We now look at a few topics in parallel computing
We now look at a few topics in parallel computing

Each year this unit is given may cover different topics so don’t be too worried if past exam papers ask questions on things that were not covered this year
Linda: an adaption of the thread pool/worker idea
Linda: an adaption of the thread pool/worker idea

Like these, it is task based, with threads choosing tasks and executing them
Linda: an adaption of the thread pool/worker idea

Like these, it is task based, with threads choosing tasks and executing them

However, now, the tasks have extra structure to guide the choice
Linda: an adaption of the thread pool/worker idea

Like these, it is task based, with threads choosing tasks and executing them.

However, now, the tasks have extra structure to guide the choice.

And the view of the system is flipped from thinking of it as a thread pool to thinking of it as a task pool.
The world is based around *tuples*: these are simple (short) ordered sequences of values
The world is based around *tuples*: these are simple (short) ordered sequences of values

E.g., [1, "hello"] and [2, "goodbye"]
The world is based around *tuples*: these are simple (short) ordered sequences of values

E.g., [1, "hello"] and [2, "goodbye"]

And there is a global pool or *tuplespace* containing these tuples
The world is based around *tuples*: these are simple (short) ordered sequences of values

E.g., [1, "hello"] and [2, "goodbye"]

And there is a global pool or *tuplespace* containing these tuples

Threads communicate via the pool by putting tuples in and taking tuples out
All communications via the pool
There are four operations the threads can execute:
There are four operations the threads can execute:

- **out** send a tuple to the pool
There are four operations the threads can execute:

- **out** send a tuple to the pool
- **in** get and remove a tuple from the pool
There are four operations the threads can execute:

- `out` send a tuple to the pool
- `in` get and remove a tuple from the pool
- `read` get but don’t remove a tuple from the pool
There are four operations the threads can execute:

- **out** send a tuple to the pool
- **in** get and remove a tuple from the pool
- **read** get but don’t remove a tuple from the pool
- **eval** create a new thread
The important bit is how `in` and `read` work
The important bit is how in and read work

The arguments to these are either (a) a literal constant (e.g., string, integer) or (b) a pattern variable, e.g., ?s, or something suitable for the language you are using
The important bit is how `in` and `read` work.

The arguments to these are either (a) a literal constant (e.g., string, integer) or (b) a pattern variable, e.g., `?s`, or something suitable for the language you are using.

A matching tuple is returned from the pool where a match is defined by:

- the tuple is the same length and
- the constant literals match
The important bit is how `in` and `read` work

The arguments to these are either (a) a literal constant (e.g., string, integer) or (b) a pattern variable, e.g., `?s`, or something suitable for the language you are using

A matching tuple is returned from the pool where a match is defined by

- the tuple is the same length and
- the constant literals match

Then the pattern variables are set to the corresponding values in the chosen tuple
For example, if the pool contains

\[1, "hello"]\], \[2, "goodbye"]\], \[1, "world"]\]

then there are two matches for \[1, ?s]\], namely \[1, "hello"]\] and \[1, "world"]\]
For example, if the pool contains

[1, "hello"], [2, "goodbye"], [1, "world"]

then there are two matches for [1, ?s], namely [1, "hello"] and [1, "world"]

One of these will be chosen, non-deterministically
For example, if the pool contains

\[[1, "hello"], [2, "goodbye"], [1, "world"]\]

then there are two matches for \[[1, ?s]\], namely \[[1, "hello"]\] and \[[1, "world"]\]

One of these will be chosen, *non-deterministically*

If the former is chosen, then the variable \(s\) will be given the value "hello"
Note that the matching and choosing done by the pool: a possible bottleneck if the implementation of the Linda library is not careful.
Note that the matching and choosing done by the pool: a possible bottleneck if the implementation of the Linda library is not careful.

Of course, the pool might itself be a multithreaded system.
Note that the matching and choosing done by the pool: a possible bottleneck if the implementation of the Linda library is not careful.

Of course, the pool might itself be a multithreaded system.

If no matching tuple exists, the call will block until one arrives.
Note that the matching and choosing done by the pool: a possible bottleneck if the implementation of the Linda library is not careful

Of course, the pool might itself be a multithreaded system

If no matching tuple exists, the call will block until one arrives

If more than one thread simultaneously matches a tuple using in, exactly one will get the tuple
Note that the matching and choosing done by the pool: a possible bottleneck if the implementation of the Linda library is not careful

Of course, the pool might itself be a multithreaded system

If no matching tuple exists, the call will block until one arrives

If more than one thread simultaneously matches a tuple using in, exactly one will get the tuple

The action of match and removal for an in is atomic
If both threads use read, there is no problem, they both get a copy
If both threads use `read`, there is no problem, they both get a copy.

If one uses an `in` and the other a `read`, it can go either way:

- `read` before `in`: they both get the tuple
- `in` before `read`: the `in` gets the tuple, the `read` doesn’t
If both threads use \texttt{read}, there is no problem, they both get a copy.

If one uses an \texttt{in} and the other a \texttt{read}, it can go either way:

- \texttt{read} before \texttt{in}: they both get the tuple
- \texttt{in} before \texttt{read}: the \texttt{in} gets the tuple, the \texttt{read} doesn’t

This non-deterministic outcome would normally be considered a programmer error.
These subtleties mean you must be quite careful with Linda.
These subtleties mean you must be quite careful with Linda.

A common paradigm is to use an initial *tag*, often an integer, as in `[1, "hello"]`, to impose some structure on the tuples.
Topics
Linda

Dining Philosophers in Linda
Dining Philosophers in Linda

We have five philosophers and shall prevent deadlock by only letting four sit at a time
Dining Philosophers in Linda

We have five philosophers and shall prevent deadlock by only letting four sit at a time

Initial conditions:
out("place ticket") four times;
out("chopstick", i) for i = 0…4
eval(phil, i) for i = 0…4
defun phil(i) {
    while true {
        think()
        in("place ticket")
        in("chopstick", i)
        in("chopstick", i+1 mod 5)
        eat()
        out("chopstick", i)
        out("chopstick", i+1 mod 5)
        out("place ticket")
    }
}

This example contains no patterns, only constant literals
defun phil(i) {
    while true {
        think()
        in("place ticket")
        in("chopstick", i)
        in("chopstick", i+1 mod 5)
        eat()
        out("chopstick", i)
        out("chopstick", i+1 mod 5)
        out("place ticket")
    }
}

This example contains no patterns, only constant literals

Note Linda does not eliminate the possibility of deadlock in badly written programs: just put the (in "place ticket") after the in of the chopsticks
Producers/Consumers is just as easy

defun producer(n) {
    out(n, make-product())
    producer(n + 1)
}
defun consumer(n) {
    var prod
    in(n, (? prod)) ; pattern
    consume-product(prod)
    consumer(n + 1)
}
Producers/Consumers is just as easy

defun producer(n) {
    out(n, make-product())
    producer(n + 1)
}
defun consumer(n) {
    var prod
    in(n, (? prod)) ; pattern
    consume-product(prod)
    consumer(n + 1)
}

We use a tag to ensure we consume values in the same order as they are produced (if that is important)
Exercise. Think about the assessed coursework using Linda
Exercise. Think about the assessed coursework using Linda. Questions of granularity are just as important in Linda as elsewhere.
Linda is easy to add to existing languages, usually as a library, occasionally with a minor tweak to the syntax for the patterns.
Linda is easy to add to existing languages, usually as a library, occasionally with a minor tweak to the syntax for the patterns.

Versions exist for C, Perl, Java and Prolog and others.
Also

• the blocking semantics lead to unwanted non-determinism: see the in vying against the read above

• some implementations have non-blocking variants of in and read, but this just adds to the uncertainty

• the low-level, unstructured nature of Linda can lead to awkward code: every application needs some mechanism to structure the tuples (tags being the simplest)

• there is no fairness on selecting tuples: a tuple can be ignored indefinitely if there are others that can be chosen

• junk can collect in the pool: tuples put in but never taken out. This can slow down the matching

• the pool can be a bottleneck
Also

• the blocking semantics lead to unwanted non-determinism: see the `in` vying against the `read` above
Also

- the blocking semantics lead to unwanted non-determinism: see the `in` vying against the `read` above
- some implementations have non-blocking variants of `in` and `read`, but this just adds to the uncertainty
Topics
Linda

Also

- the blocking semantics lead to unwanted non-determinism: see the `in` vying against the `read` above
- some implementations have non-blocking variants of `in` and `read`, but this just adds to the uncertainty
- the low-level, unstructured nature of Linda can lead to awkward code: every application needs some mechanism to structure the tuples (tags being the simplest)
Also

- the blocking semantics lead to unwanted non-determinism: see the `in vying against the `read above
- some implementations have non-blocking variants of `in and `read, but this just adds to the uncertainty
- the low-level, unstructured nature of Linda can lead to awkward code: every application needs some mechanism to structure the tuples (tags being the simplest)
- there is no fairness on selecting tuples: a tuple can be ignored indefinitely if there are others that can be chosen
• the blocking semantics lead to unwanted non-determinism: see the in vying against the read above
• some implementations have non-blocking variants of in and read, but this just adds to the uncertainty
• the low-level, unstructured nature of Linda can lead to awkward code: every application needs some mechanism to structure the tuples (tags being the simplest)
• there is no fairness on selecting tuples: a tuple can be ignored indefinitely if there are others that can be chosen
• junk can collect in the pool: tuples put in but never taken out. This can slow down the matching
the blocking semantics lead to unwanted non-determinism: see the \texttt{in} vying against the \texttt{read} above

some implementations have non-blocking variants of \texttt{in} and \texttt{read}, but this just adds to the uncertainty

the low-level, unstructured nature of Linda can lead to awkward code: every application needs some mechanism to structure the tuples (tags being the simplest)

there is no fairness on selecting tuples: a tuple can be ignored indefinitely if there are others that can be chosen

junk can collect in the pool: tuples put in but never taken out. This can slow down the matching

the pool can be a bottleneck
Further
Further

- detecting when the program needs to terminate is a problem: this could be done by putting a special “end of program” tuple in the pool; but then threads have the overhead of constantly checking for that tuple (and you need an non-blocking \texttt{read} to do so). Or have an extra field in every tuple that is a status flag, etc.
detecting when the program needs to terminate is a problem: this could be done by putting a special “end of program” tuple in the pool; but then threads have the overhead of constantly checking for that tuple (and you need an non-blocking read to do so). Or have an extra field in every tuple that is a status flag, etc.

aliasing is a problem: careful constructions of name schemes (tags, usually) are needed to ensure that tuples are not accidentally picked up by the wrong threads
Further

- detecting when the program needs to terminate is a problem: this could be done by putting a special “end of program” tuple in the pool; but then threads have the overhead of constantly checking for that tuple (and you need an non-blocking read to do so). Or have an extra field in every tuple that is a status flag, etc.

- **aliasing** is a problem: careful constructions of name schemes (tags, usually) are needed to ensure that tuples are not accidentally picked up by the wrong threads

- related is **temporal aliasing**, where information about the order tuples were put into the pool is lost: again an enumeration tag can fix this, but it has to be coded
So extensions of Linda exist, e.g., using multiple pools to structure, thus avoiding the first kind of aliasing
So extensions of Linda exist, e.g., using multiple pools to structure, thus avoiding the first kind of aliasing.

Now pools become first-class objects, and you can pass pools via pools to other threads.
So extensions of Linda exist, e.g., using multiple pools to structure, thus avoiding the first kind of aliasing.

Now pools become first-class objects, and you can pass pools via pools to other threads.

But, as always, this moves away from the initial simplicity of the Linda concept.
Linda

- is a simple abstract model of parallelism
Linda

- is a simple abstract model of parallelism
- can map reasonably well to different kinds of hardware (shared and distributed)
Linda

- is a simple abstract model of parallelism
- can map reasonably well to different kinds of hardware (shared and distributed)
- is explicitly non-deterministic, with the non-determinism mostly well delineated
Linda

- is a simple abstract model of parallelism
- can map reasonably well to different kinds of hardware (shared and distributed)
- is explicitly non-deterministic, with the non-determinism mostly well delineated
- is not suited for all kinds of problem
Linda

- is a simple abstract model of parallelism
- can map reasonably well to different kinds of hardware (shared and distributed)
- is explicitly non-deterministic, with the non-determinism mostly well delineated
- is not suited for all kinds of problem
- is not widely used, but you do see Linda being mentioned now and again, mostly for coordination between other systems
For example, the computational chemistry (molecule simulation) package Gaussian uses OpenMP on a node and uses Linda between nodes.