DoC 437 - 2008

Distributed Algorithms

Part 5: Asynchronous Algorithms

Asynchronous models

- Asynchronous models make no assumptions about *time*

    execution time

        no bound on time to execute a local process step

    communication time

        no bound on message transmission delay

    no synchronized clocks

        no rounds to structure the algorithm

Asynchronous models

- Main differences from synchronous models involve *liveness conditions*
  - uncertainty caused by asynchrony and distribution
  - more general distributed algorithms, as they embody weaker assumptions
  - more general than most distributed systems
  - usually more difficult to program than under the synchronous model

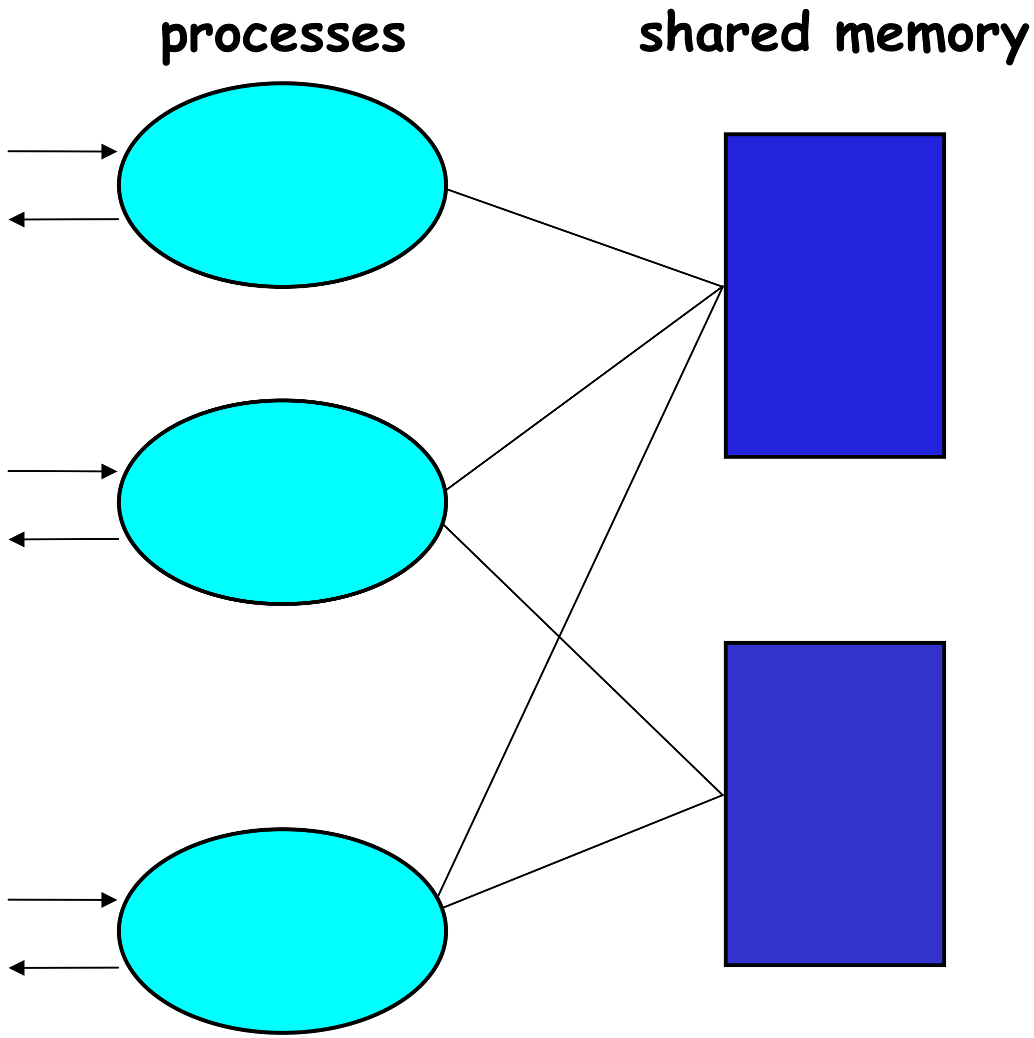
Recall: communication medium

- communication medium
  - message-passing network
    - point-to-point address
    - content-based address
    - broadcast

      specific topologies

      ring, star, tree, ...
  - shared memory

Asynchronous shared memory



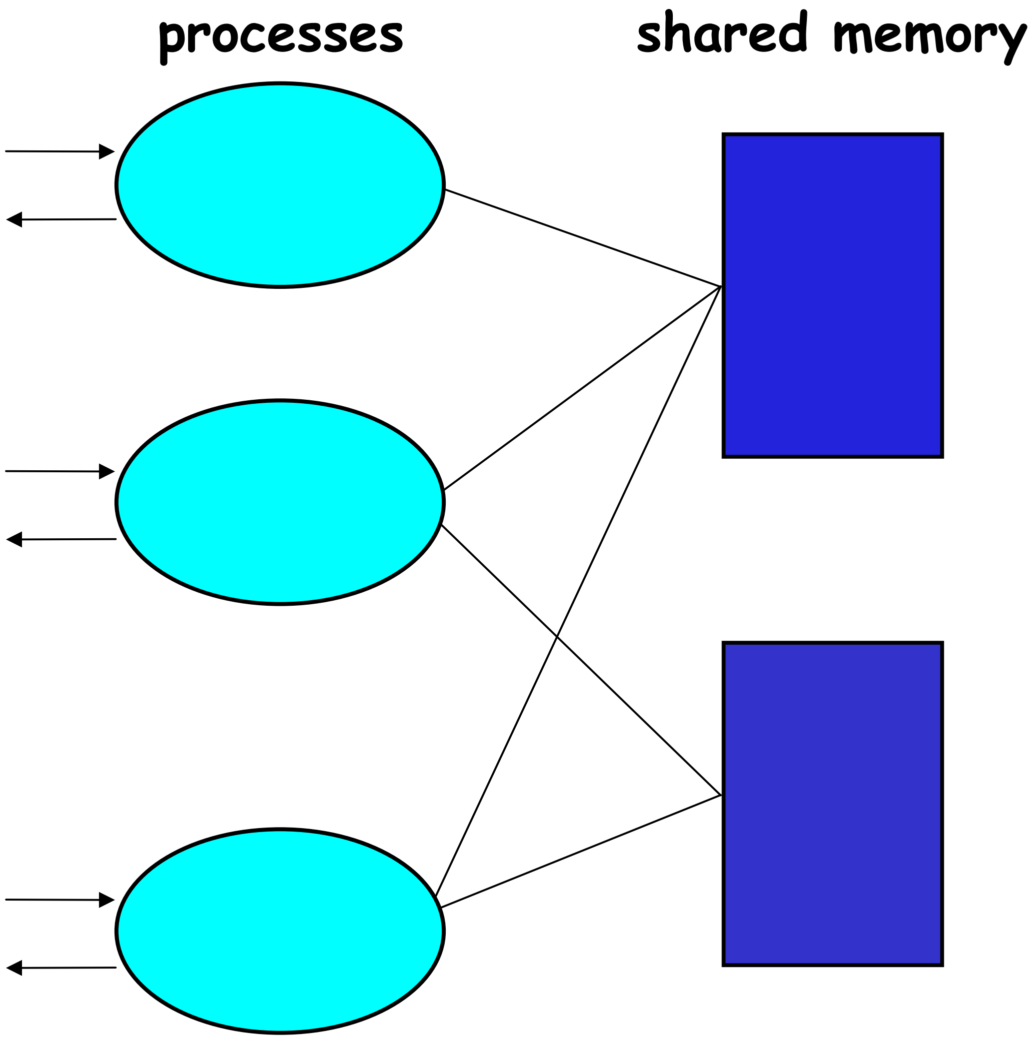
- Dual of asynchronous network model

  can freely translate between them

- A simpler model about which to reason
- Further simplification

  ignore faults

Shared variables in shared memory



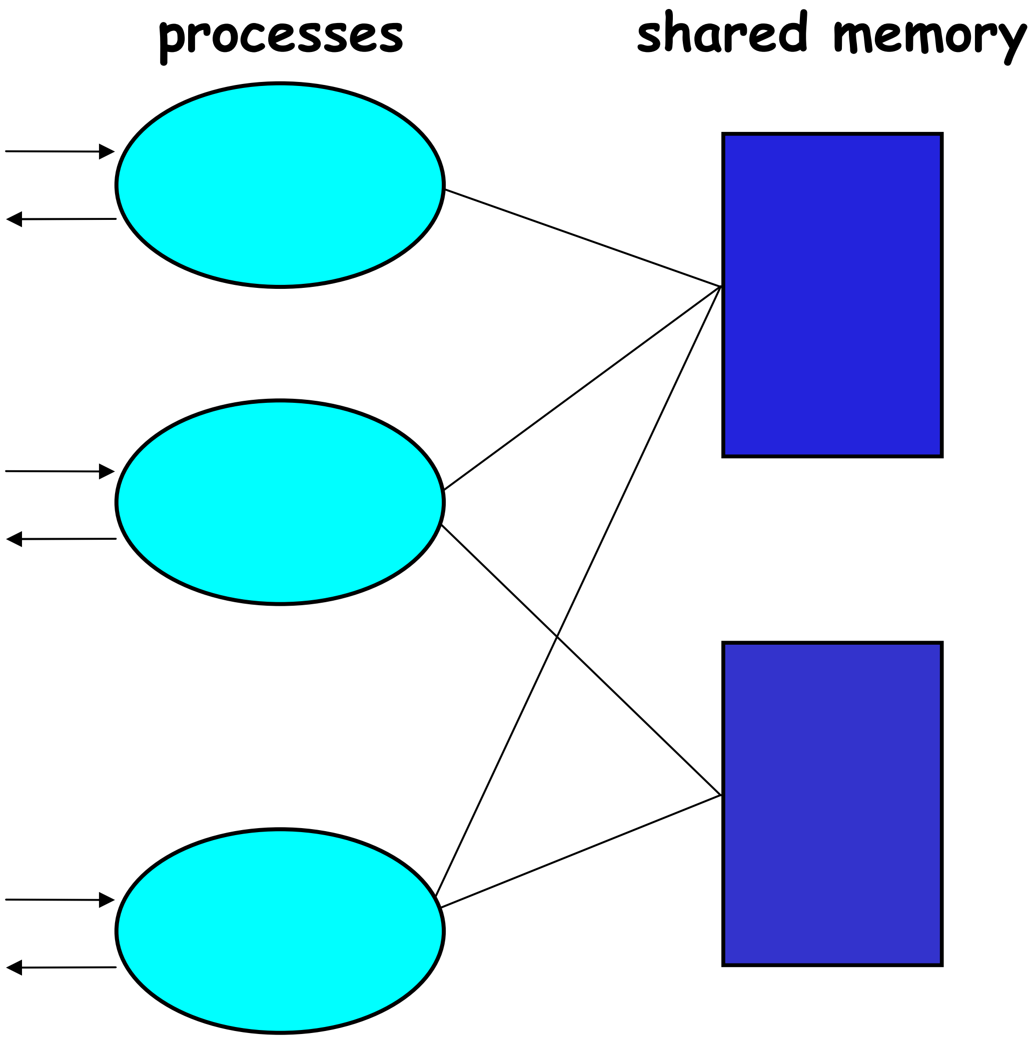
- Each shared variable is defined by:

  a set of *values*

  an *initial* value

  read and/or write *capabilities* available to processes

Process interaction



- Each process is defined by:

  a set of *states*

  a *start* state

  a state and variable *transition* function

  that is, an algorithm represented as a *state machine*

Mutual exclusion for two processes

- The basic model

  using asynchronous shared memory, two processes, p_1 and p_2, compete for access to their critical regions (i.e., access to shared resource)

- The desired outcome (properties)

  mutual exclusion (*safety*)

  freedom from deadlock (*safety*)

  freedom from starvation (*liveness*)

Mutual exclusion, informally

- When process p_i wants to enter its critical region C, it sets a flag f_i
- If the other process's flag is set, p_i does not enter C, but instead resets f_i and tries again
- If the other process's flag is not set, p_i enters C
- p_i resets f_i after leaving C

Code skeleton

shared variables: f(i:1..2):Boolean, initially false, writable by p_i, readable by all

```
process p_i:
try:
      f(i) := true
      if (f((i%2)+1)) then {
            f(i) := false; go to try}
      else { enter/use/exit critical region }
      f(i) := false
```

T (try)

C (critical)

E (exit)

Mutual exclusion and deadlock

- What is an example of an error in the algorithm that would lead to...

    violation of deadlock freedom?

    violation of mutual exclusion?

Safety property: mutual exclusion

- Need to argue that it is not possible for p_1 and p_2 to both be in critical region C at the same time
- Proof sketch (by contradiction)

  assume p_1 and p_2 both in C, so f_1 and f_2 must be set and remain set from before C to E

  if p_1 set flag first…

    p_2 must remain in T and cannot enter C (same for p_2 and p_1)

    hence, contradiction

  if p_1 and p_2 set flags "simultaneously"…

    then both remain in T and neither can enter C

    hence, contradiction

What about starvation?

- Recall: freedom from starvation is a liveness property

  any process that reaches *T* eventually enters *C*

- Recall: processes modeled as state machines

  we might assume "fair choice"

    if a choice over a set of transitions is executed infinitely often, then every transition in the set is executed infinitely often

Modeling “fairness”

- *Strong fairness* (a form of fair choice)

  for every transition, if it is enabled infinitely often, it is taken infinitely often

- *Weak fairness*

  for every transition, if it is enabled continuously from some point on, it is taken infinitely often

- *No fairness*

  for every transition, even if enabled continuously from some point, it may not be taken unless it is the only choice

Which fairness should we use?

- A universal design principle

  for generality, impose the least constraint

  no fairness $<$ weak fairness $<$ strong fairness

- In practice, fairness derives from scheduling

  local process scheduler usually ensures that every process has an opportunity to execute, if enabled

  a process is enabled if any of its transitions is enabled

Fair mutual exclusion, informally

- When process p_i wants to enter its critical region C, it sets a flag f_i and stores i in shared variable *turn*
- While the flag of the other process is set and *turn* has value i, p_i does not enter, but instead spins
- If the flag of the other process is not set or *turn* has value other than i, p_i enters C
- p_i resets f_i after leaving C

Code skeleton (Peterson 2P)

shared variables: turn:1..2, initially undef, read/write by all; f(i:1..2):Boolean, initially false, writable by p_i, readable by all

process p_i:

```
      f(i) := true
      turn := i
      while (f((i%2)+1) && turn = i)
            {null}
      enter/use/exit critical region
      f(i) := false
```

T (try)

C (critical)

E (exit)

Safety property: mutual exclusion

- Need to argue that not possible for p_1 and p_2 to both be in critical region C at same time
- Proof sketch

  assume p_1 tries to enter C, so f_1 set and $turn = 1$

  if p_2 not competing, f_2 not set, so p_1 can enter C

  if p_2 is competing, f_2 is set, so entry depends on $turn$

    assume p_1 is first, so p_2 sets $turn = 2$, which allows p_1 but blocks p_2

Liveness property: starvation free

- Need to argue that any process reaching *T* eventually enters *C*
- Proof sketch

  if p_1 is waiting in *T*, it will be given priority through *turn* when p_2 exits from *C*

  if p_2 competes, it will give priority to p_1 by setting turn to 2
- What level of fairness is supported?

Fair mutual exclusion, informally

- How can we generalize from 2 processes to *n*?
- Use Peterson 2P iteratively in a series of *n*-1 competitions, each with its own variable *turn*
- At each competition level *k*, Peterson 2P ensures at least one loser *i* whose $turn_k$ is *i* if all compete

    at level 1, at most *n*-1 processes proceed

    at level 2, at most *n*-2 processes proceed

    at level *n*-1, at most 1 process proceeds

Code skeleton (Peterson nP)

shared variables: turn(k:1..n-1):1..n, initially undef, read/write by all; f(i:1..n):1..n-1, initially 0, writable by p_i, readable by all

process p_i:

```
for k=1 to n-1 do {
        f(i) := k
        turn(k) := i
        while ((∃j!=i:f(j)>=k) && turn(k) = i) {null}
}
enter/use/exit critical region
f(i) := 0
```

T (try)

C (critical)

E (exit)

Suitability for network environment

Q: What is a simple way to make these mutual exclusion algorithms suitable for use in a network environment?

i.e., how do we simulate shared memory?

A: Shared data can be encapsulated in processes that support communication via message passing rather than read/write