Terminology

state — the value of all program variables (including implicit, e.g., program counter).

– Assume independent registers for each process.

atomic action — indivisible program step (examine or change state).

history (a.k.a. trace) — sequence of states representing execution of concurrent program (transitions are atomic actions).

– Execution of concurrent program results in interleaving of actions executed by each process.
– Program describes a (huge) set of possible histories.
– Synchronization constrains the possible histories.

property — an attribute that is true of all possible histories of a program.

– *safety property* — program never enters a particular (bad) state (e.g, mutual exclusion).
– *liveness property* — program eventually enters a particular state (e.g., termination).

partial correctness — the final state is correct, assuming termination.

total correctness — partial correctness and guaranteed termination.

Independence

Read set — the set of variables an operation (part of a program) reads, but does not alter.

Write set — the set of variables an operation changes the value of (and may read).

(By variable, we mean any value that is written or read atomically.)

Two parts of a program are *independent* if the write set of each is disjoint from both the read and write sets of the other part.

If program parts are independent, then they're candidates for concurrent execution.

Example: Searching in a file

```
string line[2];
int r = 0;
read line of input into line[0];
while (!EOF) {
  co look for pattern in line[r];
    if (pattern is in line[r])
      write line[r];
  // read line of input into line[(r+1)%2]
  oc
  r = (r + 1) % 2;
}
```

- Two parallel tasks are independent so this is equivalent to program where they are done sequentially.
- This pattern is called “co inside while”.

- To reduce overhead (process creation) we can transform to "while inside co" — requires synchronization.

```
string line[2];
bool full[2] = { false }; # true if line[i] is full
bool done = false;
co # process 1: check for pattern
  int l = 0; # the line to search in
  while (!done) {
    wait for full[l] or done;
    if (done) break;
    look for pattern in line[l];
    if (pattern is in line[l])
      write line[l];
    full[l] = false; # done with this line
    l = (l+1) % 2;
  }
// # process 2: read next line
```



```
  int r = 0; # line to read into
  while (!EOF) {
    wait for !full[r]; # wait for line[r] to be checked
    read next line into line[r];
    full[r] = true; # signal that line[r] is full
    r = (r+1) % 2;
  }
  done = true;
oc
}
```


Atomicity

fine-grained atomicity — implemented directly by the hardware (e.g., variable read/write).

(Assume private registers and stack per process.)

`x = ` *e* will appear to be atomic if *e* doesn't reference any variable changed by another process.

```
int y = 0, z = 0;
co x = y+z; // y = 1; z = 2; oc;
```

```
int x = 0, y = 0;
co x = y+1; // y = y+1; oc;
```


At Most Once Property

critical reference — a reference in an expression to a variable that is changed by another process.

`x = ` *e* satisfies the *At Most Once* property if either:

1) *e* contains at most one critical reference and `x` is not read by another process, or

2) *e* contains no critical references.

Assignments satisfying AMO will appear to be atomic.

For expressions that are not assignment statements, AMO is satisfied if it contains no more than one critical reference.

Coarse-Grained Atomicity — Synchronization

To describe one or more statements that execute atomically:

$\langle$`await (B) S;`$\rangle$

`B` is a condition (no side effects),
`S` is a statement block (one or more statements), that is guaranteed to terminate.

- Will not execute until `B` is true (conditional synchronization).
- No parts of `S` may be interleaved with statements from other processes.
- May not be efficiently implemented in all cases, but useful for describing algorithms.

Special cases

Mutual Exclusion — `B` = "true", so it is omitted: $\langle$`S;`$\rangle$

Conditional Synchronization — `S` is empty, so it is omitted: $\langle$`await (B)`$\rangle$

- If `B` satisfies AMO, can be implemented by *spin loop*:
  `while (!B) ;`

Example: Producer/Consumer

Copy a[n] into b[n], using buf.

Synchronization requirement: $c \leq p \leq c + 1$

```
int buf, p = 0, c = 0;

process Producer {
  int a[n];
  while (p < n) {
    < await (p == c); >
    buf = a[p];
    p++;
  }
}

process Consumer {
  int b[n];
  while (c < n) {
    < await ( p > c); >
    b[c] = buf;
```


Axiomatic Semantics

Axioms: A distinguished set of formulae that are assumed to be true.

Inference rule: $\dfrac{\mathrm{H}_1, \mathrm{H}_2, \ldots, \mathrm{H}_n}{\mathrm{C}}$

If all of H_i (the *hypotheses*) are true, then we can infer that C (the *conclusion*) is true.

Proof: Sequence of lines, each of which is an axiom or can be derived from previous lines by inference rules.

Theorem: A line in a proof.

Interpretation: Maps each formula to true of false.

Soundness: (w.r.t. an interpretation)

- Axiom is sound iff it is true.
- Inference rule is sound iff its conclusion is true assuming all the hypotheses are true.
- Logic is sound iff all axioms and inference rules are sound. (The interpretation is a *model* for the logic.)

Completeness: A logic is complete (w.r.t. an interpretation) iff any formula that is true is a theorem (i.e., can be proven in the logic).

Gödel's incompleteness theorem: Any logic that includes arithmetic cannot be complete.

Programming Logic

- Formula are (Hoare) *triples* of the form $\{\mathrm{P}\}\ \mathtt{S}\ \{\mathrm{Q}\}$
- P and Q are predicates referring to the values of program variables in $\mathtt{S}$ (assertions).
- $\mathtt{S}$ is one or more program statements.
- Interpretation: $\{\mathrm{P}\}\ \mathtt{S}\ \{\mathrm{Q}\}$ is true iff, whenever execution of $\mathtt{S}$ starts in a state satisfying P and execution of $\mathtt{S}$ terminates, the resulting state satisfies Q. (partial correctness)
- P is called a *pre-condition*
- Q is called a *post-condition*

Axioms/Rules for Sequential Programs

Assignment Axiom: $\{\mathrm{P}_{\mathtt{x}\leftarrow e}\}\ \mathtt{x = }\ e\ \{\mathrm{P}\}$

Composition Rule: $\dfrac{\{\mathrm{P}\}\ \mathtt{S}_1\ \{\mathrm{Q}\},\ \{\mathrm{Q}\}\ \mathtt{S}_2\ \{\mathrm{R}\}}{\{\mathrm{P}\}\ \mathtt{S}_1;\mathtt{S}_2\ \{\mathrm{R}\}}$

If Statement Rule: $\dfrac{\{\mathrm{P}\wedge\mathrm{B}\}\ \mathtt{S}\ \{\mathrm{Q}\},\ (\mathrm{P}\wedge\neg\mathrm{B})\Rightarrow\mathrm{Q}}{\{\mathrm{P}\}\ \mathtt{if\ (B)\ S;}\ \{\mathrm{Q}\}}$

While Statement Rule: $\dfrac{\{\mathrm{I}\wedge\mathrm{B}\}\ \mathtt{S}\ \{\mathrm{I}\}}{\{\mathrm{I}\}\ \mathtt{while\ (B)\ S;}\ \{\mathrm{I}\wedge\neg\mathrm{B}\}}$

Rule of Consequence: $\dfrac{\mathrm{P}'\Rightarrow\mathrm{P},\ \{\mathrm{P}\}\ \mathtt{S}\ \{\mathrm{Q}\},\ \mathrm{Q}\Rightarrow\mathrm{Q}',}{\{\mathrm{P}'\}\ \mathtt{S}\ \{\mathrm{Q}'\}}$

Inference for Concurrent Execution

Await Statement Rule: $\dfrac{\{\mathrm{P}\wedge\mathrm{B}\}\ \mathtt{S}\ \{\mathrm{Q}\},}{\{\mathrm{P}\}\ \mathtt{<\ await\ (B)\ S;\ >}\ \{\mathrm{Q}\}}$

Co Statement Rule:

$$\dfrac{\{\mathrm{P}_i\}\mathtt{S}_i\{\mathrm{Q}_i\}\text{ are interference free}}{\{\mathrm{P}_1\wedge\ldots\wedge\mathrm{P}_n\}\ \mathtt{co}\ \mathtt{S}_1;\ \mathtt{//}\ \ldots\ \mathtt{//}\ \mathtt{S}_n;\ \mathtt{oc}\ \{\mathrm{Q}_1\wedge\ldots\wedge\mathrm{Q}_n\}}$$

Interference

- A process *interferes* with another process if it invalidates an assertion in the other process.
- *assignment action* — an assignment statement or an `await` statement containing one or more assignment statements.
- *critical assertion* — a pre-condition or post-condition not within an `await` statement.

Consider assignment action a, it's pre-condition $pre(\mathtt{a})$, and a critical assertion C, from another process.

If $\{\mathrm{C} \wedge pre(\mathtt{a})\}\mathtt{a}\{\mathrm{C}\}$, then a and C are non-interfering.

i.e., if executing a doesn't change the truth of C.

Techniques for avoiding interference

Disjoint variables — Write set of one process is disjoint from *reference set* (variables in the assertions) of the other process.

Weakened assertions — Take into account effects of concurrent execution. Example:

```
## {x == 0}
co ## { x == 0 \/ x == 2 }
  < x = x + 1; >
  ## { x == 1 \/ x == 3 }
// ## { x == 1 \/ x == 1 }
  < x = x + 2; >
  ## { x == 2 \/ x == 3 }
oc
## { x == 3 }
```


Global invariants — true initially, and preserved by all assignment actions.

- Assertions in proof of each process, $\mathtt{P}_j$, in form $\mathrm{I} \wedge \mathrm{L}$, where
  - I is global invariant
  - L references only local variables in $\mathtt{P}_j$ or global variables that $\mathtt{P}_j$ is the only process to assign to.
- Proofs are interference free.
- Good general technique for concurrent program design.

Synchronization — combining sequences of statements into `await` statements:

- ignore effects of individual statements w.r.t. interference,
- internal assertions can't be interfered with.
- Two techniques:
  1) 'Hide' assertions via mutual exclusion.
  2) Strengthen pre-condition via conditional synchronization.

Safety Properties

A *property* characterizes a set of executions.

A program has (or *satisfies*) a property if every possible execution (history) of the program is in the set.

Safety property: Something must always be true (set of executions in which no undesirable states, or sequences of states, occur).

- e.g.,
  - partial correctness (don't terminate in invalid state)
  - absence of deadlock
  - mutual exclusion
- finitely refutable: violations occur at an instant
- characterized by negation of 'bad' things

Proving Safety

Let B characterize undesirable (sequences of) states

- Show that for any critical assertion C, $C \Rightarrow \neg B$, or
- Show that $\neg B$ is a global invariant.
  - $\neg B$ is *true* initially,
  - $\{\neg B\}$ S $\{\neg B\}$ is *true* for all program statements S

Special Case: Exclusion of Configurations

```
co # process 1
  ... ; { preS1 } S1; ...
//  # process 2
  ... ; { preS2 } S2; ...
oc
```

If

- $\mathrm{preS1}$ and $\mathrm{preS2}$ are not interfered with, and
- $\mathrm{preS1} \wedge \mathrm{preS2} \equiv \mathit{false}$

then the state $\mathrm{preS1} \wedge \mathrm{preS2}$ is impossible.

Liveness Properties

Something must eventually become true (set of executions which all contain some state, or sequence of states).

- e.g.,
  - termination: process must eventually stop
  - absence of starvation (processes must eventually get serviced)
- non finitely refutable: any execution can be extended to satisfy the property.

Fairness

An atomic action is *eligible* if it's the next atomic action in a process that could be executed.

scheduling policy — determines which eligible action will be executed next.

```
bool continue = true;
co while (continue) ;
// continue = false;
oc
```

Degrees of fairness:

unconditional: Every unconditional atomic action that is eligible is executed eventually.

weak: Unconditionally fair & every conditional atomic action for which the condition is continuously true (until it is executed), will eventually be executed.

strong: Unconditionally fair & every conditional atomic action for which the condition is true infinitely often, will eventually be executed.

```
bool continue = true, try = false;

co while (continue) { try = true; try = false; }
// < await (try) continue = false; >
oc
```