The Linda Model

• A communication and coordination model for concurrent processes

• Augments any existing sequential programming language

• Consists of
  • Tuple Space -- a global shared memory
  • 4 primitive operations on Tuple Space
The Linda Model

• Tuple Space -- a container of tuples
• tuple -- an ordered sequence of typed values, or value-yielding computations
  • a tuple whose values are all computed is passive
  • a tuple with at least one value still being computed is active
The Linda Model

• The Linda primitive operations:
  • rd( ) -- “read” -- to match a tuples in TS
  • in( ) -- to match/remove tuples from TS
  • out( ) -- to place new tuples in TS
  • eval( ) -- to create new Linda processes
    (places active tuple in TS)
  • first two ops are synchronous (blocking)
    * non-blocking versions also exist: rdp( ) and inp( )
  • last two operations are asynchronous (non-blocking)
  • first three operations operate on passive tuples.
The Linda Model

• Tuple Space
  • a distributed shared memory
  • not addressable memory
    (no pointers to tuples!)
  • an associative memory
    (tuples are matched)
The Linda Model

- Tuple matching is a *generalization* of how we use hashmaps

- Hashmaps
  - key - value pairs
  - lookup key; return corresponding value

- Tuples
  - multiple keys possible (by position within tuple)
  - multiple corresponding values possible (by position)
Rinda

- An implementation of the Linda Model
- Base language: Ruby
- augmented with rd(), in(), out(), and eval()
- and predicate operations rdp() and inp()
- Let’s look at some examples!
Producer/Consumer

• Eval two processes: Producer and Consumer
• Each process has it’s own array of n elements.
• Between the two processes, a shared buffer exists that will be used to transfer the contents of the producer’s buffer to the consumer’s buffer, one element at a time
Producer/Consumer using shared variables

- Here’s the pseudo code for producer and consumer:

```plaintext
//shared variables
int buf, n = 80, p = 0, c = 0;

process Producer {
    int a[n];
    while (p < n) {
        << await (p == c); >>
        buf = a[p];
        p = p+1;
    }
}

process Consumer {
    int b[n];
    while (c < n) {
        << await (p > c); >>
        b[c] = buf;
        c = c+1;
    }
}
```
Semaphores in Linda

- **P(s):** `in("sem")`
  - attempts to match/remove a one-field tuple in TS
- **V(s):** `out("sem")`
  - places a one-field tuple in TS
- For multiple semaphores
  - you decide how to implement...
Producers/Consumers using semaphores

- Here's the pseudo code for producer and consumer procs:

```plaintext
//shared variables
int buf;
sem empty = 1;    //binary semaphores: 0 or 1
sem full = 0;

process Producer(i) {
    while (true) {
        //produce data,
        //deposit in buf.
        P(empty);
        buf = data;
        V(full);
    }
}

process Consumer(i) {
    while (true) {
        //fetch data from buf,
        //and consume it.
        P(full);
        result = buf;
        V(empty);
    }
}
```
Bounded Buffer using semaphores

• Here's the pseudo code for producer and consumer procs:

```plaintext
//shared variables
int buf[n], //counting semaphores
    front = 0, rear = 0; //range from 0 to n
sem empty = n, full = 0;

process Producer {
    while (true) {
        . . .
        // produce data,
        // deposit in buf.
        P(empty);
        buf[rear] = data;
        rear = (rear+1) % n;
        V(full);
    }
}

process Consumer {
    while (true) {
        //fetch data from buf,
        //and consume it.
        P(full);
        result = buf[front];
        front = (front+1) % n;
        V(empty);
        . . .
    }
}
```
Programming Assignment

Due: ?

• Implement Ruby/Rinda versions of the producer/consumer and bounded buffer problems using semaphores.

• Replace the “…” with print statements indicating who is producing / consuming what.
Question

• How would you handle a bounded buffer with multiple producers and consumers?
Semaphores (review)

- Binary
  - values = 0 / 1
  - operations: P(s) and V(s)
- Split Binary
  - split one semaphore into two
  - $0 \leq s_1 + s_2 \leq 1$
Semaphores (review)

- Counting
  - values = 0, 1, 2, ...
  - operations: still P(s) and V(s)
  - useful for managing fixed no. of resources

- Linda implementation
  - very natural mapping to in() and out()
  - very natural extension from binary to counting
Producer / Consumer

- All versions use split binary semaphores (e.g., empty, full)

- Version 1:
  - multiple producers / consumers
  - single shared buffer

- Version 2:
  - single producer / single consumer
  - bounded buffer (an array)
Producer / Consumer

• Question: how would you handle a bounded buffer with multiple producers and consumers?

• We solved each problem separately already!

• Version 1: multiple producers / consumers with single buffer

• Version 2: single producer / consumer with bounded buffer
Producer / Consumer (combined solution)

• Here’s the pseudo code for producer and consumer procs:

```c
//shared variables
int buf[n],
int front = 0, rear = 0;   // indices to buf
sem empty = n, full = 0;   // between producers/consumers
sem mutexD = 1,             // between different producers
    mutexF = 1;             // between different consumers

process Producer[i = 1 to M] {
    while (true) {
        // produce data; deposit in buf
        P(empty);
        P(mutexD);
        buf[rear] = data;
        rear = (rear+1)%n;
        V(mutexD);
        V(full);
    }
}

process Consumer[j = 1 to N] {
    while (true) {
        // fetch data from buf; consume it.
        P(full);
        P(mutexF);
        result = buf[front];
        front = (front+1)%n;
        V(mutexF);
        V(empty);
        . . .
    }
}
```
Semaphores (TS implementation)

// Semaphore primitives P and V (works for binary and counting sems)
// -- must be implemented over tuples in tuple space

void P(char *s)                        void V(char *s)
{                                      {
   in(“sem”, s:(strlen(s)+1));            out(“sem”, s:(strlen(s)+1));
}                                      

So this invocation:
   P(“empty”);

Tries to match a tuple like this:
   (“sem”, “empty”)

and this invocation:
   V(“full”);

Will place a tuple in TS that looks like this:
   (“sem”, “full”)
Semaphore usage
(binary / counting)

binary initialization:
sem full = 0;
sem empty = 1;

Becomes this in your C-Linda code:
V("empty");

// places a tuple in TS:
// ("sem", "empty")

// do nothing to initialize
// semaphore full = 0...

counting initialization:
sem empty = n;
sem full = 0;

Becomes this in your C-Linda code:
for (i=0, i<n; i++) {
  V("empty");
}

// places n tuples in TS
// that all look like this:
// ("sem", "empty")
Bounded buffer in Tuple Space

// C declaration of a buffer as an array of ints
int buf[n];

// Assignment of three elements to buf
buf[0] = 42;
buf[1] = 43;
buf[2] = 44;

// Equivalent assignment using distributed data structure in tuple space...
// Tuples of this form are used:
//
// ("buf", index, value)

out("buf", 0, 42);
out("buf", 1, 43);
out("buf", 2, 44);

// to access value stored in buf[13]...
int i, value;
i = 13;
rd("buf", 13, ?value);

// to consume same data, change rd() to in()...
in("buf", 13, ?value);
• Here’s the pseudo code for producer and consumer procs:

```plaintext
//shared variables -- must be implemented in tuple space
int buf[n],
int front = 0, rear = 0;   // indices to buf
sem empty = n, full = 0;   // between producers/consumers
sem mutexD = 1,         // between different producers
mutexF = 1;             // between different consumers

process Producer[i = 1 to M] {
    while (true) {
        // produce data; deposit in buf
        P(empty);
        P(mutexD);
        buf[rear] = data;
        rear = (rear+1)%n;
        V(mutexD);
        V(full);
    }
}

process Consumer[j = 1 to N] {
    while (true) {
        // fetch data from buf; consume it.
        P(full);
        P(mutexF);
        result = buf[front];
        front = (front+1)%n;
        V(mutexF);
        V(empty);
        .. .
    }
}
```
Here's how to initialize tuple space with this shared data:

```c
// shared variables -- must be implemented in tuple space
int buf[n],
    int front = 0, rear = 0;    // indices to buf
sem empty = n, full = 0;    // between producers/consumers
sem mutexD = 1,             // between different producers
    mutexF = 1;             // between different consumers

// nothing for buf[n] -- until data produced...

out("front", 0);    out("rear", 0);

for (i = 0, i < n, i++) {
    V("empty");
}
// nothing for full -- until producer produces something

V("mutexD");
V("mutexF");
```