



T-106.420

Concurrent Programming

Tuple Spaces



Tuple Space

- A different process interaction paradigm:
 - shared memory & asynchronous message passing
 - an associative memory of tagged data records, tuples
 - an unordered set (bag) of tuples (“tag”, value₁, value₂ ...)
 - normally attached as a library to a programming language
 - [LindaTupleSpaces](#), [JavaSpaces\(SUN\)](#), [Tspaces\(IBM\)](#)



Tuple Space primitives

- a tuple space is an unordered set (bag) of tuples:
`("tag", value1, value2, ...)`
- a process deposits a tuple in a tuple space with:
`OUT("tag", expr1, expr2, ...);`
after the expressions have been evaluated.
- a process extracts a tuple matching a template:
`IN("tag", field1, field2, ...);`
 - Each `fieldi` is either an expression (=value), or a formal parameter: denoted by `?var` where `var` is a variable of the process.
 - The process is delayed until there is a matching tuple i.e. it has same tag and same field values.
 - The tuple is removed and and formals are replaced with actual parameters.



Tuple Space: simple examples

- An example: semaphore `sem`:
 - V(sem) operation: `OUT ("sem") ;`
 - P(sem) operation: `IN ("sem") ;`
 - ⇒ The number of ("sem") -tuples in the tuple space equals the value of the semaphore `sem`.

- An array of semaphores:
 - V operation: `OUT ("forks", i) ;` # `V(forks [i])`
 - P operation: `IN ("forks", i) ;` # `P(forks [i])`

Exercise: Is there a way to solve the dining philosophers problem with the following simple protocol: “`IN (...) ; eat ; OUT (...) ;`” using tuples ?



Tuple Space: RD primitive

- To read without consuming:

`RD (...) ;`

like `IN (...)` but it doesn't consume the tuple;

- Both `IN` and `RD` have also non-blocking variants

`RDP` and `INP`

– if no matching tuple is found, they will return `false`.



Another example: Barrier synchronization

- Problem: N processes have to wait for each other behind a “barrier” until all of them are behind it.
- count the number of processes behind the barrier with `counter`
- assign a tag “barrier” to identify the tuple containing `counter`
- the `counter` is initialized by some process:

```
OUT("barrier", 0);
```

- a process entering the barrier must increment the counter:

```
IN("barrier", ?counter); #get barrier tuple
```

```
OUT("barrier", counter+1) #put it back with new value
```

- and starts to wait until everybody is behind he barrier:

```
RD("barrier", n); #returns without consuming  
#when matching to a tuple
```

Exercise: Assuming that the barrier synchronization between the processes is needed repeatedly, how would you change the solution and still keep it symmetric ?



Tuple Space: EVAL

- To start a new process:

```
EVAL("tag", expr1, expr2, ...);
```

- at least one of the expressions is a function call
- expressions are evaluated concurrently as separate processes
- the caller will not wait
- after all evaluations are terminated, the tuple becomes passive

- Example:

```
for (i = 1; i <= n; i++)  
    EVAL ("a" , i, f(i));
```

produces n tuples:

```
("a" , 1, f(1)), ("a" , 2, f(2)) ... ("a" , n, f(n))
```



Tuple Space: Prime number generation

- Problem: Generate the first n primes concurrently
- Solution:
 - one master forks several workers (sc. process farm paradigm)
 - master knows when enough primes have been generated and signals "stop" to all workers
 - workers check out each odd number "candidate" whether it's a prime or not
 - for testing a worker needs all "prime"s that are smaller or than $\text{sqrt}(\text{candidate})$
 - master accepts the testing "results" from workers in strictly increasing order and broadcast the "prime" s to all workers to be used in later tests.
 - "prime", "results", "candidate", "stop" refer to tuples
 - all communication between processes in done through the tuple space
 - next C-code (with a preprocessor) example is from the text book.



Tuple Space: Prime number generation

```
#include "linda.h"
#define limit 1000

void worker() {
    int primes [LIMIT] = {2,3};           /* my table of primes.
    int numPrimes = 1,i,candidate,isprime; /* other local variables.

    while (true){                         /* check the next non-tested candidate,
        if (RDP("stop"))                  /* if master doesn't tell to STOP.
            return;

        IN("candidate", ?candidate);      /* get next candidate and update it.
        OUT("candidate", candidate+2);

        i=0; isprime=1;                   /* check out factoring with primes
        while(primes[i]* primes[i]<= candidate){ /* starting from the smallest.
            if candidate%primes[i] == 0(
                isprime= 0; break;        /* candidate is not a prime.
            }
            i++;
            if (i>numPrimes {              /* bigger primes needed to continue
                numPrimes++;               /* the test factoring against candidate.
                RD("prime", numPrimes, ?primes[numPrimes]) /*get, or possibly wait for a new
            }                               /*prime from master.
        }
        OUT("results",candidate, isprime; /* return the result of the candidate.
    }
}
```

11/29/05



Tuple Space: Prime number generation

```
real_main() (int argc, char *argv[]) {
    int primes [LIMIT] = {2,3};          /* my table of primes.
    int limit, numWorkers, i,isprime;
    int numPrimes = 2,value = 5;
    limit =atoi(argv[1]);              /*get the number of primes to be computed and the
    numWorkers =atoi(argv[2]);         /*number of workers from the command line arguments.

    for (i =1; i<=numWorkers; i++)      /*fork the workers.
        EVAL("worker", worker());
    OUT("candidate", value);            /*give the first candidate (=5) to get started.
    while (numPrimes < limit) {         /*wait for new results until limit and
        IN("result", value, ?isprime)   /*accept the results in strictly increasing order.
        if (isprime) {
            primes[numPrimes] = value;   /*if result is prime, store it for later output.
            OUT("prime", numPrimes,value); /*communicate it back also to all workers.
            numPrimes++;                 /*increase the number of primes found.
        }
        value = value+2;                 /*update the value of next result to be accepted.
    }
    OUT("stop")                          /*broadcast signal to stop all workers.
    for (i= 0; i < limit; i++)          /*output primes.
        printf("&d\n", primes[i]);
}
```