Introduction

Sequential programs
- one thread of execution/control

Concurrent programs
- multiple threads of execution/control

Shared Memory concurrency
- threads communicate via variables in shared memory
- access must be synchronized

Message Passing Concurrency
- threads communicate by sending and receiving messages
- all memory local to threads/processes

Distributed programs
- executions on different machines
- communication over a network

Why concurrent programs?
- performance;
  - if work can be subdivided into concurrent tasks, time saved

- to model concurrent phenomena
  - GUI events
  - Access to same information by many

- kind of unavoidable in the "age of multicore"
Communicating between processes

Shared variables

Message passing

More refined pictures

- Many simultaneous views of the same variable (x)
Shared Memory Concurrency

- Important because
  - matches common hardware
    - many CPUs, each have many cores
  - supported by mainstream languages
    - C++, Java, ...
  - even on one-core machine, a natural model for capturing real-life concurrency

- Basic properties
  - Each thread executes its own sequence of operations
  - System decides which thread gets a turn to execute, for how long, and on which core
  - programmer may block threads, but not force them to execute
Simple Programming language

- Normal C-style syntax
- Hoare-style pre/post conditions

Example 1:
\{ x == a \land y == b \}

\begin{align*}
    x &= x + z \\
y &= y + z
\end{align*}

\{ x == a + z \land y == b + z \}

- Preconditions
  - What is assumed to be true before a program
- Postconditions
  - What will be true after the program, if it terminates

- Pre/post conditions are not executed !!

- Parallel execution operator

\texttt{co \( S_1 \parallel S_2 \parallel \ldots \parallel S_n \) oc}

- \( S_1, \ldots, S_n \) executed concurrently
- Terminates when all processes terminated
Parallel version of the above program:

\[
\{ x == a \land y == b \}
\]
\[
\text{co} \quad x = x + z; \quad y = y + z
\]
\[
\text{oc} \quad \{ x == a + z \land y == b + z \}
\]

Pre/post conditions still valid because processes independent:
- do not write to the same variables
- neither writes to a variable the other reads

\( \Rightarrow \) no interference

In general, parallelizing statements invalidates postconditions.

**Example 2**

\[
\{ x == 0 \} \quad x = x + 1 ; \quad x = x - 1 \quad \{ x == 0 \}
\]
\[
\{ x == 0 \} \quad \text{co} \quad x = x + 1 \quad \| \quad x = x - 1 \quad \text{oc} \quad \{ x == 0 \}
\]
\[
\{ x == -1 \lor x == 0 \lor x == 1 \}
\]

- Result nondeterministic
- Depends on timing of execution
- Scheduler
- Compiler
Possible execution orders

Program order

Execute operations in the order they appear in the (Sequential) program.

How to interleave operations?

Sequential consistency

Assume the operations of all individual processes are executed in a sequential order where each process' operations are in program order. If the result of an execution is the same as some such order, the execution is sequentially consistent.
- Conceptual model
  - Every processor issues (shared) memory reads and writes in program order
  - Switch accepts commands from one processor at a time

- Sequential consistency (wishful thinking)
  - Delays in when one process sees a write by another
  - Compilers and hardware reorder operations

- Reality is relaxed memory models

  - Sequential consistency for data-race free programs
    - Programmer ensures no data-races, then gets sequential consistency by the system

- Data race = two processes concurrently access the same memory location; at least one access is a write; (accesses are not synchronized)

- Synchronization = restricting possible interleavings (to avoid "bad")
Atomic operations
- cannot be subdivided
  ⇒ no observable intermediate state
- language dependent
  - read variable usually yes
  - write variable usually yes
  - read + write usually no

\[ X = Y^j \quad // \quad \text{not atomic} \]
\[ \text{++ } X^j \quad // \quad \text{not atomic} \]
- typically there are special atomic operations / variable types
  \text{atomic } \langle \text{int} \rangle \ X^j \quad // \quad \text{atomic}
  \]
  \[ X \text{ ++ } 1 \]
  \[ \text{atomic\_compare\_exchange\_weak}(a, b, c) \]
  \[ // \quad \text{atomic} \]
- language constructs may allow creating large atomic block
  \text{atomic } \{ \]
  \[ X = Y^j \]
  \[ \text{++ } Y^j \]
  \[ \text{++ } X^j \]
  \[ \}
  \[ j \]
Atomic operations
- a statement with at most one 
  atomic operation + operations on 
  local variables can be considered atomic

Mutual Exclusion
- atomic operations on a variable 
  cannot happen simultaneously 
  - one “happens before” the other

Example

\[
P_1 : \quad C_0 \quad x = x + 1 \quad \parallel \quad C_0 \quad \quad x = x - 1 \quad \quad C_0
\]

\[
\text{read } x ; \quad R_1 \\
\text{inc } \quad \text{write } x ; \quad W_1
\]

\[
\text{read } x ; \quad R_2 \\
\text{dec } \quad \text{write } x ; \quad W_2
\]

- Four atomic operations: \( R_1, W_1, R_2, W_2 \)
- Program order:
  - \( R_1 \) happens before \( W_1 \)
  - \( R_2 \) happens before \( W_2 \)
- inc, dec local 
  - can be considered to be part of 
    \( R \) or \( W \)

6 interleavings
- 2 good
- 4 bad

\[
\begin{array}{cccccccc}
R_1 & R_1 & R_2 & R_2 & R_2 & W_2 & W_1 & W_1 \\
W_1 & R_2 & R_2 & R_1 & R_1 & R_1 & W_1 & W_2 \\
R_2 & W_1 & W_2 & W_1 & W_2 & R_1 & W_2 & W_1 \\
W_2 & W_2 & W_1 & W_2 & W_1 & R_1 & W_1 & W_2 \\
\end{array}
\]
Number of interleavings (SC)

- \( m \) processes
- \( \eta_i \) atomic statements in process \( P_i \)
- number of interleavings
  \[
  \frac{m!}{\prod_{i=1}^{m} \eta_i!} \left( \sum_{j=i}^{m} \eta_j \right)
  \]

- e.g.: \( \text{co } P_1 \parallel P_2 \parallel P_3 \text{ co} \)

<table>
<thead>
<tr>
<th>( \eta_1, \eta_2, \eta_3 )</th>
<th>Interleavings</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>((3)! (2)! = 6)</td>
</tr>
<tr>
<td>2</td>
<td>90</td>
</tr>
<tr>
<td>3</td>
<td>1680</td>
</tr>
<tr>
<td>4</td>
<td>34,650</td>
</tr>
<tr>
<td>5</td>
<td>7,567,560</td>
</tr>
</tbody>
</table>

In relaxed memory model \( \left( \frac{m!}{\prod_{i=1}^{m} \eta_i!} \right)! \)
- ignores program order

\( m = \eta_i = 4 \)

- SC: \( 6 \times 10^7 \)
- Relaxed: \( 2 \times 10^{13} \)
Summary of Assumptions

- reads and writes of values that fit into a word are atomic (int, char, T*, ...)
- values manipulated in registers
- registers local to processes
  - if same processor changes from one
    process to another => context switch
    (save/restore registers)
- intermediate results (temporaries) are local

"At-most-one" property for \( x = e \)

Def: critical reference is a reference to a variable written to by another process

- if expression \( e \) has no critical references, \( e \) appears atomic

\( z = e \) satisfies "amo" if either

1. \( e \) contains at most one critical reference and \( z \) is not read by another process
   or

2. \( e \) contains no critical references (we say \( e \) is amo)

- Assignment appears atomic if it is "amo"
Examples of Amo

int x=0, y=0
co x=x+1 || y=y+1 oc
   - no critical references => amo

int x=0, y=0
co x=y+1 || y=y+1 oc
   - y critical in P1, put x not read in P2
   - post: x ∈ \{1,2\}, y = 1

int x=0, y=0
co x=y+1 || y=x+1 oc
   - neither is amo

- if expression or assignment is not amo, it often must be arranged to be executed atomically
  - use synchronization to create a coarse-grained atomic action
  - a sequence of fine-grained atomic actions that appears atomic

- x=x atomic? x=x-x atomic? sets x to 0?
await language

- Book uses a C-like language with a few extra constructs
  
  - Special quantifiers:
    
    - for \([i = 0 \text{ to } n - 1]\) \(a[i] = 0\);
  
  - Concurrent statements
    
    - \textbf{co} s1; s2; s3; \textbf{oc}
    
    - \textbf{co} \([i = 0 \text{ to } n]\) \{ a[i] = 0 \};
  
  - Processes
    
    - process foo \{ ... \}
    
    - process bar\([i = 1 \text{ to } n]\) \{ write(i); \}
  
  - Await statements
    
    - \(<\text{await (B)} \ S;>\)
      
      - atomic: \(<S>\)
      
      - conditional synchronization: \(<\text{await (B)};>\)
Disjoint processes, read/write variables

- $V$: statement $\rightarrow$ variable set
  - set of global variables in a statement (or expression)

- $W$: statement $\rightarrow$ variable set
  - set of global write variables

\[
V(x = e) = V(e) \cup \{x\} \\
V(S_1; S_2) = V(S_1) \cup V(S_2) \\
V(\text{if } b \text{ then } S) = V(b) \cup V(S)
\]

\[
W(x = e) = \{x\} \\
W(S_1; S_2) = W(S_1) \cup W(S_2)
\]

- No common variables for $S_1$, $S_2$

\[
V(S_1) \cap V(S_2) = \emptyset \quad \text{means no interference}
\]

- Weaker condition suffices

\[
V(S_1) \cup W(S_2) = W(S_1) \cap V(S_2) = \emptyset
\]

Read only variables cause no interference
Coarse-grained atomic actions
- when non-interference does not hold
  must restrict interleavings
    (synchronization, atomic blocks)

\[ \text{Co} \langle X = X + 1 \rangle || \langle X = X - 1 \rangle \text{ OC} \]

- intermediate states not visible to other processes
- variable changes from other processes not observed

- NOTE! Book confuses critical sections and atomic blocks a bit

- if S satisfies muη, then \( \langle S \rangle \) and \( S \) have the same effect

Conditional atomic statement - await
\[ \text{await} (B) \ S; \]
- wait until guard B true,
  then evaluate S

- atomic:
  - B evaluated atomically
  - B is true when S begins
  - S evaluated atomically
Properties of concurrent programs

- **state**: snapshot of values of all shared variables
- **history**: = sequence of states, or = sequence of memory operations

- **property**
  = predicate over program history

- **true property**
  = predicate that is true for all possible histories

Some properties of interest

- **Safety**: program cannot reach a bad state
- **liveness**: program will eventually reach a desired state

- **partial correctness**:
  if program terminates, if does so in a desirable final state

- **termination**:
  all histories are finite

- **total correctness**
  : partial correctness + termination
How to ensure desired properties?

**Testing**
- Increases confidence, but not a proof
- Impractical to cover all states

**Operational reasoning**
- Analyze all possible histories of programs

**Formal analysis**
- Produce a proof
- Chain Hoare-triples

```plaintext
{1} #
int x = get_number();
{ x >= 0 } #
{ x >= 0 } #
if (x == 0) throw "too small";
{ x >= 1 } #
{ x >= 1 } #

y = isprime(x);
{ y = 0 || y = 1 } #
{ y = 0 || y = 1 } #

{ n >= 1 } #
int isprime(int n) {
    ... #
    { result == 0 || result == 1 } #
```
Invariants

- **Invariant** (adj): constant, unchanging
- cf. loop invariant, class/object invariant

Def. **Invariant**
- property of program state that holds for all reachable states
  - global invariant
    - about state shared by all processes
  - local invariant
    - about state local to one process (shared + local variables)

To prove invariants, induction useful
- if holds initially, and each atomic statement preserves it, then holds holds everywhere
Producer/consumer example

int buf, p = 0, c = 0; //globals

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;
    }
}

- local invariant
  - producer: 0 ≤ p ≤ n
  - consumer: 0 ≤ c ≤ n
- global invariant
  \[ c ≤ p ≤ c + 1 \]