Asynchronous algorithms

Essentially, Asynchronous Models make no assumptions about time.

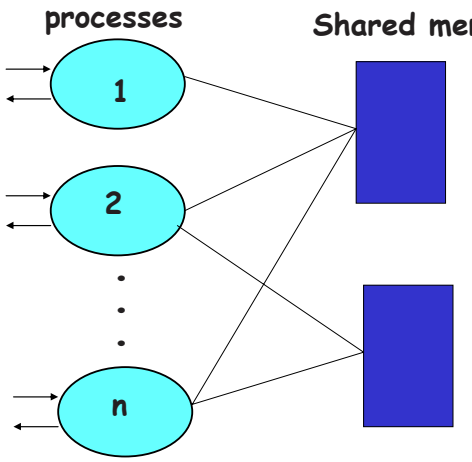
- **Execution time** (No bound on time to execute a local process step; however, time to execute a local step is finite)
- **Communication time** (No bound on message transmission delay)
- **No synchronized clocks** (no rounds)

Main differences involve liveness conditions (uncertainty caused by asynchrony and distribution)

- **Generally more difficult to program than synchronous**
- **More general than most actual distributed systems**
- **More general and portable algorithms as have fewer assumptions.**

Distr Distributed Algorithms, Nancy Lynch – Chapters 8, 9, 10

Asynchronous Shared Memory Model



First look at these systems as

- o **simpler**
- o **possible to translate into asynch networks**

We neglect failures and just deal with asynchrony.

Shared Variables and Processes

Each **shared variable** defined by set of values, initial value and read/write capabilities by processes.

Each **process** described by an algorithm, and formally modelled by a state-machine:
(*start*, *states*, *trans*)

start - initial states

states – set of states

trans - set of transitions
- maps current state and shared variable values to new state and new values.

Mutual Exclusion for 2 processes (based on Dekker/Dijkstra)

Two processes n1, n2 compete for access to their critical regions (ie. Access to a reusable resource).

Informal Algorithm:

When process n(i) wants to enter, it sets its flag(i).
If the other process's flag is set, it does not enter but resets flag(i) and tries again.
If the other process's flag is **not** set, it enters the critical region.
It resets flag(i) after leaving the region.

Properties: mutual exclusion (safety)
no deadlock (safety)
no starvation (liveness)

Lynch – Chapter 10

Mutual exclusion

shared variables: flag(i:1..2):Bool, initially **false**,
writeable by n(i), readable by all.

Process n(i):

```
...
Try:
    flag(i):= true;
    if flag((i%2)+1) then {flag(i):= false; go to Try}
                     else { enter critical region;
                            use resource
                            exit critical region }
    flag(i):= false;
...
```

R remainder
T try
C critical
E exit

Properties - **Safety**

EXCLUSION:

Need to argue that not possible for n1 and n2 to be in C, even in presence of asynchronous execution (full interleaving).

Informal PROOF by contradiction:

Consider that both n1 and n2 are in C. Both flag 1 and flag 2 must be set, and remain set from before C to E .

Consider that n1 set its flag first: then n2 must remain in T and cannot enter C as flag 1 was set. Similarly for n2. Hence contradiction.

Consider that they set their flags "simultaneously": then both remain in T and neither can enter C. Hence contradiction.

***Formal PROOF** requires that every state is specified with its pre- and post-condition, and that arguments are made about every possible execution sequence.*

cf. model checking

Properties - **Safety**

```
fluent
  CRITICAL[i:1..2] = <n[i].enter, n[i].exit>

//safety property
assert
  EXCLUSION  = [] ! (CRITICAL[1] && CRITICAL[2])

//Witness
assert
  WITNESS_EXCLUSION = !EXCLUSION
```

LTSA safety demo

*Similarly for **DEADLOCK - proof or safety check***

Properties - **Safety**

```
Violation of LTL property:
  @WITNESS_EXCLUSION
Trace to terminal set of states:
Cycle in terminal set:
  n.1.flag.1.setTrue
  n.1.flag.2.0
  n.1.enter          CRITICAL.1
  n.1.exit
  n.1.flag.1.setFalse
```

LTSA safety demo

Properties - **Liveness**

FREEDOM FROM STARVATION:

Any process that reaches T eventually enters C.

```
fluent TRYING[i:1..2] = <n[i].flag[i].setTrue,
                         n[i].enter>
assert NoSTARVATION = forall[i:1..2]
                 [](TRYING[i] -> <> CRITICAL[i])
```

Since all processes repeatedly satisfy `TRYING[i]`, we simply check:

```
assert
   NoSTARVATION = forall[i:1..2] [] <> CRITICAL[i]
```

LTSA models **strong fairness** using **fair choice**:

> **Fair Choice:** If a choice over a set of transitions is executed infinitely often, then every transition in the set is executed infinitely often.

LTSA demo

Fairness

But, this does not guarantee freedom from lockout or starvation under less fair, more **adverse conditions**. For adverse execution conditions, we cannot assume strong fairness. *Formally:*

- ***Strong Fairness:***

  for every transition, if it is enabled infinitely often, it is taken infinitely often. `[]<>p -> []<>q`

- ***Weak Fairness***

  for every transition, if it is enabled continuously from some point on, it is taken infinitely often. `<>[]p -> []<>q`

- ***No Fairness:***

  for every transition, even if it is enabled continuously from some point on, it may not be taken unless no other transitions are enabled.

Fairness

Which is the most appropriate fairness assumption?

For algorithm generality, we would like to impose the least constraint, where satisfaction of a property is such that …

- **no fairness -> weak fairness -> strong fairness**

In practice, we need to consider Process scheduling: :

(Distributed) Processes generally obey a form of fairness, where the local scheduler ensures that every process has an opportunity to execute, if enabled.

A process is enabled if any of its transitions is enabled.

ie. Local scheduling ensures that a **process** eventually executes (a transition) if any of its transitions are enabled infinitely often.

Fairness of Processes

We therefore test our algorithms under three fairness conditions:

- ***Strong Fairness: (as before)***
- ***Form of "Weak Fairness":***

  a combination of strong fairness of processes and no fairness of transitions.
- ***No Fairness: (as before)***

NOTE: Fairness assumptions are only needed for liveness, and do not affect safety properties.

How can we specify and check for this?

Weak Fairness

Model checking:

```
assert (fairness) -> (property).
```

We assert that, for those executions which satisfy the fairness condition, the required property is satisfied.

Specifying weak fairness for cyclic processes:

```
assert WEAKFAIR =
   forall processes[i] []<> {any action of [i]}
```

Satisfied by those executions which always eventually include transitions in **every** process [i]. Violated by those executions which allow any process to be ignored.

Weak Fairness

```
assert WEAKFAIR =
   forall[i:1..2] []<> {n[i].{actions of n(i)}}

assert
   WEAK_NoSTARVATION = (WEAKFAIR -> NoSTARVATION)
```

NOTE:
if no executions satisfy **WEAKFAIR**, then **WEAK_NoSTARVATION** will be trivially satisfied! We must therefore check that there are some **witnesses** for **WEAKFAIR**

```
assert WITNESS_WEAKFAIR = !WEAKFAIR
```

Strong fairness?

Weak fairness?

No fairness?

LTSA liveness demo

Weak Fairness

```
Violation of LTL property: @WEAK_NoSTARVATION
Trace to terminal set of states:
  n.1.flag.1.setTrue
  n.2.flag.2.setTrue
Cycle in terminal set:
  n.1.flag.2.1
  n.1.flag.1.setFalse
  n.1.flag.1.setTrue
  n.2.flag.1.1
  n.2.flag.2.setFalse
  n.2.flag.2.setTrue
LTL Property Check in: 156ms
```

LTSA liveness demo

Mutual Exclusion for 2 processes (Peterson2P)

Two processes n1, n2 compete for access to their critical regions, using a shared variable, **turn**.

Informal Algorithm:
When process n(i) wants to enter,
set flag(i) and set turn to i;
While the other process's flag is set **and** turn=i
{null};
Enter the critical region;
//other process's flag is **not** set **or** turn!=i,
Reset flag(i) after leaving the region.

Properties: mutual exclusion (safety)
freedom from deadlock (safety)
freedom from starvation (liveness)

Mutual exclusion

shared variables: turn:1..2, initially **undef**, read/write by all.
flag(i:1..2):Bool, initially **false**, writeable by n(i), readable by all.

Process n(i):

```
...
    flag(i):= true;
    turn := i;
    while flag((i%2)+1) && turn=i {null};
        enter critical region;
          ...
        exit critical region;
    flag(i):= false
...
```

R remainder
T try
C critical
E exit

Properties

EXCLUSION:

Not possible for n1 and n2 to be in C, even in presence of asynchronous execution (full interleaving).

Informal PROOF:

Consider that n1 wishes to gain access. It sets flag(1) and sets turn to 1.

If n2 is not competing, then flag(2) is not set and n1 can enter C. If n2 is competing, then it depends on turn. If n1 sets turn to 1 **then** n2 sets it 2, n1 can enter C, but not n2; if n2 **then** n1 sets turn, n2 can enter C, but not n1.

ie. For a process n(i) to be in C, then both flag(i) **and** turn!=i.

Formal PROOF requires that every state is specified with its pre- and post-condition, and that arguments are made about every possible execution sequence.

cf. LTSA model checking

Properties

STARVATION-FREEDOM:

Any process that reaches T eventually enters C.

Informally, if n1 is waiting in T, it will be given priority by turn when n2 exits from C. Even if n2 competes, it will give priority to n1 by setting turn to 2.

Strong fairness?

Weak fairness?

No fairness?

LTSA demo

Properties

```
Violation of LTL property: @WITNESS_WEAK_NoSTARVATION
Trace to terminal set of states:
   n.2.flag.2.setTrue
Cycle in terminal set:
   n.1.flag.1.setTrue
   n.2.turn.setto.2
   n.1.turn.setto.1
   n.2.flag.1.1
   n.2.turn.1
   n.2.enter          CRITICAL.2
   n.2.exit
   n.2.flag.2.setFalse
   n.1.flag.2.0
   n.1.turn.1
   n.1.enter          CRITICAL.1
   n.1.exit
   n.1.flag.1.setFalse
   n.2.flag.2.setTrue
LTL Property Check in: 0ms
```

LTSA liveness demo

Mutual Exclusion for N processes? (PetersonNP)

How can we generalise this two process algorithm to deal with N Processes? Series of competitions...

Informal Algorithm:

Use Peterson2P iteratively in a series of N-1 competitions at levels 1, 2, ..N-1, each with own turn.
At each successive competition level k, Peterson2P ensures at least one loser i, whose turn(k) is i if all compete.
At level 1, at most N-1 processes can proceed.
At level 2, at most N-2. ...
At level N-1, at most 1.

Properties: mutual exclusion (safety)
freedom from deadlock (safety)
freedom from starvation (liveness)

Mutual exclusion

shared variables:

turn(k:1..N-1):1..N initially **undef**, read/write by all.

flag(i:1..N):1..N-1, initially **0**, writeable by n(i), readable by all.

Process n(i):

...

```
For k=1 to N-1 do
    flag(i) := k;
    turn(k) := i;
    while (∃j!=i: flag(j)>=k) && turn(k)=i {null};
        enter critical region;
        ...
        exit critical region;
flag(i):= 0
```

...

R remainder

T try

C critical

E exit

Question

Argue (informally) that PetersonNP preserves the required property of EXCLUSION.

```
//safety property
assert
   EXCLUSION  = [](forall[i:1..N-1]
                  (CRITICAL[i] && CRITICAL[i+1..N]))
```

STARVATION-FREEDOM:

Strong fairness?

Weak fairness?

No fairness?

LTSA model checking

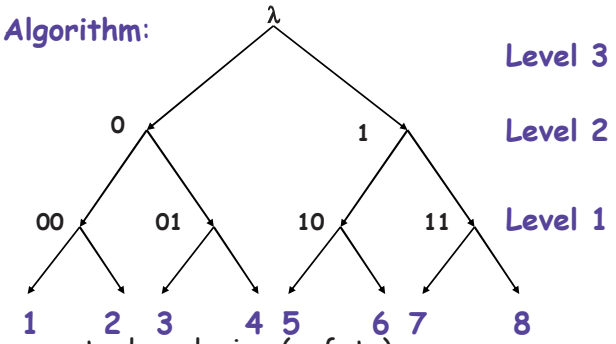
Weak NoStarvation

```
Violation of LTL property: @WITNESS_WEAK_NoSTARVATION
Trace to terminal set of states:
        n.2.flag.2.setto.1
        n.3.flag.3.setto.1
        n.3.turn.1.setto.3
        n.2.turn.1.setto.2
        n.3.flag.1.0
        n.3.turn.1.2
        n.3.flag.2.1
        n.3.turn.1.2
        n.3.flag.3.1
        n.3.turn.1.2
        n.3.flag.3.setto.2
        n.3.turn.2.setto.3
Cycle in terminal set:
        n.1.flag.1.setto.1
        n.1.turn.1.setto.1
        n.1.flag.1.1
        n.2.flag.1.1
        n.2.turn.1.1
        n.2.flag.2.1
        n.2.turn.1.1
        n.2.flag.3.2
        n.2.turn.1.1
        n.2.flag.2.setto.2
        n.2.turn.2.setto.2
        n.2.flag.1.1
        n.2.turn.2.2
        n.2.flag.2.2
        n.3.flag.1.1
        n.3.turn.2.2
        n.3.flag.2.2
        n.3.turn.2.2
        n.3.flag.3.2
        n.3.turn.2.2
        n.3.enter       CRITICAL.3
        n.3.exit
        ... ...
        n.2.enter       CRITICAL.2
        n.2.exit
        ... ...
        n.1.enter       CRITICAL.1
        n.1.exit
```

Mutual Exclusion for N processes? (PetersonNP)

Alternative generalisation of Peterson2P to deal with N Processes? Tournament of N=2^level processes …

Informal Algorithm:



Properties: mutual exclusion (safety)
freedom from deadlock (safety)
freedom from starvation (liveness)

Notes

This section has introduced **asynchronous algorithms** which share variables, and important **fairness** considerations for liveness property checking.

Can these algorithms can be implemented in a distributed fashion?

The shared data can be encapsulated in processes which support communication via message passing rather than read/write.