgumentException("<=0");
        this.numSlots = numSlots;
        buffer = new Object[numSlots];
        elements = new CountingSemaphore(0);
        spaces = new CountingSemaphore(numSlots);
        mutexS = new Object();
        mutexR = new Object();
    }
}
Bounded Buffer Message Passing

```java
public final void send(Object value) {
    if (value == null) {
        throw new NullPointerException();
    }
    synchronized (mutexS) {
        P(spaces);
        buffer[putIn] = value;
        putIn = (putIn + 1) % numSlots;
        V(elements);
    }
}
```

Bounded Buffer Message Passing

```java
public final Object receive() {
    Object value = null;
    synchronized (mutexR) {
        P(elements);
        value = buffer[takeOut];
        takeOut = (takeOut + 1) % numSlots;
        V(spaces);
    }
    return value;
}
```

Piped Message Passing

```java
public final class PipedMessagePassing extends MessagePassingRoot {
    private PipedOutputStream outPipe = null;
    private PipedInputStream inPipe = null;
    private DataOutputStream outData = null;
    private DataInputStream inData = null;
    private Socket socket = null;
    private final Object sending = new Object();
    private final Object receiving = new Object();
    \ methods
}
```

Piped Message Passing

```java
public PipedMessagePassing() {
    super("PipedMessagePassing");
    outPipe = new PipedOutputStream();
    try {
        inPipe = new PipedInputStream(outPipe);
    } catch (IOException e) {
        throw new MessagePassingException();
    }
    outData = new DataOutputStream(outPipe);
    inData = new DataInputStream(inPipe);
}
```
Piped Message Passing

public final void send(int m) {
    synchronized (sending) {
        try { outData.writeInt(m); }
        catch (IOException e) {
            throw new MessagePassingException();
        }
    }
}

Piped Message Passing

public final int receiveInt() {
    int value = 0;
    synchronized (receiving) {
        try {
            value = inData.readInt();
        } catch (IOException e) {
            throw new MessagePassingException();
        }
    }
    return value;
}

Piped Message Passing

// For threads in different JVMs
public PipedMessagePassing(Socket socket) {
    super("PipedMessagePassing");
    if (socket == null) throw new NullPointerException();
    this.socket = socket;
    try {
        outData = new DataOutputStream(socket.getOutputStream());
        inData = new DataInputStream(socket.getInputStream());
    } catch (IOException e) {
        throw new MessagePassingException();
    }
}

Piped Message Passing

public final class ObjPipedMessagePassing
extends MessagePassingRoot {
    private PipedOutputStream outPipe = null;
    private PipedInputStream inPipe = null;
    private ObjectOutputStream outData = null;
    private ObjectInputStream inData = null;
    private Socket socket = null;
    private final Object sending = new Object();
    private final Object receiving = new Object();
    \ methods
}
Parallel Programming

Synchronization Using Message Passing

class Buffer {
    public String who;
    public double value;
    public long when;
    public Buffer() {
        who = null; value = 0.0; when = 0;
    }
}

Synchronization Using Message Passing

class Producer extends MyObject implements Runnable {
    private int pNap = 0;
    private MessagePassing mpEmpty = null, mpFull = null;
    public Producer(String name, int pNap, MessagePassing mpEmpty, MessagePassing mpFull) {
        super(name);
        this.pNap = pNap;
        this.mpEmpty = mpEmpty;
        this.mpFull = mpFull;
        new Thread(this).start();
    }
    // run()
}

Message Passing
Synchronization Using Massage Passing

class Consumer extends MyObject implements Runnable {
    private int cNap = 0;
    private MessagePassing mpEmpty = null, mpFull = null;
    public Consumer(String name, int cNap, 
        MessagePassing mpEmpty, MessagePassing mpFull) {
        super(name);
        this.cNap = cNap;
        this.mpEmpty = mpEmpty;
        this.mpFull = mpFull;
        new Thread(this).start();
    }
    // run()
}

Synchronization Using Massage Passing

class ProducerConsumer extends MyObject {
    public static void main() {
        int numSlots = 20;
        int pNap = 3; int cNap = 2; int runTime = 60;
        MessagePassing mpEmpty = new AsyncMessagePassing();
        MessagePassing mpFull = new AsyncMessagePassing();
        new Producer("Producer", pNap*1000, mpEmpty, mpFull);
        new Consumer("Consumer", cNap*1000, mpEmpty, mpFull);
        for (int i = 0; i < numSlots; i++) send(mpEmpty, new Buffer());
        nap(runTime*1000);
        System.exit(0);
    }
}

Testing Message Passing

class ChannelTest extends MyObject {
    public static void main() {
        int pNap = 3; int cNap = 2; int runTime = 60;
        MessagePassing mp = new AsyncMessagePassing();
        // or MessagePassing mp = new SyncMessagePassing();
        // or MessagePassing mp = new BBMessagePassing(5);
        // or MessagePassing mp = new PipedMessagePassing();
        // or MessagePassing mp = new ObjPipedMessagePassing();
        new Producer("Producer", pNap*1000, mp);
        new Consumer("Consumer", cNap*1000, mp);
        nap(runTime*1000);
        System.exit(0);
    }
}
Testing Message Passing

class Producer extends MyObject implements Runnable {
    public void run() {
        int napping; double item;
        while (true) {
            napping = 1 + (int) random(pNap);
            nap(napping);
            item = random();
            send(mp, item);
        }
    }
}

Testing Message Passing

class Consumer extends MyObject implements Runnable {
    public void run() {
        double item;
        int napping;
        while (true) {
            napping = 1 + (int) random(cNap);
            nap(napping);
            item = receiveDouble(mp);
        }
    }
}

Distributed Mutual Exclusion

- Solve the N-thread mutual exclusion problem with an algorithm that works in a distributed environment and does not involve a central server
- Assume that
  - Error-free communication between nodes
  - Messages may be out of order
  - Nodes do not fail or halt

Distributed Mutual Exclusion

class Message {
    public int number, id;
    public Message(int number, int id) {
        this.number = number;
        this.id = id;
    }
}
**Parallel Programming**

### Distributed Mutual Exclusion

class DistributedMutualExclusion extends MyObject {
    public static void main(String[] args) {
        int numNodes = 5;
        int runTime = 60;
        int[] napOutsideCS = new int[numNodes];
        int[] napInsideCS = new int[numNodes];
        for (int i = 0; i < numNodes; i++) {
            napOutsideCS[i] = 8;
            napInsideCS[i] = 2;
        }
        MessagePassing[] requestChannel = null,
        replyChannel = null;
        for (int i = 0; i < numNodes; i++) {
            requestChannel[i] = new AsyncMessagePassing();
            replyChannel[i] = new AsyncMessagePassing();
        }
        for (int i = 0; i < numNodes; i++) {
            new Node("Node", i, numNodes, 
              napOutsideCS[i]*1000, napInsideCS[i]*1000, 
              requestChannel, replyChannel);
            System.exit(0);
        }
    }
}

**Distributed Mutual Exclusion**

class Node extends MyObject implements Runnable {
    private static final int MAIN = 0, REQUESTS = 1, REPLIES = 2;
    private int whichOne = 0, id = -1, numNodes = -1;
    private int napOutsideCS = 0, napInsideCS = 0;
    private MessagePassing[] requestChannel = null;
    private MessagePassing[] replyChannel = null;
    private MessagePassing requestsToMe = null;
    private MessagePassing repliesToMe = null;
    private int number = 0, highNumber = 0;
    private boolean requesting = false;
    private int replyCount = 0;
    private BinarySemaphore s = new BinarySemaphore(1);
    private BinarySemaphore wakeUp = new BinarySemaphore(0);
    private boolean[] deferred = null; // methods
}

**Distributed Mutual Exclusion**

class DistributedMutualExclusion {
    public Node(String name, int id, int numNodes,
      int napOutsideCS, int napInsideCS,
      MessagePassing[] requestChannel, MessagePassing replyChannel[]) {
        super(name + " " + id); this.id = id; this.numNodes = numNodes;
        this.napOutsideCS = napOutsideCS; this.napInsideCS = napInsideCS;
        this.requestChannel = requestChannel;
        this.replyChannel = replyChannel;
        this.requestsToMe = requestChannel[id];
        this.repliesToMe = replyChannel[id];
        deferred = new boolean[numNodes];
        for (int i = 0; i < numNodes; i++) deferred[i] = false;
        new Thread(this).start();
    }
}
Distributed Mutual Exclusion

public void run() {
    int meDo = whichOne++;
    if (meDo == MAIN) {
        new Thread(this).start();
        main();
    } else if (meDo == REQUESTS) {
        new Thread(this).start();
        handleRequests();
    } else if (meDo == REPLIES) {
        handleReplies();
    }
}

Distributed Mutual Exclusion

private void main() {
    while (true) {
        outsideCS();
        chooseNumber(); // PRE-PROTOCOL
        sendRequest();
        waitForReply(); // PRE-PROTOCOL
        insideCS();
        replyToDeferredNodes(); // POST-PROTOCOL
    }
}

Distributed Mutual Exclusion

private void chooseNumber() {
    P(s);
    requesting = true;
    number = highNumber + 1;
    V(s);
}

private void sendRequest() {
    replyCount = 0;
    for (int j = 0; j < numNodes; j++)
        if (j != id) send(requestChannel[j], new Message(number, id));
}

private void waitForReply() {
    P(wakeUp);
}

Distributed Mutual Exclusion

private void replyToDeferredNodes() {
    P(s);
    requesting = false;
    V(s);
    for (int j = 0; j < numNodes; j++)
        if (deferred[j]) {
            deferred[j] = false;
            send(replyChannel[j], id);
        }
Distributed Mutual Exclusion

private void outsideCS() {
    int napping;
    napping = ((int) random(napOutsideCS)) + 1;
    nap(napping);
}

private void insideCS() {
    int napping;
    napping = ((int) random(napInsideCS)) + 1;
    nap(napping);
}

Distributed Mutual Exclusion

private void handleRequests() {
    while (true) {
        Message m = (Message) receive(requestsToMe);
        int receivedNumber = m.number, receivedID = m.id;
        highNumber = Math.max(highNumber, receivedNumber);
        P(s);
        boolean decideToDefer = requesting & & (number < receivedNumber
            || (number == receivedNumber & & id < receivedID));
        if (decideToDefer) deferred[receivedID] = true;
        else send(replyChannel[receivedID], id);
        V(s);
    }
}

Distributed Mutual Exclusion

private void handleReplies() {
    while (true) {
        int receivedID = receiveInt(repliesToMe);
        replyCount++;
        if (replyCount == numNodes - 1) V(wakeUp);
    }
}

Distributed Mutual Exclusion

- **Mutual exclusion**: a node does not enter its critical section until it receives replies from all other nodes
- **No deadlock**: ties are broken by node identifiers
- **No starvation**: if none of the other nodes wants to enter its critical section, replies are immediate, and after a node exits its critical section, it chooses a sequence number that will be higher than those of other contending nodes
Conditional Message Passing

- In conditional message passing, the message remains queued until some condition, specified by the receiver, becomes true.
- At that time, the message is passed to the receiver, unblocking it.

```java
public interface Condition {
    public abstract boolean checkCondition(Object m);
}

public interface ConditionalMessagePassing {
    public abstract void send(Object message);
    public abstract Object receive(Condition condition);
    public abstract void close();
}
```

```java
public class AsyncConditionalMessagePassing extends MyObject
    implements ConditionalMessagePassing, Runnable {

    private Thread me = null;
    private Vector blockedMessages = null;
    private Vector blockedConditions = null;
    private Vector blockedReceivers = null;

    \ methods
}
```

```java
public AsyncConditionalMessagePassing() {
    blockedMessages = new Vector();
    blockedConditions = new Vector();
    blockedReceivers = new Vector();
    me = new Thread(this);
    me.setDaemon(true);
    me.start();
}
Conditional Message Passing

```java
public void run() {
    synchronized (me) {
        while (true) {
            while (matchedMessageWithReceiver()) {
                try { me.wait(); } catch (InterruptedException e) { }
            }
        }
    }

    public void close() {
        if (me != null) me.stop();
    }
}
```

Conditional Message Passing

```java
private boolean matchedMessageWithReceiver() {
    int numMessages = blockedMessages.size();
    int numReceivers = blockedReceivers.size();
    if (numMessages == 0 || numReceivers == 0) return false;
    for (int i = 0; i < numReceivers; i++) {
        for (int j = 0; j < numMessages; j++) {
            Object m = blockedMessages.elementAt(j);
            Condition c = (Condition) blockedConditions.elementAt(i);
            // if (c.checkCondition(m)) { … }
        }
    }
    return false;
}
```

Conditional Message Passing

```java
if (c.checkCondition(m)) {
    blockedMessages.removeElementAt(j);
    blockedConditions.removeElementAt(i);
    SyncMessagePassing mp = (SyncMessagePassing) blockedReceivers.elementAt(i);
    mp.send(m);
    return true;
}
```

Conditional Message Passing

```java
public void send(Object message) {
    if (message == null) {
        throw new NullPointerException();
    }
    synchronized (me) {
        blockedMessages.addElement(message);
        me.notify();
    }
}
```
### Conditional Message Passing

class ProducersConsumers extends MyObject {
    public static void main(String[] args) {
        boolean sync = false; int pNap = 2, cNap = 3, runTime = 60;
        int numProducers = 2, numConsumers = 3;
        ConditionalMessagePassing cmp = null;
        if (sync) cmp = new SyncConditionalMessagePassing();
        else cmp = new AsyncConditionalMessagePassing();
        for (int i = 0; i < numProducers; i++)
            new Producer("PRODUCER"+i, pNap*1000, cmp);
        for (int i = 0; i < numConsumers; i++)
            new Consumer("Consumer"+i, cNap*1000, cmp);
        nap(runTime*1000); System.exit(0);
    }
}

### Conditional Message Passing

class ConsumerCondition extends MyObject implements Condition {
    private double value = 0;

    public ConsumerCondition(double value) {
        super("ConsumerCondition: value=" + value);
        this.value = value;
    }

    public boolean checkCondition(Object message) {
        return ((Double) message).doubleValue() < value;
    }
}

### Conditional Message Passing

class Producer extends MyObject implements Runnable {
    private int pNap = 0;
    private ConditionalMessagePassing cmp = null;
    public Producer(String name, int pNap, 
        ConditionalMessagePassing cmp) {
        super(name);
        this.pNap = pNap;
        this.cmp = cmp;
        new Thread(this).start();
    }
    // run()
}
Parallel Programming

**Conditional Message Passing**

```java
public void run() {
    double item;
    int napping;
    while (true) {
        napping = 1 + (int) random(pNap);
        nap(napping);
        item = random();
        cmp.send(new Double(item));
    }
}
```

**Conditional Message Passing**

class Consumer extends MyObject implements Runnable {
    private int cNap = 0;
    private ConditionalMessagePassing cmp = null;
    public Consumer(String name, int cNap, ConditionalMessagePassing cmp) {
        super(name);
        this.cNap = cNap;
        this.cmp = cmp;
        new Thread(this).start();
    }
    // run()
}

**Conditional Message Passing**

```java
public void run() {
    double item;
    int napping;
    while (true) {
        napping = 1 + (int) random(cNap);
        nap(napping);
        double limit = random();
        item = ((Double) cmp.receive(new ConsumerCondition(limit))).doubleValue();
    }
}
```

**Distributed Dining Philosophers**

```java
class DiningPhilosophers extends MyObject {
    public static void main(String[] args) {
        int numPhilosophers = 5, runTime = 60;
        int[] napThink = new int[numPhilosophers];
        int[] napEat = new int[numPhilosophers];
        for (int i = 0; i < numPhilosophers; i++) {
            napThink[i] = 8; napEat[i] = 2;
        }
        AsyncConditionalMessagePassing[] channel =
            new AsyncConditionalMessagePassing[numPhilosophers];
        for (int i = 0; i < numPhilosophers; i++) {
            channel[i] = new AsyncConditionalMessagePassing();
        }
```
Distributed Dining Philosophers

```java
boolean[] haveL = new boolean[numPhilosophers];
boolean[] haveR = new boolean[numPhilosophers];
boolean[] dirtyL = new boolean[numPhilosophers];
boolean[] dirtyR = new boolean[numPhilosophers];
for (int i = 0; i < numPhilosophers; i++) {
    haveL[i] = true;
    haveR[i] = dirtyL[i] = dirtyR[i] = false;
}
haveL[0] = haveR[0] = dirtyL[0] = dirtyR[0] = true;
haveL[numPhilosophers-1] = false;
```

Distributed Dining Philosophers

```java
Servant[] servant = new Servant[numPhilosophers];
for (int i = 0; i < numPhilosophers; i++)
    servant[i] = new Servant("Servant", i,
        haveL[i], dirtyL[i], haveR[i], dirtyR[i],
        channel[i], channel[(i+1)%numPhilosophers],
        channel[(i-1+numPhilosophers)%numPhilosophers]);
for (int i = 0; i < numPhilosophers; i++)
    new Philosopher("Philosopher", i,
        napThink[i]*1000, napEat[i]*1000, servant[i]);
nap(runTime*1000);
System.exit(0);
```

Distributed Dining Philosophers

```java
class Philosopher extends MyObject implements Runnable {
    private int id = 0;
    private int napThink = 0;
    private int napEat = 0;
    private Servant myServant = null;
    public Philosopher(String name, int id, int napThink, int napEat, Servant myServant) {
        this.id = id;
        this.napThink = napThink;
        this.napEat = napEat;
        this.myServant = myServant;
        new Thread(this).start();
    }
    // methods
}
```

Distributed Dining Philosophers

```java
private void think() {
    int napping;
    napping = 1 + (int) random(napThink);
    nap(napping);
}

private void eat() {
    int napping;
    napping = 1 + (int) random(napEat);
    nap(napping);
}
```
Distributed Dining Philosophers

```java
public void run() {
    while (true) {
        think();
        myServant.takeForks(id);
        eat();
        myServant.putForks(id);
    }
}
```

Distributed Dining Philosophers

```java
// message types
class Hungry {} // philosopher sends this to its servant
class NeedL {} // servants
class NeedR {} // send
class PassL {} // these to
class PassR {} // each other
```

Distributed Dining Philosophers

class ServantCondition extends MyObject implements Condition {
    private boolean hungry = false;
    private boolean dirtyL = false, dirtyR = false;

    public ServantCondition(boolean hungry, boolean dirtyL, boolean dirtyR) {
        this.hungry = hungry;
        this.dirtyL = dirtyL;
        this.dirtyR = dirtyR;
    }
    // checkCondition()
}

Distributed Dining Philosophers

```java
public boolean checkCondition(Object m) {
    if (m instanceof Hungry) return true;
    else if (!hungry) return true;
    else if (m instanceof PassL || m instanceof PassR) return true;
    else if (m instanceof NeedL && dirtyL) return true;
    else if (m instanceof NeedR && dirtyR) return true;
    else return false;
}
```
Distributed Dining Philosophers

```java
class Servant extends MyObject implements Runnable {
    private int id = 0;
    private AsyncConditionalMessagePassing myChannel = null;
    private AsyncConditionalMessagePassing leftServantChannel = null;
    private AsyncConditionalMessagePassing rightServantChannel = null;
    private boolean haveL = false, dirtyL = false;
    private boolean haveR = false, dirtyR = false;
    private BinarySemaphore eat = new BinarySemaphore(0);
    private BinarySemaphore releaseForks = new BinarySemaphore(0);

    \ methods
}
```

Distributed Dining Philosophers

```java
public Servant(String name, int id, boolean haveL, boolean dirtyL, boolean haveR, boolean dirtyR, AsyncConditionalMessagePassing myChannel, AsyncConditionalMessagePassing leftServantChannel, AsyncConditionalMessagePassing rightServantChannel) {
    this.id = id;
    this.haveL = haveL; this.dirtyL = dirtyL;
    this.haveR = haveR; this.dirtyR = dirtyR;
    this.myChannel = myChannel;
    this.leftServantChannel = leftServantChannel;
    this.rightServantChannel = rightServantChannel;
    new Thread(this).start();
}
```

Distributed Dining Philosophers

```java
public void takeForks(int id) {
    myChannel.send(new Hungry()); // non blocking
    P(eat); // wait for empty message
}

public void putForks(int id) {
    V(releaseForks); // send empty message
}
```

Distributed Dining Philosophers

```java
public void run() {
    Object message = null; ServantCondition sc = null;
    boolean hungry = false;
    while (true) {
        sc = new ServantCondition(hungry, dirtyL, dirtyR);
        message = myChannel.receive(sc);
        if (message instanceof Hungry) { ... ... ...
            } else if (message instanceof NeedR) {
                haveR = false; dirtyR = false;
                rightServantChannel.send(new PassL());
                } else if (message instanceof NeedL) {
                    haveL = false; dirtyL = false;
                    leftServantChannel.send(new PassR());
                    }
        }
    }
```
Parallel Programming

Distributed Dining Philosophers

hungry = true;
if (!haveR) rightServantChannel.send(new NeedL());
if (!haveL) leftServantChannel.send(new NeedR());
while (!(!haveR && haveL)) { // while hungry, wait for forks
    sc = new ServantCondition(hungry, dirtyL, dirtyR);
    message = myChannel.receive(sc);
    if (...) { ... }
}
V(eat); dirtyR = true; dirtyL = true;
P(releaseForks);
hungry = false;

Distributed Dining Philosophers

if (message instanceof PassL) { // left servant sends fork
    haveL = true; dirtyL = false;
} else if (message instanceof PassR) {
    haveR = true; dirtyR = false;
} else if (message instanceof NeedL) { // dirtyL is true
    haveL = false; dirtyL = false;
    leftServantChannel.send(new PassR());
    leftServantChannel.send(new NeedR());
} else if (message instanceof NeedR) { // dirtyR is true
    haveR = false; dirtyR = false;
    rightServantChannel.send(new PassL());
    rightServantChannel.send(new NeedL());
}

Distributed Dining Philosophers

• **Starvation** is prevented by requiring a philosopher to give up a dirty fork he holds
• **Deadlock** is prevented by distributing the forks initially in an asymmetric pattern:
  philosopher 1 get both forks in the dirty state, philosophers 2 through n – 1 get one clean fork each, philosopher n get no forks.
  If a circular chain starts to develop around the table, the chain breaks when it tries to pass through the philosopher having two dirty forks