**Paradigms**

- Iterative parallelism

- Recursive parallelism

- Producers and consumers

- Clients and servers

- Interacting peers

**Embarrassingly Parallel**

- **Read set**: set of variables read by a process

-**Write set:** set of variables written to by a process

- Two operations are **independent** if the write set of each is disjoint from both the read and write sets of the other.

**Independent procedure calls**

- If a procedure does not reference global variables and has only value parameters, then every call of the procedure will be independent.

**Semantics of concurrent programs**

- **State:** the values of all the variables at a point in time.

- **Atomic action**: Action which indivisibly examine or change state.

- **History or interleaving or trace**: A particular sequence of the atomic actions of a program. Results in a particular sequence of states: s0 -> s1 -> s3 … -> sn

- **Critical section**: A section that cannot be interleaved with other actions in other processes that reference the same variables.

- **Property**: Something which is true of every possible history of a program.

**Safety**: nothing bad ever happens.

- Final state is correct.

- Mutual exclusion.

- No deadlock.

**Liveness**: something good eventually happens.

- Program terminates

- Process eventually enters critical section.

- A request for service will eventually be honored.

- A message will reach its destination.

**Partial correctness**: final state is correct, assuming the program terminates. A safety property.

-**Termination:** every loop and procedure call terminates. All histories are finite. A liveness property

-**Total correctness**: partially correct and terminates.­

**At-Most-Once Property**

- Assume a critical reference is to a simple variable.­

- An assignment x = e is at-most-once if either

+ e contains at most one critical reference and x is not read by another process

+ e contains no critical references

- Such an assignment statement will appear to be atomic.

**Examples:**

**-----------------------------------------------------------------------------------**

No critical references:

int x = 0, y = 0;

co x = x+1; // y = y+1; oc;

x and y are both 1

One critical reference and one lhs read by another:

int x = 0, y = 0;

co x = y+1; // y = y+1; oc;

x is 1 or 2, y is 1

Neither satisfies at-most-once:

int x = 0, y = 0;

co x = y+1; // y = x+1; oc;

Values could be 1 and 2, 2 and 1, or even 1 and 1

**The Await Statement**

<await (B) S;>

B specifies a delay condition.

S is a sequence of statements guaranteed to terminate.

Atomic actions are specified by angle brackets, < and >

Therefore B is guaranteed to be true when execution of S begins.

No internal state in S is visible to other processes.

**Atomic actions**

- An **unconditional atomic** action is one that does not contain a delay condition B.

- A **conditional atomic** action is an await statement with a guard B.

- If B is false, it can only become true as the result of actions by other processes.

**Proof strategies**

**Conditional proof:** To prove something like X => Y

- Assume X is true on one line of a proof.

- Show that Y follows on some later line.

- Box up those lines to show that you're done.

**Indirect proof:** To prove something like X:

- Assume (not) X on one line of a proof.

- Show that a contradiction (Z and (not) Z) shows up on later lines.

- Box up those lines to show that you're done.

**Programming Logic**

{P} S {Q} is a called a triple

- If P is true before S is executed, and S terminates, then Q is true after S executes.

- P and Q are assertions

- P is the precondition

- Q is the post condition

**Example:**

Prove: {x < 5} x = x + 1 {x < 7}

Axiom: {Px\t} x = t {P}

1: {x + 1 < 7} x = x + 1 {x < 7} Axiom

2: x < 5 Assumption

3: x + 1 < 6 2, arithmetic

4: 6 < 7 arithmetic

5: x + 1 < 7 3,4, transitive

6: (x < 5) => (x + 1 < 7) 2,5, cond proof

7: {x < 5} x = x + 1 {x < 7} 1,6, consequence

**Definition of Noninterference**

- An **assignment action** is an assignment statement or an await statement that contains one or more assignments.

- A **critical assertion** is a precondition or post condition that is not within an await statement.

- Rename all local variables in all processes so that no two processes have the same local variable names.

- Let **a** be an assignment action in one process and let pre(**a**) be

It’s precondition. Let **C** be a critical assertion in another process.

Then **a** does **not interfere** with **C** if the following is a theorem:

**{C ^ pre(a)} a {C}**

- **C** is invariant with respect to assignment **a**

**techniques for avoiding interference**

- Many statements and parts are **disjoint**.

- **Weakened assertion:** one that admits more program states than another assertion that might be true of a process in isolation

- **Global invariant** (PC)

- **Synchronization** (using await)

**Scheduling and Fairness**

- **Fairness:** each process gets a chance to proceed.

- An atomic action is **eligible** if it is the next one in the process that could be executed.

- A **scheduling policy** determines which eligible action will be executed next.

**Unconditional fairness**: if every unconditional atomic action that is eligible is executed eventually.

**Weak fairness:**

1. It is unconditionally fair, and

2. Every conditional atomic action that is eligible is executed eventually, assuming that its condition becomes true and then remains true until it is seen by the process executing the conditional atomic action.

- Round-robin and time slicing are weakly fair.

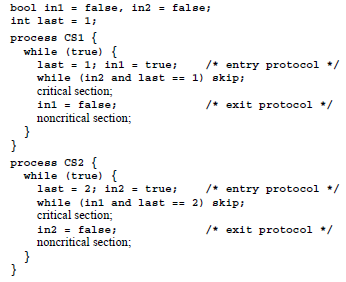
- Not sufficient to guarantee that any eligible await statement eventually executes

**Strong fairness:**

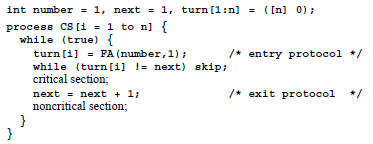
2. Every conditional atomic action that is eligible is executed eventually, assumingthat its condition is infinitely often true.

**Critical Sections: Fair Solutions**

**Tie breaker algorithm**



**Ticket algorithm**



**Semaphores**

|  |  |
| --- | --- |
| **Decrement** | **Increment** |
| P | V |
| Wait | Signal |
| Wait | Post |
| Acquire | Release |
| Get | Release |

**Flag Synchronization Principles:**

• The process that waits for a synchronization flag to be set is the one that should clear

that flag.

• A flag should not be set until it is known that it is clear

**Single Instruction Multiple Data**

• Every processor executes exactly the same instructions in lock step.

• Barriers not needed since all finish before looping.

• Every process fetches old sum before writing new one.

• Parallel assignments thus appear to be atomic.

• if statements always take the maximum time.

