Processes and Synchronization

Outline

- Basic concepts
- Atomicity
- Axiomatic semantics
- Safety, Liveness, and Fairness

Terminology

state — the value of all program variables

- $(x:0, y:0)$, $(x:0, y:1)$, $(x:1, y:0)$, ...

atomic action — indivisible program step

- Each atomic action represents a set of pairs of states.
- $\langle x := x+1 \rangle$:

$$\begin{aligned}\{ \quad & ((x:0, y:0), (x:1, y:0)), \\ & ((x:1, y:0), (x:2, y:0)), \ldots\}\end{aligned}$$

history (a.k.a. trace) — a possibly infinite sequence of states

- $\langle (x:0, y:0), (x:1, y:0), (x:1, y:1) \rangle$

programs — describe a set of histories.

- History of concurrent program results from interleaving of the actions of each process.
- $(\mathbf{co}\ \langle x := x+1; \rangle \langle x := x+1 \rangle\ //\ \langle y := x+1; \rangle\ \mathbf{co})$:

$$\begin{aligned}\{ \quad & \langle (x:0, y:0), (x:1, y:0), (x:2, y:0), (x:2, y:3) \rangle, \\ & \langle (x:0, y:0), (x:1, y:0), (x:1, y:2), (x:2, y:2) \rangle, \\ & \langle (x:0, y:1), (x:0, y:1), (x:1, y:1), (x:2, y:1) \rangle, \ldots \}\end{aligned}$$

Properties

property — a boolean function of histories

- A property holds for a program if it holds for all possible histories of the program.
- **safety property** — something bad never happens
  - E.g. mutual exclusion.
  - Safety properties can only be refuted by finite histories
  - I.e. if a safety property does not hold for a history, there is some point in the history at which we can tell it does not hold without needing to see any of the history past that point.
- **liveness property** — something good eventually happens
  - E.g. responsiveness
  - Liveness properties can only be refuted by infinite histories.
  - I.e. Any finite history can always be extended in some way so that the liveness property holds.

For sequential programs we are particularly interested in two kinds of properties.

- **partial correctness** — the program can not terminate in an unsatisfactory state (safety)
- **termination** — program eventually terminates (liveness)
- **total correctness** — partial correctness and termination: A combination of a safety and a liveness property.
- Note that all properties that are neither safety nor liveness properties can be expressed as a combination of safety and liveness properties. (Alpern & Schneider, 'Defining liveness', IPL 21 (1985) 181–185).

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.

$$W_a \cap (R_b \cup W_b) = \emptyset \quad \wedge \quad W_b \cap (R_a \cup W_a) = \emptyset$$

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; }
```

Note that the two parallel tasks are independent and thus this is equivalent to a program where they are done sequentially.

This pattern is called “co inside while”.

To reduce process creation overhead can transform to “while inside co”.

But then we must synchronize the processes.

```
string line[2];
bool full[2] := { false }, done := false;
# for all i, full[i] iff line[i] has an unsearched line
co # process 1: check for pattern
        int l := 0; # the line to search in
        loop {
                wait for full[l] or done;
        exit when( done )
                look for pattern in line[l];
                if (pattern is in line[l]) write line[l];
                full[l] := false;
                l := (l+1) % 2; }
// # process 2: read next line
        int r = 0; # line to read into
        while (!EOF) {
                wait for !full[r];
                read next line into line[r];
                full[r] := true; # line[r] is full
                r := (r+1) % 2; }
        done := true;
oc }
```

At Most Once Property

For variables that fit in one memory word, we can assume that reads and writes are atomic.

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

An expression E satisfies the *At Most Once property* iff

- E contains at most one critical reference.

An assignment $x := E$ 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.

Expressions and assignments satisfying AMO will appear to be atomic.

Why?

- An expression or assignment consists of a sequence of reads possibly followed by a single write.
- If the AMO property is obeyed, only one of these reads and writes is to a critical variable.
- All the other reads and writes can be moved without affecting (or being affected by) other processes.
- We can move all the reads and writes of an AMO expression or statement together without affecting any process.

Example

$w, z := 42, 13;$ **co** $y := z + 1;$ **//** $y := w + 1;$ **oc**

P1:	R(z)13		W(y)14	
P2:		R(w)42		W(y)43

We can imagine interchanging the order of the 2nd and 3rd actions

P1:	R(z)13	W(y)14		
P2:			R(w)42	W(y)43

Example

$w, y := 2, 2;$ **co** $x := w + y;$ // $y := 10;$ **oc**

$$\begin{array}{lllll}
\text{P1:} & \text{R(w)2} & \boxed{\text{R(y)2}} & & \text{W(x)4} \\
\text{P2:} & & & \text{W(y)10} &
\end{array}$$

Because x is not read by another process, we can move the time of the write to x.

$$\begin{array}{lllll}
\text{P1:} & \text{R(w)2} & \boxed{\text{R(y)2}} & \text{W(x)4} & \\
\text{P2:} & & & & \text{W(y)10}
\end{array}$$

Example

$w, y := 2, 2;$ **co** $x := w + y;$ // $y := 10; x := 13;$ **oc**

$$\begin{array}{llllll}
\text{P1:} & \text{R(w)2} & & \boxed{\text{R(y)10}} & & \text{W(x)12} \\
\text{P2:} & & \boxed{\text{W(y)10}} & & \text{W(x)13} &
\end{array}$$

Three swaps move the assignment together.

$$\begin{array}{llllll}
\text{P1:} & & & \text{R(w)2} & \boxed{\text{R(y)10}} & \text{W(x)12} \\
\text{P2:} & \boxed{\text{W(y)10}} & \text{W(x)13} & & &
\end{array}$$

Await Statements

To describe one or more statements that execute atomically:

$\langle \mathbf{await}\ (E)\ S \rangle$

E is a condition (no side effects),

S is a statement block (one or more statements), that is guaranteed to terminate.

- Will not execute until E is true.
- No parts of S may be interleaved with statements from other processes.
- Useful for describing algorithms. Later we'll look at implementation.

Abbreviations for special cases

Mutual exclusion: await $(true)$ can be omitted:

$$\langle \mathbf{await}\ (true)\ S \rangle = \langle S \rangle$$

Conditional synchronization. S is **skip** it may be omitted:

$$\langle \mathbf{await}\ (E)\ \mathbf{skip} \rangle = \langle \mathbf{await}\ (E) \rangle$$

- If E satisfies AMO, it can be implemented by *spin loop*:

$$\langle \mathbf{await}\ (E) \rangle \approx \mathbf{while}(\mathrm{not}\ E)\ \mathbf{skip}$$

Example of await: Producer/Consumer

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

Global invariant: $0 \le c \le p \le c+1 \le n+1$

Therefore $p = c$ or $p = c + 1$.

Global invariant: $p = c + 1 \Rightarrow buf = a[c]$

```
**int** $buf, p := 0, c := 0$ ;
```

```
process Producer {
        int $a[n]$;
        while( $p < n$ ) {

            ⟨await( $p = c$ );⟩
             $buf := a[p]$;
             $p := p + 1$; } }
```

```
process Consumer {
        int $b[n]$ ;
        while( $c < n$ ) {

            ⟨await( $p > c$) ;⟩
             $b[c] := buf$;
             $c := c + 1$; } }
```

Can we use spin loops?

Can you prove the invariants?

Proof Systems

An axiom system describes a set of theorems by means of a set of axioms and inference rules.

- **Axioms:** A distinguished set of formulae that are assumed to be theorems.
- **Inference rules:** $\dfrac{H_1, H_2, \ldots, H_n}{C}$

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

(Hilbert-Style) 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.

I'll use the notation $\vdash P$ to indicate that P is a theorem.

Hoare Logic for Sequential Programming

Hoare Triples

- Formulae are (Hoare) *triples* of the form $\{P\}\ S\ \{Q\}$
- P and Q are *conditions* referring to the values of program variables in S and other variables.
- S is one or more program statements.
- Interpretation: $\{P\}\ S\ \{Q\}$ is true iff, whenever execution of S starts in a state satisfying P, every state it could terminate in satisfies Q. (partial correctness)
- P is called the *precondition*
- Q is called the *postcondition*

Examples of valid Hoare triples

$$\{x == 23\}\ x := x + 1;\ \{x == 24\}$$

$$\{0 \leq x < 10\}\ x := x + 1;\ \{1 \leq x < 11\}$$

$$\{x == X\}\ x := x + 1;\ \{x == X + 1\}$$

$$\{(x+1)^2 + (x+1) + 9 < 20\}\ x := x + 1;\ \{x^2 + x + 9 < 20\}$$

$$\{(x+1)^2 + (x+1) + 9 < 10\}\ x := x + 1;\ \{x^2 + x + 9 < 20\}$$

$$\{x == x_0\}\ x := x + 2;\ y := x/2;\ \{y == x_0/2 + 1\}$$

$$\{x == 0\}\ \mathbf{while}(\ \mathit{true}\ )\ x := x + 1\ \{x == \pi\}$$

$\{true\}$
$x := 0;$
$s := 0;$
$\textbf{while}(\ x \neq N)\ \{\ s+=A[x];\ x+=1;\}$
$\{s == \sum_{i=0}^{N-1} A[i]\}$

Examples of invalid Hoare triples

$$\{0 \leq x < 10\}\ x := x + 1;\ \{1 \leq x < 10\}$$

Axioms/Rules for Sequential Programs

The basic rule for assignments is that the precondition must imply the *substituted postcondition*

$$\frac{\vdash P \Rightarrow Q_{x \leftarrow E}}{\vdash \{P\}\ x := E\ \{Q\}}\text{(Assign)}$$

We can extend this to other programming constructs.

For simultaineous assignment to 2 variables:

$$\frac{\vdash P \Rightarrow Q_{x,y \leftarrow E,F}}{\vdash \{P\}\ x, y := E, F\ \{Q\}}\text{(Assign)}$$

For skip

$$\frac{\vdash P \Rightarrow Q}{\vdash \{P\}\ \mathbf{skip}\ \{Q\}}\text{(Assign)}$$

For sequential composition we have

$$\frac{\begin{array}{c}\vdash \{P\}\ S\ \{Q\} \\ \vdash \{Q\}\ T\ \{R\}\end{array}}{\vdash \{P\}\ S\,T\ \{R\}}\text{(Seq)}$$

For iteration we have

$$\frac{\begin{array}{c}\vdash \{P \wedge E\}\ S\ \{P\} \\ \vdash P \wedge \neg E \Rightarrow Q\end{array}}{\vdash \{P\}\ \mathbf{while}(E)\ S\ \{Q\}}\text{(While)}$$

(P here is called a *loop invariant*)

For if statements we have

$$\frac{\begin{array}{c}\vdash \{P \wedge E\}\ S\ \{Q\} \\ \vdash \{P \wedge \neg E\}\ T\ \{Q\}\end{array}}{\vdash \{P\}\ \mathbf{if}(E)\, S\, \mathbf{else}\, T\ \{Q\}}\text{(if)}$$

An example proof

Here is part of a Hilbert style proof. (Omitting the proofs of the various implications.)

10	$x \neq N \wedge 0 \leq x \leq N \wedge s = \sum_{i=0}^{x-1} A[i]$ $\Rightarrow 0 \leq x+1 \leq N \wedge s + A[x] = \sum_{i=0}^{x} A[i]$	...
11	$0 \leq x+1 \leq N \wedge s = \sum_{i=0}^{x} A[i]$ $\Rightarrow 0 \leq x+1 \leq N \wedge s = \sum_{i=0}^{x+1-1} A[i]$	...
12	$x = N \wedge 0 \leq x \leq N \wedge s = \sum_{i=0}^{x-1} A[i]$ $\Rightarrow s = \sum_{i=0}^{N-1} A[i]$	...
13	$\{x \neq N \wedge 0 \leq x \leq N \wedge s = \sum_{i=0}^{x-1} A[i]\}$ $s := s + A[x];$ $\{0 \leq x+1 \leq N \wedge s = \sum_{i=0}^{x} A[i]\}$	Assign(10)
14	$\{0 \leq x+1 \leq N \wedge s = \sum_{i=0}^{x} A[i]\}$ $x := x + 1;$ $\{0 \leq x \leq N \wedge s = \sum_{i=0}^{x-1} A[i]\}$	Assign(11)
15	$\{x \neq N \wedge 0 \leq x \leq N \wedge s = \sum_{i=0}^{x-1} A[i]\}$ $s := s + A[x];\ x := x + 1;$ $\{0 \leq x \leq N \wedge s = \sum_{i=0}^{x-1} A[i]\}$	Seq. (13,14)
16	$\{0 \leq x \leq N \wedge s = \sum_{i=0}^{x-1} A[i]\}$ $\mathbf{while}(\ x \neq N)\ \{\ s+ = A[x];\ x+ = 1; \}$ $\{s = \sum_{i=0}^{N-1} A[i]\}$	While (15,12)

Proof Outlines

A proof outline summarizes the assertions used in a proof.

```
{$\mathit{true}$}
$x := 0$ ;
{$x = 0$}
$s := 0$ ;
{$\mathit{Loop\ Inv} : 0 \leq x \leq N \wedge s = \sum_{i=0}^{x-1} A[i]$}
while( $x \neq N$ ) {
        {$x \neq N \wedge \mathit{Loop\ Inv}$}
        $s := s + A[x];$
        {$0 \leq x + 1 \leq N \wedge s = \sum_{i=0}^{x} A[i]$}
        $x := x + 1;$
}
{$s = \sum_{i=0}^{N-1} A[i]$}
```

If execution starts in a state satisfying the precondition, then whenever an assertion is reached, it will be satisfied by the state at that point in time.

We still need to prove the basic facts such as

$$\begin{aligned} &\{\mathit{true}\}\, x := 0; \{x = 0\} \\ \{x = {}& 0\} s := 0; \{\mathit{Loop\ Inv}\} \\ &\ldots \\ &\mathit{Loop\ Inv} \wedge x = N \Rightarrow s = \sum_{i=0}^{N-1} A[i] \end{aligned}$$