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

<table>
<thead>
<tr>
<th>to</th>
<th>from</th>
</tr>
</thead>
<tbody>
<tr>
<td>work</td>
<td>read</td>
</tr>
<tr>
<td>shared memory</td>
<td></td>
</tr>
</tbody>
</table>

Message passing

Po \<---\ send \rightarrow\ Pi

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-Syntax
- Hoare-style pre/post conditions

Example 1

\{ x == a \land y == b \} 
\begin{align*} 
  x &= x + z; \\
  y &= y + z; \\
\{ x == a + z \land y == b + z \} 
\end{align*}

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

- Pre/postconditions are not executed !!

- Parallel execution operator

\begin{align*} 
  &\text{co} \ S_1 || S_2 || \ldots || S_n \text{ co} \\
  &- S_1, \ldots, S_n \text{ executed concurrently} \\
  &- \text{terminates when all processes terminated} 
\end{align*}
- parallel version of the above program

\[
\begin{align*}
\{& x = a \land y = b \} \\
& \begin{cases} \\
\text{co} & x = x + z; \ || \\
& y = y + z; \\
\text{co} & \{ x = a + z \land y = b + z \} \\
\end{cases}
\]

- 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**

\[
\begin{align*}
\{ & x = 0 \} \; x = x + 1; \; x = x - 1 \quad \{ x = 0 \} \\
\{ & x = 0 \} \; \text{co} \; x = x + 1 \; || \; x = x - 1 \; \text{co} \\
\{ & x = 0 \} \\
\end{align*}
\]

- 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 with full 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

\[ x = y; \quad // \text{not atomic} \]
\[ ++x; \quad // \text{not atomic} \]
- typically there are special atomic operations on variable types
  \[
  \text{atomic } \langle \text{int} \rangle x; \quad // \text{atomic}
  \]
  \[
  x += 1; \quad // \text{atomic}
  \]
  \[
  \text{atomic\_compare\_exchange\_weak}(a,b,c); \quad // \text{atomic}
  \]
- language constructs may allow creating large atomic block
  \[
  \text{atomic } c
  \]
  \[
  x = y; \]
  \[
  t += y; \]
  \[
  t += x; \]
  \[
  f
  \]
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

\[ P_1 \quad C_0 \quad x = x + 1 \quad || \quad P_2 \quad x = x - 1 \quad C_0 \]

- Four atomic operations: \( R_1, W_1, R_2, W_2 \)

- Program order:
  - \( R_1 \) happens before \( W_1 \)
  - \( R_2 \) happens before \( W_2 \)

- \( \text{inc, dec} \) local
  - can be considered to be part of
    \( R \) or \( W \)

6 interleavings
- 2 good
- 4 bad
Number of interleavings (SC)

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

- e.g.: $c_0 \parallel P_1 \parallel P_2 \parallel P_3 \parallel c_0$

<table>
<thead>
<tr>
<th>$n_1$, $n_2$, $n_3$</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>34650</td>
</tr>
<tr>
<td>5</td>
<td>756756</td>
</tr>
</tbody>
</table>

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

$m = n_c = 4$

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

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

\[ \text{int } x = 0, \ y = 0 \]
\[ \text{co } x = x + 1 \ \Box \ y = y + 1 \ \Box \]
\[ - \text{no critical references } \Rightarrow \text{amo} \]

\[ \text{int } x = 0, \ y = 0 \]
\[ \text{co } x = y + 1 \ \Box \ y = y + 1 \ \Box \]
\[ - y \text{ critical in } P_1, \text{ put } x \text{ not read in } P_2 \]
\[ - \text{post: } x \in \{1, 2\}, \ y = 1 \]

\[ \text{int } x = 0, \ y = 0 \]
\[ \text{co } x = y + 1 \ \Box \ y = x + 1 \ \Box \]
\[ - \text{neither is amo} \]

- 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 to n - 1] a[i] = 0;

  • Concurrent statements
    • co s1; s2; s3; oc
    • co [i = 0 to n] { a[i] = 0 };

  • Processes
    • process foo { ... }
    • process bar[i = 1 to n] { write(i); }

  • Await statements
    • <await (B) S;>
      • atomic: <S>
      • conditional synchronization: <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

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

- No common variables for \( S_1, S_2 \)

\( V(S_1) \cap V(S_2) = \emptyset \) means no interference

- Weaker condition suffices

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

read only variables cause no interference
Coarse-grained atomic actions

- when non-interference does not hold
  - must restrict interweavings
    - (synchronization, atomic blocks)

\[ \text{Co} \left( X = x + 1 \right) \parallel \left( X = x - 1 \right) \text{ Oc} \]

- 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 mua, then \( \langle S \rangle \) and S have the same effect

Conditional atomic statement - await

\[ \langle \text{await} \,(B) \,S; \rangle \]
- 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, it 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
{3
  int x = get_number();
  {x > 03
    {x > 03
      if (x == 0) throw "too small";
      {x > 13

    {x > 13
    y = isprime(x);
    {y = 011 y = 13

{3
  int get_number();
  ...
  3
  result = 011 result = 13

{3
  int isprime(int n) {
    ...
  3
  result > 03
```
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 processors
  - 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**

```c
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
```

Comments:
- poor parallelism
- book's version was data races

Diagram:
- Nodes: a, b, p, c
- Edges: a → b, p < n, c < n
- Arrows: p → b, c → a

Locks and barriers

- Need: non-atomic reads and writes must appear atomic

- One solution:
  
  protect critical sections
  
  = part of program that needs to be protected against interference
  
  execute under mutual exclusion

- One solution to mutual exclusion:
  
  lock
  
Access to critical section (CS)

- Several processes compete for access to a shared resource
- Only one allowed at a time
  = mutual exclusion

- Examples:
  
  - bank transactions
  - access printer
  - access cout
Example

process \( p \) [i = 1 to n] {
    while (true) {
        CSEntry // entry protocol
        CS
        CSExit // exit protocol
        non-CS
    }
}

Assumptions:
- a process that enters CS eventually exits through protocol: beware exceptions!
- all accesses to critical sections
  - if access same resource, protect by same lock

First solution

Bad:
- enforces strict turns
- data races, breaks mutual exclusion

int in = 1;

Process p1 {
    while (true) {
        while (in == 2) {
            CS
            in = 2;
            non-CS
        }
        CS
        in = 1;
        non-CS
    }
}
int in = 1;

Process p1 {
    while (true) {
        while (in == 2) {
            fence
            in = 2;
            non-CS
        }
        CS
    }
}

Process p1 {
    while (true) {
        while (in == 1) {
            fence
            in = 1;
            non-CS
        }
        CS
    }
}

--- some HW/OS/compiler support needed to prevent reordering of memory operations to in variable ---
Desired properties

\{ 
\begin{align*}
\text{mutual exclusion} & \quad \text{at most one process at a time} \\
\text{no deadlock} & \quad \text{if two or more processes try to enter, one will succeed} \\
\text{absence of unnecessary delay} & \quad \text{if no other process is executing a CS, a process can enter CS} \\
\end{align*}
\}

\{ 
\begin{align*}
\text{eventual entry} & \quad \text{a process waiting to enter CS will eventually succeed} \\
\end{align*}
\}

\text{Easy solution: make all CS atomic}

process p [i = 1 to n] { 
    while (true) { 
        \text{entry protocol}
        \langle CS \rangle \\
        \text{exit protocol}
        \langle \rangle \\
        \text{non-CS}
    }
Critical sections using "locks"

- here, lock just an ordinary shared variable
  
  ```
  bool lock = false;
  process i = 1 to N  {
    while (true) {
      <await (!lock); lock = true>
      CS;
      fence;
      lock = false;
      non-CS;
    }
  }
  ```

- satisfies
  - mutex
  - absence of deadlock
  - absence of unnecessary waiting

- What about eventual entry?
  - depends on scheduler

- can <> be taken out?
  - what breaks if we do?

- In practice, one uses special lock objects
  with lock/unlock operations
  - they also put up fences
    ```
    while (true) {
      lock.lock();
      CS
      lock.unlock();
      non-CS
    }
    ```
Test and Set

- Method/pattern for implementing conditional atomic actions

\[ \text{TS (lock)} \{
\text{< bool initial = lock; // save initial value}
\text{lock = true; // set lock}
\text{return initial;}
\}\]

- after TS(lock), lock == true
- return value true indicates that lock was and remains true
- return value false indicates that lock was acquired

- TS usually exists as atomic HW operation
  - important building block for "lock-free" algorithms

A CS protocol with TS (spin lock)

bool lock;
process [i = 1 to n] {  
  while (true) {  
    while (TS(lock)) skip;
    CS
    fence
    lock = false;
  }  
non-CS

- mutex
- needs "strong" fairness for eventual entry
- lock variable a "hot spot"
Test and test and set

In test-and-set processes, read and write the same shared variable in a loop when trying to access a critical section:
- Memory contention → slowness
- Write even if no change → unnecessary cache invalidations → slowness

Optimized protocol:

```java
bool lock = false
process [i = 1 to n] {
    while (true) {
        1. while (lock) skip; // entry protocol
        2. while (TS(lock)) {
            3. fence;
            lock = false;
            non-CS;
            max;
        }
        4. while (lock) skip; // exit protocol
    }
    5. Spin until lock could be unlocked
    6. Try to lock atomically
    7. Spin until lock could be unlocked
    - Better but after lock becomes false, many processes could still execute TS.
```
Implementing await statements

- Assume CEntry, CExit implement entry and exit protocols for CS

- \(<S>\) is CEntry; S; CExit

- \(<\text{await}(B)\; S>\)
  is
  
  CEntry
  while (!B) { CExit; CEntry }
  
  CS
  
  CExit
  
  consider adding a delay
Liveness property: eventual entry
- often depends on scheduling

**Enabledness**
- command enabled if it can in principle execute next
- often more than one enabled statements

```c
bool x = true;
co while (x) skip; || x = false; oc
```

**Scheduling**
- pick one of enabled statements

**Fair scheduling (informally)**
- do not systematically neglect an enabled statement

**Def**: Unconditional fairness
= each unconditional atomic action eventually scheduled
  e.g. x = false above ev. chosen

Example: round-robin

**Def**: Weak fairness
- unconditionally fair and
- every conditional atomic action will eventually be chosen, assuming its condition becomes true and remains true until chosen

```c
bool x = true; int y = 0;
co while (x) y = y + 1; || await y >= 10; x = false; oc
```
- program terminates
Strong fairness

- unconditionally fair, and
- each conditional atomic action will eventually be chosen if the condition becomes true infinitely often

```c
bool x = true;  y = false;
co
  while (x) {
    y = true;  y = false;
  }
ll
  < await (y)  x = false >

oc
- strongly fair scheduler → guaranteed termination
- weakly fair scheduler → no guarantee

- In practice, schedulers are weakly fair
  =) many processes compete for same lock
  =) lock value alternates
  =) condition becomes true infinitely often
  =) no eventual entry guarantee
  to the CS
Tie Breaker algorithm

```cpp
bool in1 = false, in2 = false; int last;

// Assume no reordering!

process p1 {
  while (true) {
    in1 = true;
    last = 1
    <await ((!in2) || last == 2);>
    CS
    in1 = false;
    non-CS;
  }
}

process p2 {
  while (true) {
    in2 = true;
    last = 2;
    <await ((!in1) || last == 1);>
    CS
    in1 = false;
    non-CS;
  }
}

— Even if processes modify variables that affect
wait conditions, once condition becomes
true, it stays true
  => weakly fair scheduler guarantees
      eventual entry

— Extends to n processor
  - Loop with n-1 2-process tie-breaker algorithms
```
**Ticket algorithm**

- Intuition: take a number at NAV
- process that arrives gets a number higher than that of any other process
- process with lowest number is served first

```plaintext
int number = 1, next = 1, turn[1:n] = ([n]0);
process p[i=1 to n] {
    while (true) {
        < turn[i] = number; number = number + 1; >
        < await (turn[i] == next); >
        while (loop ok)
            CS
            < next = next + 1; >
        non-CS
        1. A i≠j, turn[i] > 0 =⇒ turn[i] * turn[j]
        2. if πi in CS, turn[i] == next
        3. before entry, turn[i] < number

- some architectures have push_and_add (FA)
  FA (var, incr) = <int tmp = var;
  var = var + incr;
  return tmp; >

- could replace ① with
  turn[i] = FA (number, 1);
- if no atomic blocks, no FA, then use CS protocols

CS entry;
  turn[i] = number;
  number = number + 1; CS exit;

possibly not strongly fair!
- but in practice yes; denying a process forever very unlikely since CS short
Barriers

Workers + coordinator problems
- Coordinator doing busy-waiting should execute in its own core
- Even so, one core wasted
- And coordinator’s execution time proportional to number of workers
- Cycles through flags iteratively

Combining tree barrier

![Tree Diagram]

Leaf:
\[
\text{arrive}[L] = 1;
\text{await}(\text{cont}[L] == 1);
\text{cont}[L] = 0;
\]

Interior:
\[
\text{await}(\text{arrive}[\text{left}] == 1);
\text{arrive}[\text{left}] = 0;
\text{await}(\text{arrive}[\text{right}] == 1);
\text{arrive}[\text{right}] = 0;
\text{arrive}[I] = 1;
\text{await}(\text{cont}[I] == 1);
\text{cont}[I] = 0;
\text{cont}[\text{left}] = 1; \text{cont}[\text{right}] = 1;
\]

Root:
\[
\text{await}(\text{arrive}[\text{left}] == 1);
\text{arrive}[\text{left}] = 0;
\text{await}(\text{arrive}[\text{right}] == 1);
\text{arrive}[\text{right}] = 0;
\text{continue}[\text{left}] = 1; \text{continue}[\text{right}] = 1;
\]
Symmetric Barriers
- barrier for two processes $i$ and $j$

For $i$:

```plaintext
<await (arrive [i] == 0);>
arrive [i] = 1;
<await (arrive [j] == 1);>
arrive [j] = 0;
```

For $j$:

```plaintext
<await (arrive [j] == 0);>
arrive [j] = 1;
<await (arrive [i] == 1);>
arrive [i] = 0;
```

- To build a $n$-process barrier, need $\lceil \log_2 n \rceil$ stages

**Example:** Butterfly barrier

<table>
<thead>
<tr>
<th>Workers</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stage 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stage 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Example:** Dissemination barrier

<table>
<thead>
<tr>
<th>Workers</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stage 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stage 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Data parallel algorithms

Prime use of barriers
- Several processes execute the same code for different pieces of data
- Barriers synchronize phases of execution
- Example: Map-reduce
- Example: Parallel prefix computation
  - Folding an associative binary operation over a collection of elements
  - E.g., all partial sums of an array
    \[
    \begin{align*}
    \text{sum}[0] &= a[0]; \\
    \text{for } [i=1 \text{ to } n-1] & \{ \text{sum}[i] &= \text{sum}[i-1] + a[i] \}\end{align*}
    \]

- Initial values of a
  - sum after dist 1
    - 1 2 3 4 5 6
  - dist 2
    - 1 3 5 7 9 11
  - dist 4
    - 1 3 6 10 15 21
int a[n], sum[n], old[n];

process Sum[i = 0 to n-1] {
    int d=1;
    sum[i] = a[i]; // initialize sum
    barrier(i);
    while (d < n) {
        old[i] = sum[i]; // save old value
        barrier(i);
        if ((i-d >= 0) sum[i] = old[i-d] + sum[i-d];
        barrier(i);
        d = d + d;
    }
}
Semaphores
- synchronization tool
- introduced by Dijkstra 1968
- inspired by traffic synchronization

railroad semaphores

In programming, semaphore is a special program variable
- value is a non-negative integer
- can only be manipulated by two atomic operations

P: wait for signal (passeren)
effect: wait until value > 0
then decrease value by 1

V: signal an event (Vrijgeren)
effect: increase value by 1

Today’s names different: up/down, wait/notify
- Binary vs. counting semaphores
- Mutex & binary semaphore
Syntax

sem s; // initialize to zero
sem s = n; // init to n
sem s[n] = ([n] 1); // array of sems

semantics

P(s):
V(s)

< await (s > 0); s = s - 1 > < s = s + 1 >

- There is no other access to s!
  - if (s = -1) ... NOT ALLOWED

Fairness for processes waiting at P?
- as for await statements, weak fairness
- often FIFO queue
  - processes delayed by P operations are awoken in order
**Mutual exclusion with semaphores**

```
sem mutex = 1;
process CS { i = 1 to n; }
    while (true) {
        P(mutex);
        CS
        V(mutex);
        non-CS
    }

- semaphore is initially 1, so that
  one process can enter CS
- always P before V, value stays ≤ 1
```

**Barrier with semaphores**

```
sem arrive1 = 0, arrive2 = 0;
process P1 {
    V(arrive1); // notify
    P(arrive2); // I'll wait for P2
}
process P2 {
    V(arrive2); // notify
    P(arrive1); // I'll wait for P1
}
```

- use case of signalling events
  - semaphores initialized to 0
  - signal event = V
  - wait for event = P
Split binary semaphores

- A set of semaphores, whose sum of values is 1
- Achieves mutual exclusion of many processes
- Initialize one semaphore to 1, others to 0
- In every execution path:
  call to P on some semaphore
  is followed at some point with a call to V on some semaphore

\[ \Rightarrow \text{all statements between } P \text{ and } V \text{ execute in mutual exclusion;}
\text{and all semaphores are 0} \]

Example

```c
typedef int buffer; sem empty = 1, full = 0;
process producer[ci=1 to m]{
   while (true) {  
      ...  
      P(empty);  
      buffer = data;  
      V(full);  
   }  
}
```

```c
process consumer[ci=1 to n]{
   while (true) {  
      ...  
      P(full);  
      data = buffer;  
      V(empty);  
   }  
}
```

- empty, full are both binary semaphores
- together they are a split binary semaphore
Bounded buffer,
- example use of general semaphores (values > 1)
- Prod/cons with one element buffer is not good
- Generalize to a n-element buffer
  - asynchronous communication
- Ring buffer
  - array of rear and front indices
  - semaphores to keep track of the number
    of used/free slots
  - general semaphore

```
T buf[n];
int front = 0, rear = 0;
sem empty = n; // # empty slots
sem full = 0;  // # full slots

process Producer {
    while (true) {
        P(empty);
        buf[rear] = data;
        rear = rear + 1
        V(full);
    }
}

process Consumer {
    while (true) {
        P(full);
        result = buf[front];
        front = (front + 1) % n;
        V(empty);
    }
}
```
Producer-consumer with multiple producers and consumers

- new synchronization problems
  - deposit but [rear] and increment rear must be atomic
  - fetch but [front] and increment front must be atomic

=> add 2 new semaphores
  - mutex Deposit, mutex Fetch

```c
process Producer [i = 1 to k] {
  int front = 0, rear = 0;
  sem empty = N;
  sem full = 0;

  while (true) {
    P(Empty);
    P(mutex Deposit);
    buf[rear] = data;
    rear = rear + 1 % N;
    V(mutex Deposit);
    V(full);
  }
}

process Consumer [i = 1 to l] {
  sem mutex Deposit = 1;
  sem mutex Fetch = 1;

  while (true) {
    P(mutex Fetch);
    result = buf[front];
    front = (front + 1) % N;
    V(mutex Fetch);
    V(Empty);
  }
}
```
Dining Philosophers (Dijkstra)

- Circular table, 5 philosophers
- One fork between each seat
- Philosophers alternate between thinking and eating
- Need 2 forks for eating, 0 for thinking
- Goal: Philosophers should not starve.

```c
process Philosopher [i = 0 to 4] {
    while true {
        think;
        acquire 2 forks; // 2
        eat;
        release 2 forks; // 2
    }
}

Attempt 1: pick left fork first

sem for[i] = (CS 1);
process Philosopher [i = 0 to 4] {
    while true {
        think;
        P(fork[i]); P(fork[(i+1) % 5]);
        eat;
        v(fork[i]); v(fork[(i+1) % 5]);
    }
}

Deadlock!
```

- Deadlock can happen with circular waiting
- Conc. programming libraries have tools for acquiring many locks, either all of them or none of them, so that deadlock can be avoided

- eg. `try_lock` in C++

```
while (!try_lock(leftfork, rightfork)) {}
    eat();
    leftfork.unlock(); rightfork.unlock();
```
Attempt 2: Break symmetry to break the circular waiting

```c
sem fork[5] = (CS 1);
process Philosopher[i=0 to 3] {
    while (true) {
        think;
P(fork[i]); P(fork[i+1]);  // left first
        eat;
        V(fork[i]); V(fork[i+1]);
    }
}

3
process Philosopher 4 {
    while (true) {
        think;
P(fork[0]); P(fork[4]);  // right first
        eat:
        V(fork[0]); V(fork[4]);
    }
}
```

- Probabilistic solutions exist where each process executes the same algorithm
The Readers/Writers Problem

- classical synchronization problem
- resource: database, file, or such
  - readers (R) only read
  - writers (W) (also) modify

- W access is exclusive
- many R accesses allowed

- Synchronization view:
  - readers must wait until no active writers
  - writers must wait until no active readers or writers

- Implementation approaches
  - mutual exclusion
  - condition synchronization
    - with atomic blocks & await, or
    - with split binary semaphores
      - adapts to different scheduling strategies
Readers / Writers with mutual exclusion

\[ \text{sem } rw = 1; \]

```c
process R[i=1 to m] { 
    while (true) {
        P(rw); // read
        V(rw);
    }
}
```

- too cautious; only allows 1 simultaneous reader

```c
int nr = 0; // # active readers
sem r = 1; // reader mutex
sem rw = 1; // reader/writer mutex
```

```c
process W[i=1 to m] { 
    while (true) {
        P(rw); // write
        V(rw);
    }
}
```

Attempt 2

```c
process R[i=1 to m] { 
    while (true) {
        P(r);
        nr = nr + 1;
        if (nr == 1) P(rw);
        V(r);
        // read
        P(r);
        nr = nr - 1;
        if (nr == 0) V(rw);
        V(r);
    }
}
```

- prefers readers
- Two separate synchronization problems
  1. readers' & writers' access to resource
  2. reader's access to the nr counter
Readers/Writers with condition synchronization

**Invariant:**

\[ (nr = 0 \lor nw = 0) \land nw \leq 1 \]

- \(nr\) = # active readers
- \(nw\) = # active writers

- Straightforward using awaits

```plaintext
process R [i=1 to m] {
  while (true) {
    // read
    <nr = nr-1>
  }
}
```

```plaintext
process W [i=1 to n] {
  while (true) {
    // write
    <nw = nw-1>
  }
}
```

Condition Synchronization with split binary semaphores

- `await` can be implemented with split bin. semaphores, any condition synchronization problem implementable with s.b.s.

- General pattern:
  - entry semaphore (= 1)
  - for each guard \(B_i\), use
    - 1 semaphore to delay processes waiting for \(B_i\)
    - 1 counter for # processes waiting for \(B_i\)
    - initialize both to zero

  \(\Rightarrow\) entry semaphore + delay semaphores form a split binary semaphore
Readers/ Writers with split binary semaphores

Following the general pattern gives
3 semaphores and 2 counters

\[ \text{sem } c = 1; \]
\[ \text{sem } r = 0; \]
\[ \text{sem } w = 0; \]

\[ \text{int } dr = 0; \]
\[ \text{int } dw = 0; \]

\[ \text{counters} \]

\[ \text{entry semaphore} \]

\[ \text{split binary semaphore} \]

\[ \text{delay semaphores} \]

- all execution paths start with P and end with V
  \[ \Rightarrow \text{guarantees mutual exclusion} \]

- signalling mechanism for selecting
  which semaphore’s V to call

\[ \text{int } nr = 0, nw = 0; \]

process \( R[i=1 \text{ to } m] \) if

\[ \text{loop} \{ \]
\[ \downarrow P(e); \]
\[ \text{if} (nw > 0) \{ \]
\[ \quad dr = dr + 1; \]
\[ \quad \uparrow V(e); \]
\[ \downarrow P(r); \]
\[ \} \]
\[ nr = nr + 1; \]
\[ \uparrow \text{SIGNAL} \]
\[ \text{// read} \]
\[ \downarrow P(e); \]
\[ nr = nr - 1 \]
\[ \uparrow \text{SIGNAL} \]

ends with \( V \)

process \( W[i=1 \text{ to } n] \) if

\[ \text{loop} \{ \]
\[ \downarrow P(e); \]
\[ \text{if} (nr > 0 \text{ or } nw > 0) \{ \]
\[ \quad dw = dw + 1; \]
\[ \quad \uparrow V(e); \]
\[ \downarrow P(w); \]
\[ \} \]
\[ nw = nw + 1; \]
\[ \uparrow \text{SIGNAL} \]
\[ \text{// write} \]
\[ \downarrow P(e); \]
\[ nw = nw - 1; \]
\[ \uparrow \text{SIGNAL} \]
SIGNAL:
    if (nw == 0 & dr > 0) {  // no writers, delayed
        dr = dr - 1;
        V(r);
    } else if (nr == 0 & nw == 0 & dw > 0) {  // no writers or readers,
        dw = dw - 1;
        V(w);
    } else {  // no delayed processes
        V(e);
    }

we emulate \langle\texttt{await Bi} \mid S\rangle,
where \texttt{Bi} holds at start of \textbf{S}

In emulation, \texttt{Bi} holds because
- SIGNAL only wakes up \textbf{S} when \texttt{Bi} holds
- reader/writer process checks \texttt{Bi}
  (protected by semaphore)

Note: deadlock avoided by checking
counters before signalling delay semaphores
- \texttt{V(r)} only called if \exists delayed readers
- \texttt{V(w)} only called if \exists delayed writers

SIGNAL can be simpler for some uses
(R1):
    if (dr > 0) { dr = dr - 1; V(r); } else V(e);
(W1):
    V(e);
(R2):
    if (nw == 0 & dw > 0) { dw = dw - 1; V(w); } else V(e);
(W2):
    if (dr > 0) { dr = dr - 1; V(r); } else if (dw > 0) { dw = dw - 1; V(w); } else V(e);
Passing the Baton

- Our use of split binary semaphores is an instance of "passing the baton" strategy
  - when process exits a CS, it passes the baton to another process
  - if nobody to take the baton, deposit it at the entry semaphore

- by changing the signalling mechanism, to which process to give the baton, we get different scheduling strategies
  - e.g.: favor writers, alternate, flip a coin

- In readers/writers we have two classes of processes
  - in general can have $N$ classes
  - each process can be in its own class!

  a) semaphore and counter for each class
  - arbitrary scheduling policies can be implemented
    - priority queues (smallest, quickest first)
    - aging
    - random
    - FIFO
General Resource allocation problem
- Processes compete for units of shared resource
- Requests parameterized:
  - how many units
  - what kind
  - requester's identity
- Each unit is either free or in use
- Request satisfied if all requested units free
- Release operation also parameterized
  - which units
  - NOTE: units can be released in different order and in different amounts

Code Outline

```c
request (params):
  P(e);
  if (request cannot be satisfied) {
    DELAY
    take units;
    SIGNAL;
  }

release (params):
  P(e);
  return units;
  SIGNAL;
```
Monitors

- Semaphores good
  - Simple;
    
    \[
    \text{sem } s := 1; \text{ } P(s); \text{ } V(s);
    \]
  - Easy to program mutex, signalling
  - Any synchronization problem solvable, if one is systematic

- Not so good
  - Low-level
  - Global to all processes,
    - Must examine the entire program for any safety guarantees about shared variables
  - Must be careful to
    - Use the right semaphore
    - Not omit and not execute extra P, V
    - Protect all critical sections

- Semaphores used both for mutex and condition synchronization
  - Purpose of P, V operations may not be immediately clear

\[\Rightarrow\text{ monitors}\]
**Monitor** $\equiv$ abstract data type + synchronization

- encapsulates a representation of an abstract object
- provides methods that are the only way to manipulate the representation
- implicit mutual exclusion:
  - only one method can execute at a time
- condition synchronization via condition variables

**Programming with monitors, idea**

- processes (active)
- monitors (passive)
  - all shared variables in monitors

Two processes interact only by calling methods of the same monitor

**Benefits**

- process calling a monitor's method does not need to know its implementation
- monitor does not need to know about its users and can change its implementation

$\implies$ modularity

Java, also C++, support monitor abstraction
Syntax & Semantics

monitor m {  
permanent variables // shared by all processes  
initialization  
procedures // public

}  

- outside:  
  - only procedures visible  
    call m.proc(args);  

- inside:  
  - no access to variables outside the monitor  
  - initialization occurs before any procedure executed

- monitor invariant  
  - predicate that is true when no procedure is running  
  - describes the valid/good states  
    cf. class invariant in OOP

Sketch:+++

class m {  
    member variables  
    lock l; // intrinsic lock  
public:  
    m() { ... }  
void method1(args) {  
    l.lock(); // every method surrounded  
    // ...  
    l.unlock(); // by lock/unlock
    }
}
**Condition Variables**

- special variables in monitor
  
  ```cond cv;```
- used for delaying a process when it has to wait for a condition and for waking up delayed processes
- value is
  
  - a queue of delayed processes
  - not directly accessible to the programmer

```cond cv;    // declaration
empty (cv);  // is cv’s queue empty
wait (cv);   // make process wait on cv's queue
signal (cv); // wake up a process in queue
signal_all (cv); // wake up all processes in queue```

**Semaphore implementation with monitors**

```monitor Semaphore {
    int s = 0;
    cond pos;
    procedure P() {
        while (s == 0) { wait (pos); }
        s = s - 1;
    }
    procedure V() {
        s = s + 1;
        signal (pos);
    }
}```
Signalling disciplines

- signal (cv):
  - if cv's queue empty, no effect
  - otherwise, wake up first process
  - wait/signal perhaps FIFO, could be other strategy too

- when a process executes signal, it is inside monitor.
  => two active processes
    - current
    - awaken

- Two strategies
  1. Signal and Continue, (SC)
     - signaler continues running
  2. Signal and Wait (SW)
     - Signalled starts executing

- waiting code in SC
  
- waiting code in SW

  ```
  while (cond) {
    wait (cv);
  }
  ```

  ```
  if (cond) {
    wait (cv);
  }
  ```

  [This is what is common in practice]
  
  - even spurious wakeups possible
Contrasting semaphores and monitors
- Both P and Wait delay a process,
  - Wait always delays a process, until signal,
    P only if semaphore is 0,
  - Signal has no effect if no one delayed,
    V increments semaphore value (and then possibly awakens a process)

C++ API

- cv. wait (lock)
  - release lock, wait on cv
- cv. wait (lock, predicate)
  = While (!pred()) { wait. lock(); } // ignores spurious awakenings
- cv. notify_one()
  - unblock one of threads waiting for cv
- does not need to hold the same lock
  as the waiting threads
  - usually would be a pessimization
    wake up a thread just to be blocked
- cv. notify_all()
- cv. wait_for (lock, duration)
- cv. wait_until (lock, time)
- cv. wait_until (lock, time, predicate)
Example: Bounded Buffer Synchronization (producer, consumer)

- buffer of size \( n \)
- producer puts, waits if full
- consumer gets, waits if empty
- assume FCFS discipline

- when process woken up, goes to entry queue
- competes with other processes there
- for entry to monitor
- arbitrary delay between awakening and start of execution

=> must retest condition

```plaintext
monitor bounded_buffer {
    type T but [n];
    int front=0, rear=0, count=0; //inv: rear=(front+count)%n
    cond not-full, not-empty;

    procedure put (type T data) {
        while (count == n) wait (not-full);
        but [rear]=data; rear=(rear+1)%n; ++count;
        signal (not-empty);
    }

    procedure get (type T &data) {
        while (count == 0) wait (not-empty);
        data = but [front]; front=(front+1)%n; --count;
        signal (not-full);
    }
}
```

// clients:
process prod [i=1 to n] {
    loop {
        call bb.put(a);
    }
}

process cons [i=1 to m] {
    loop {
        call bb.get(b);
    }
}
Readers/Readers with monitors
- common resource (db)
- R & W processes
  - resource initially consistent
  - each transaction, in isolation, maintains consistency
- Access restriction:
  \[ \#W \leq 1 \ \& \ \& \ \#R \leq 0 \]

Monitor implementation of R/W
- Encapsulate db in a monitor?
  - No, would prevent concurrent reader access
  =) monitor arbitrates access to database, database globally shared
- Processes do not enter a critical section until they have paused

```
RW_controller_monitor

monitor RW_controller { 
  int nr = 0, nw = 0; \ // inv: (nr = 0 V nw = 0) \ \& \ nw \leq 1
  cond rok, wok

  procedure request_read() { 
    while (nw > 0) wait (rok);
    nr = nr + 1; \ // ensure inv.
  }

  procedure release_read() { 
    nr = nr - 1;
    if (nr = 0) signal (wok);
  }

  procedure request_write() { 
    while (nr > 0 || nw > 0) wait (rok);
    hw = nw + 1;
  }

  procedure release_write() { 
    hw = hw - 1;
    signal (wok);
    signal_all (rok);
  }
```

About invariant
- must hold:
  - after initialization
  - when procedure terminates
  - when suspend, execution with wait
  - may assume to hold
  - after wait
  - beginning of procedure
Sleeping Barber (client/server example)

- Shop: two doors, chairs for waiting, barber's chair
- Customers enter through one door, leave through other
- Only 1 customer on barber's chair
- No customers = barber sleeps
- Customer arrives:
  - If barber sleeps, wake up barber, customer served
  - If barber busy, customer naps
- Once served, customer exits
  - If waiting customers, wake up one, serve
  - Otherwise barber naps

Monitor procedures:

Client: get_haircut; // returns customer has exited

Barber: get_next_customer;
  // cut hair
  finish_haircut; // let customer out

Rendez-vous barrier

- Two process barrier, but not with a fixed process
- Here,
  1. Barber waits for any customer
  2. Customer waits for barber to be available
Organizing Synchronization

- Barber must wait until
  - Customer in chair
  - Customer has left
- Customer must wait until
  - Barber available
  - Barber opens exit door

$\Rightarrow$ both barber and customers proceed in synchronized stages starting with rendezvous

Idea 1
- Counters for how many processes have reached each stage
  - Customers:
    - $c \text{ in chair}$; // in barber’s chair
    - $c \text{ leave}$; // leaving
  - Barber:
    - $b \text{ avail}$; // ready for customer
    - $b \text{ busy}$; // cutting hair
    - $b \text{ done}$; // ready to let customer exit

- Invariants
  $C_1 : c \text{ in chair} \geq c \text{ leave} \wedge b \text{ avail} \geq b \text{ busy} \geq b \text{ done}$
  $C_2 : c \text{ in chair} \leq b \text{ avail} \wedge b \text{ busy} \leq c \text{ in chair}$
  $C_3 : c \text{ leave} \leq b \text{ done}$

- (Small) problem: Counters increase without bound

Idea 2
- Counters for difference of stage-counters
  \begin{align*}
  \text{barber} &= b \text{ avail} - c \text{ in chair} \\
  \text{chair} &= c \text{ in chair} - b \text{ busy} \\
  \text{open} &= b \text{ done} - c \text{ leave}
  \end{align*}
- All initially $\emptyset$, values in $\{0,1\}$
- For implementation, see barber_shop.cpp
Program correctness

- How to convince oneself that program behaves as expected
- More difficult for concurrent programs

\[
x = 5; \{ x = 5 \} \quad x = x + 1; \{ x = 6 \} \quad // \text{holds}
\]

\[
x = 5; \{ x = 5 \} \quad \langle x = x + 1 \rangle; \{ x = 6 \} \quad // \text{may not hold}
\]

About approaches for verifying correctness

- Testing
  - “Testing shows the presence of bugs, not their absence!” (Dijkstra, 1969)
  - worthwhile regardless
- Operational reasoning
  - analyse all possible executions
- Formal analysis
  - specify program behaviour
  - construct a formal argument of correctness (without running the program)

Program state

- values of all program variables
- state space
  - all possible values of variables
- In sequential program, one thread modifies state

\[
\text{states: } \{ p_0 \} \quad \{ s_1 \} \quad \{ p_1 \} \quad \{ s_2 \} \quad \{ p_2 \} \ldots \quad \{ p_{n-1} \} \quad \{ s_n \} \quad \{ p_n \}
\]
- Given

\{ \text{Po} \} S_1 \{ \text{P}_1 \} S_2 \{ \text{P}_2 \} \ldots \{ \text{P}_{n-1} \} S_n \{ \text{P}_n \}

- We get a specification for the program

\{ \text{Po} \} S_1 S_2 \ldots S_n \{ \text{P}_n \}

\uparrow \quad \text{initial state} \quad \uparrow \text{final state}

**Assertions**

- Instead of specific states, usually express assertions about states

\begin{align*}
\exists x = y & \quad \exists x = 2 \times x; y = 2 \times y; \{ x = y \} \\
\exists x < y & \quad \exists x = 2 \times x; y = 2 \times y; \{ x < y \}
\end{align*}

\begin{align*}
\exists x > 0 \land y < 0 & \quad \exists x = 2 \times x; y = 2 \times y; \{ x > 0 \land y < 0 \}
\end{align*}

- Assertion characterizes a set of states

\begin{align*}
x = y & \quad \text{all states where } x, y \text{ equal} \\
x = 2 \land y = 3 & \quad \text{exactly one state} \\
\text{true} & \quad \text{all states} \\
\text{false} & \quad \text{no state}
\end{align*}
Formal System

- Syntax:
  1. a set of symbols
     - variables $x, y, z, ...$ (program + additional)
     - relation symbols $\leq, \geq, \ldots$
     - function symbols $+, -, \ldots, 0, 1, 2, \ldots$, true, false
     - equality $=$
  2. a set of formulas
     - well-formed sequences of symbols
     - if $A, B$ formulas then following are also formulas
       $\neg A$
       $A \lor B$
       $A \land B$
       $A \Rightarrow B$
     - if $x$ variable, $A$ formula (mentioning $x$), then
       $\forall x : A(x)$, $\exists x : A(x)$ are formulas

- Derivation machinery
  3. a set of distinguished formulas called axioms
     - assumed to be true
  4. a set of inference rules

\[ \begin{align*}
    H_1 & \quad \cdots \quad H_n \quad \text{premises} \\
    \hline
    C \quad \text{conclusion}
\end{align*} \]

- conclusion true, if assumptions true
- specify how to derive additional true formulas from axioms and prior true formulas

Proof?
- sequence of formulas, where each
  - is axiom, or
  - can be derived from preceding formulas using inf. rules

Theorem?
- any formula in a proof
Example axioms
\[ A \lor \neg A \]
\[ A \Rightarrow A \]

Example inferences rules
\[
\begin{align*}
A & \quad B \\
\wedge \text{-I} & \\
A \land B
\end{align*}
\]
\[
\begin{align*}
A & \quad \neg A \\
\lor \text{-I} & \\
A \lor \neg A
\end{align*}
\]
\[
\begin{align*}
A & \Rightarrow B \\
A & \Rightarrow \neg \text{-E} \\
& \quad B
\end{align*}
\]

Example instances:
\[
\begin{align*}
\ldots & \quad \ldots \\
\ldots & \quad \ldots \\
\ldots & \quad \ldots \\
\ldots & \quad \ldots
\end{align*}
\]
\[
\begin{align*}
x > 0 & \Rightarrow y > 0 \\
x > 0 & \Rightarrow \neg \text{-E} \\
y > 0
\end{align*}
\]

Some terminology
- interpretation of logic: describe each formula as true or false
- sound logic:
  - if formula can be proven, it is true
    (in the interpretation)
- complete logic:
  - if formula is true, it can be proven

Program Logic
- allows for expressing and proving properties of programs
- formulas:
  \[ \{ P_1 \} S \{ P_2 \} \]
- \( S \) is program statement(s)
- \( P_1, P_2 \) assertions over program states
  \( P_1 \) precondition, \( P_2 \) postcondition of \( S \)
Reasoning about programs

1. Specify programs behavior with pre/post conditions
2. Use PL to construct a proof that the program satisfies the specification

Interpretation of Hoare-triples

- Partial correctness interpretation
  \[ \{ P \} S \{ Q \} \text{ is true if} \]
  - assuming initial state of S satisfies P and S terminates, then Q is true at the final state of S

Examples:

\[ \{ x = 0 \} x = x + 1; \{ x = 1 \} \]
\[ \{ x = 4 \} x = x \downarrow 5; \{ x \leq 5 \} \]
\[ \{ \text{true} \} x = 5; \{ x \leq 5 \} \]
\[ \{ y = 4 \} x = 5; \{ y = y \downarrow 4 \} \]
\[ \{ x = 0 \} x = x + 1; \{ x = 0 \}; // \text{not true} \]
\[ \{ x > y \} x = x + 1; y = y + 1; \{ x < y \}; // \text{not true} \]

\[ \{ P \} S \{ \text{false} \} // S \text{ does not terminate} \]
\[ \{ P \} S \{ \text{true} \} // Q \text{ holds regardless of initial state} \]
\[ \{ \text{true} \} S \{ Q \} \]
\[ \{ \text{false} \} S \{ Q \} // \text{trivially true} \]

Soundness

- If \( \{ P \} S \{ Q \} \) is a theorem in PL, its interpretation is true

\[ \vdash \{ P \} S \{ Q \} \]

⇒ If we can prove a property of a program with PL, it holds for all executions of the program
Assignment axiom

\[ \{ P[e/x] \} \ x = e \quad \{ P \} \]

- \([e/x] P\) means: substitute \(e\) for \(x\) in all \((\text{free})\) occurrences of \(x\)

- backward construction:
  - given postcondition of assignment, we can calculate precondition

  - example

    \[ \{ 3 \} \ x = e + 2 ; \quad \{ x = 5 \} \]
    \[ (x = 5) \ [e + 2 / x] \] \[ x = e + 2 ; \quad \{ x = 5 \} \]
    \[ e + 2 = 5 \implies x = e + 2 ; \quad \{ x = 5 \} \]

Free and bound occurrences

- \(\forall, \exists\) quantifiers bind variables

  \((\exists y : x + y > 0)\ [1/x] \iff \exists y : 1 + y > 0\)

  Free \quad \text{bound}

  \((\exists x : x + y > 0)\ [1/x] \iff \exists x : x + y > 0\)

  \((\exists x : x + y > 0)\ [x/y] \iff \exists z : z + x > 0\)

  - must rename bound var.
  - for not to make \(x\) bound
Proving a Hoare triple on assignment

\[ P \Rightarrow Q[e/x] \quad \{ Q[e/x] \} \ x = e; \ \{ Q \} \]

\[ \{ P \} \ x = e; \ \{ Q \} \]

- \( Q[e/x] \) is the largest set of states for which \( Q \) holds after the assignment
- weakest precondition
- set of states satisfying \( P \) must be shown to be in that set

**Example:**

\[
\begin{align*}
&x = 0 \Rightarrow x + 1 = 1 \quad \{ x + 1 = 1 \} x = x + 1 \{ x = 1 \} \\
&\{ x = 0 \} x = x + 1 \{ x = 1 \}
\end{align*}
\]

\[
\begin{align*}
x &= a \land y = b \Rightarrow \\
x + y &= a + b \land y = b \\
&\{ x + y = a + b \land y = b \} x = x + y; \ { x = a + b \land y = b } \\
&\{ x = a \land y = b \} x = x + y; \ { x = a + b \land y = b }
\end{align*}
\]

Rule of consequence

\[ P' \Rightarrow p \quad \{ p \} \ s \ \{ Q \} \quad Q \Rightarrow Q' \]

\[ \{ p' \} \ s \ \{ Q' \} \]

**Example:**

\[
\begin{align*}
p &= \emptyset \\
\Rightarrow 1 \notin p \\
&\{ 1 \notin p \} p.\text{insert}(1); \ { 1 \in p } \quad 1 \in p \Rightarrow \\
p.\text{size}() > 0
\end{align*}
\]

\[
\{ p = \emptyset \} p.\text{insert}(1); \ { p.\text{size}() > 0 } \]
Skip axiom

\{P\} skip; \{P\}

Inference rules

\[ \begin{align*}
\{P\} & S_1, \{R\} & \{R\} S_2 \{Q\} & \text{SEQ} \\
\{P\} & S_1 ; S_2 \{Q\} & \\
\end{align*} \]

\[ \begin{align*}
\{P \wedge B\} & S \{Q\} & P \wedge \neg B \Rightarrow Q & \text{COND} \\
\{P\} & \text{if } (B) S ; \{Q\} & \\
\end{align*} \]

\[ \begin{align*}
\{I \wedge B\} & S \{I'\} & \text{WHILE} \\
\{I\} & \text{while } (B) S ; \{I \wedge \neg B\} & \\
\end{align*} \]

\[ \begin{align*}
\{P\} & S \{Q\} & \{P'\} & S \{Q'\} & \text{CONJUNCTION} \\
\{P \wedge P'\} & S \{Q \wedge Q'\} & \\
\end{align*} \]

Ghost variables:
- variables that only exist in pre/post conditions

\[ \begin{align*}
\{x = x_0\} & \text{if } (x < 0) x = -x ; \{x > 0 \wedge (x = x_0 \lor x = -x_0)\} \\
\end{align*} \]
Examples

\[(x=2y) \quad [2y/x] \]

\[
\begin{align*}
2x &= 2y \\
\frac{2x=2y}{x=2x} &\text{ by the equality rules} \\
\frac{x=y}{\{x=y\}} &\text{ by assumption}
\end{align*}
\]

\[
\{x=y\} \quad x=2x \\
\{x=2y\} \quad y=2y; \{x=y\}
\]

\[
\frac{x=y}{\{x=y\}} \\
\{x=y\} \quad x=2x \\
\{x=y\} \quad y=2y
\]

If verify:

\[
\{x=x_0 \} \quad \text{if } (x<0) \quad \text{then } x=-x; \{x \geq 0 \land (x=x_0 \lor x=-x_0)\}
\]

\[
\begin{align*}
\{P \land \neg B \} &\quad S \{Q\} \quad P \land \neg B \implies Q \\
\{P \land \neg B \} &\quad \text{if } (B) \quad \text{then } S \{Q\}
\end{align*}
\]

- if \(P \land \neg B \): \(x=x_0 \land x<0\)

- assignment axiom gives proof obligation.

- \(x=x_0 \land x<0 \implies -x \geq 0 \land (-x=x_0 \lor x=-x_0)\)

- \(P \land \neg B \implies Q\)

- \(x=x_0 \land \neg (x<0) \implies x \geq 0 \land (x=x_0 \lor x=-x_0)\)
Verifying loops
- number of iterations
  - unknown
  - or keeping track of them too complex

⇒ Define a (loop) invariant \( I \)
- an assertion preserved by the loop

- Finding the right invariant not necessarily easy
  - To prove \( \{ P \} \text{ while } (B) S; \{ Q \} \):
    1. Find invariant \( I \)
    2. Show \( P \Rightarrow I \)
    3. Show \( \{ I \wedge B \} S \{ I \} \)
    4. Show \( I \)

Verification:
\( \{ 0 \leq n \wedge k = 0 \} \text{ while } (k < n) k = k + 1; \{ k = n \} \)

1. Guess \( I = k \leq n \)
2. Show \( 0 \leq n \wedge k = 0 \Rightarrow k \leq n \)
3. Show
   \[
   \begin{array}{c}
   k \leq n \wedge k < n \Rightarrow k + 1 \leq n \\
   \{ k + 1 \leq n \} k = k + 1; \{ k \leq n \}
   \end{array}
   \]
   \[ \text{while} \]
   \[ \{ k \leq n \} \text{ while } (k < n) k = k + 1; \{ k = n \wedge \neg (k < n) \} \]
4. Show \( k \leq n \wedge \neg (k < n) \Rightarrow k = n \)
**Await rule**

\[ \{p \land B\} \text{ } S \text{ } \{q\} \]

\[ \{p\} \langle \text{await } (B) \rangle \text{ } S \text{ } \{q\} \]

- note: termination not required
- if B true, S could be executed, or not

**Concurrent execution**

Assume:

\[ \{p\} \langle S_1 \rangle \{q_1\} \quad \{p_2\} \langle S_2 \rangle \{q_2\} \]

\[ \{p_1 \land p_2\} \text{ co } \langle S_1 \rangle \parallel \langle S_2 \rangle \text{ oc } \{q_1 \land q_2\} \]

A (failed) attempt:

\[ \{p_1\} \langle S_1 \rangle \{q_1\} \quad \{p_2\} \langle S_2 \rangle \{q_2\} \]

\[ \{p_1 \land p_2\} \text{ co } \langle S_1 \rangle \parallel \langle S_2 \rangle \text{ oc } \{q_1 \land q_2\} \]

Example instance:

\[ \{x=0\} \langle x=x+1 \rangle \{x=1\} \quad \{x=0\} \langle x=x+2 \rangle \{x=2\} \]

\[ \{x=0\} \text{ co } \langle x=x+1 \rangle \parallel \langle x=x+2 \rangle \text{ oc } \{x=1 \land x=2\} \]

- we can see that the program does terminate, so rule must be wrong
- postcondition should be \( x=3 \)
- problem is interference between processes and preconditions

\[ S_1: \{x=0\} \langle x=x+1 \rangle \{x=1\} \]

\[ S_2: \{x=0\} \langle x=x+2 \rangle \{x=2\} \]

- if \( S_1 \) effects visible to \( S_2 \), \( S_2 \) starts with \( x=1 \)
- if \( S_2 \) effects visible to \( S_1 \), \( S_1 \) starts with \( x=2 \)

\( \Rightarrow \) weaken preconditions with other processes' post conditions
$S'_1$: \{\text{x==0 V x==2} \} \langle x=x+1 \rangle \{\text{x==1 V x==3} \}$

$S'_2$: \{\text{x==0 V x==1} \} \langle x=x+2 \rangle \{\text{x==2 V x==3} \}$

Second attempt

\{\text{x==0 V x==2} \} \langle x=x+1 \rangle \{\text{x==1 V x==3} \}$

\{\text{x==0 V x==1} \} \langle x=x+2 \rangle \{\text{x==2 V x==3} \}$

\{(\text{x==0 V x==2}) \land (\text{x==0 V x==1}) \}$

\{\text{co <x=x+1> || <x=x+2> oc (k==1 V x==3) \land (k==2 V x==3)} \}$

\{\text{x==0} \} \text{ co <x=x+1> || <x=x+2> oc \{x==3 \}$

- Problem with $S'_1$ and $S'_2$ was that they interfered with each other's preconditions ($P_1$, $P_2$)
  - If $P_1$ holds, running $S'_2$ may cause it not hold
  - If $P_2$ holds, running $S'_1$ may cause it not hold

- $S'_1$ and $S'_2$ do not have this problem
**Rule for concurrent execution**

\[ \{ P_0 \} S_0 \{ a_0 \} \ldots \{ P_n \} S_n \{ a_n \} \text{ are interference-free!} \]

\[ \{ P_0 \land \ldots \land P_n \} \text{ co } S_1 \| \ldots \| S_n \text{ co } \{ a_1 \land \ldots \land a_n \} \]

---

**Interference freedom**

- a process \( P_i \) interferes with process \( P_j \)
  - if it (may) execute an assignment
    - that invalidates an assertion of \( P_j \),
      - either pre or postcondition assertions.
  - essentially, invalidates the proof of
    - the Hoare-triple of the process code.

**Formally:**

- **assignment action** is a single assignment
  - or an await statement that contains
    - (many) assignments.
  - a **critical assertion** is a pre- or post
    - condition not in an await statement.

---

**Det Non-interference**

- \( S \) a statement in a process, \( \text{pre}(S) \)
  - its precondition
- \( C \) critical assertion in another process
- \( S \) does not interfere with \( C \) if

\[ \vdash \{ C \land \text{pre}(S) \} \ S \{ C \} \]

\( C \) is invariant w.r.t execution of \( S \).
Example
\[
\{ x = 0 \} \quad \vdash \quad P_1 \quad S_1 \quad Q_1
\]
\[
\{ x = 0 \lor x = 2 \} \quad \vdash \quad \{ x = 1 \} \quad \{ x = 1 \lor x = 3 \}
\]
\[
\vdash \quad \{ x = 0 \lor x = 1 \} \quad \{ x + 1 \} \quad \{ x = 2 \lor x = 3 \}
\]
\[
\{ x = 3 \} \quad P_2 \quad S_2 \quad Q_2
\]
\[
P_1, P_2, Q_1, Q_2 \quad \text{critical assertions}
\]
- Does \( S_2 \) interfere with \( P_1 \)?
  - We take \( \mathcal{C} = P_1 \) and show that
  \[
  \vdash \quad \mathcal{C} \land \text{pre}(S_2) \quad \mathcal{S}_2 \quad \mathcal{C} \quad \mathcal{C}
  \]
- \( \mathcal{S}_2 \) interferes with \( Q_1 \)
- \( \vdash \quad \mathcal{S}_2 \)
- \( \vdash \quad \mathcal{P}_2 \)
- \( \vdash \quad \mathcal{Q}_2 \)

Example
\[
S_1: \quad \{ x = 0 \} \quad \{ x = x + 1 \} \quad \{ x = 1 \}
\]
\[
S_2: \quad \{ x = 0 \} \quad \{ x = x + 2 \} \quad \{ x = 2 \}
\]
\[
\{ x = 2 \}
\]
\[
\{ x = x + 2 \}
\]
\[
\{ x = 2 \}
\]
\[
\vdash \quad \{ C \land \text{pre}(S) \} \quad S \quad \{ C \}
\]
\[
\{ v = 0 \land x = 0 \} \quad \{ x = x + 2 \} \quad \{ x = 6 \}
\]
\[
\vdash \quad C \quad \text{pre}(S_2) \quad C \quad \text{pre}(S_1)
\]
\[
\text{Not True!}
\]
- Weaken assertions (example from above)
\[
S'_1: \quad \{ x = 0 \lor x = 2 \} \quad \{ x = x + 1 \} \quad \{ x = 1 \lor x = 3 \}
\]
\[
S'_2: \quad \{ x = 0 \lor x = 1 \} \quad \{ x = x + 2 \} \quad \{ x = 2 \lor x = 3 \}
\]
Avoiding Interference: Disjoint Variables

- **Reminder:**
  - **V set:** global variables a process refers to (\(R \cup W\))
  - **W set:** global variables a process writes to

- **Reference set:***
  - global variables in critical assertions of a process

- Let \(S_1\) and \(S_2\) be statements in processes \(P_1\) and \(P_2\), respectively.

There is no interference if:

- \(W(S_1)\) is disjoint from reference set of \(S_2\), and
- \(W(S_2)\) is disjoint from reference set of \(S_1\).

\[
\{ x = 0 \} \langle x = x + 1 \rangle \{ x = 1 \}
\]
\[
\| \{ y = 0 \} \langle y = y + 1 \rangle \{ y = 1 \} \|_c
\]

- this happy situation sometimes happens
- matrix algorithms where each process operates on a block of data
- independent database updates
Avoiding interference: global invariants

A global invariant
- refers to global variables only
- holds initially
- preserved by all assignments
  (this includes < > blocks)
- if critical conditions can be expressed in
  the form

\[
\{ I \land L \} 
\]

where

\( I \) global invariant
\( L \) only refers to the process' local variables

then no interference

\( \implies \) each process' triples can be
verified in isolation
Avoiding interference: hide critical conditions

- hide intermediate states, and assertions about them with mutual exclusion

\[ C_0 \ldots \| S_i \ldots \| S_1; \{ c \}; S_2; \ldots \| C \]

- \( S \) might interfere with \( C \), so hide it

\[ C_0 \ldots \| S_i \ldots \| S_1; \{ c \}; S_2; \ldots \| C \]

- also can make \( S \) wait for a stronger condition

\[ C_0 \ldots \| \text{await} (!c \lor B) S_i \ldots \| S_1; \{ c \}; S_2; \ldots \| C \]

- \( B \) is the set of states from which \( S \) makes \( C \) true

i.e.: \( \{ B \}; S \{ c \} \)

- if \( B \) is true, \( S \) does not interfere with \( C \)
- if \( B \) is false, \( S \) has to wait until

\( C \) is false, in which case \( S_2 \) cannot execute (until after \( S \) makes \( C \) true)
Example: Producer consumer

```c
int buf, p = 0, c = 0;
{ pc : c <= p <= c+1 \land a[0:n-1] == A[0:n-1] \land 
  (p == c+1) \Rightarrow (buf == A[p-1]) }  
```

Process producer {
```c
int a[n];
{ ip : pc \land p <= n }  
while (p < n) {
  [ pc \land p < n ]
  < await (p==c) ; >
  [ pc \land p < n \land p == c ]
  buf = a[p];
  [ pc \land p < n \land p == c \land buf = A[p] ]
  p = p + 1;
  [ ip ]
}
[ pc \land p == n ]
psn \land ! (p < n)
```
}

Process consumer {
```c
int b[n];
{ ic : pc \land c <= n \land b[0:c-1] == A[0:c-1] }  
while (c < n) {
  [ ic \land c < n ]
  < await (p >= c) ; >
  [ ic \land c < n \land p >= c ]
  b[c] = buf;
  [ ic \land c < n \land p >= c \land b[c] == A[c] ]
  c = c + 1;
  [ ic ]
}
[ ic \land c == n ]
```
}

Final Postcondition:
```c
[ pc \land p == n \land c == n \land b[0:(c-1)] == A[0:p-1] ]
```
Monitor invariant

monitor m {
  variables
  initialization
  procedures
}

- Monitor invariant I describes the 'good' states of the variable's tuple
- I must hold
  - after initialization
  - after every procedure
  - when execution suspends at wait
- I can be assumed to hold
  - when procedure starts
  - when procedure continues after wait

Axioms for signal/signal_all/wait

\{ I^3 \} \text{ wait}(cv) \{ I \}
\{ p^3 \} \text{ signal}(cv) \{ p^3 \}
\{ p^3 \} \text{ signal_all}(cv) \{ p^3 \}

These are valid if I does not refer to cv

Example reader/writer

\[ I \equiv (nr == 0 \lor nw == 0) \land nw \leq 1 \]

procedure request_read () {
  \{ I^3 \}
  while (nw > 0) {
    \{ I \land nw > 0 \}^2
    \{ I^3 \} \text{ wait}(rok); \{ I^3 \}^4
    \{ I \land nw == 0 \}^5
    \{ I \land (nr+1/nr) \}^6 \equiv \{ \text{proof=0 v nw=0} \land nw \leq 1 \}
    nr = nr + 1;
  }
  \{ I^3 \}
}
procedure request_write () {
  {[=]
    while (nr > 0 || nw > 0) {
      {I ^ (nr > 0 V nw > 0)}
      {I} wait (wk);
      {[=]}
      {I ^ (nr == 0 ^ nw == 0)}
      {I} [nw + 1 \(/\)nw] = {(nr == 0 V nw + 1 == 0) ^ nw + 1 \leq 1}
      nw = nw + 1;
      {I}
    } proof obligations:
    I ^ (nr > 0 V nw > 0) \Rightarrow I
    I ^ (nr == 0 ^ nw == 0) \Rightarrow (nr == 0 V nw + 1 == 0)
    ^ nw + 1 \leq 1
    <\Rightarrow I ^ (nr == 0 ^ nw == 0) \Rightarrow (nr == 0 V nw == -1)
    ^ nw \leq 0
  }
}

Reasoning about # processes waiting

- Condition variable API does not expose
  number of waiting processes
- We can still use such information in
  assertions, for cv, let #cv be num of proc
    waiting
- Axioms:
  {I [\#cv + 1 \(/\)\#cv]} wait (cv) {I}
  {((\#cv == 0) \Rightarrow P)
    ^ ((\#cv \neq 0) \Rightarrow P[\#cv - 1 \(/\)\#cv])} signal (cv) {P3}
  {P[0 \(/\)\#cv]} signal_all (cv) {P3}
- If we know that #cv > 0 before signal,
  prec simplifies to {P[\#cv - 1 \(/\)\#cv]}
Example: Semaphore verification, see fig. 5.2

```c
monitor sem {
    int s = 0;
    cond pos;
}
```

\[
I = S \geq 0 \land (S \geq 0 \Rightarrow \#\text{pos} = 0)
\]

- if \(S \geq 0\), no process is waiting

```
procedure P() {
    \{I\}
    if (S == 0) \{I \land S == 0\} \[1\]
    \{I[\#\text{pos} + 1/\#\text{pos}]\} \text{wait}(\text{pos}); \{I\}
    else \{I \land S \neq 0\} \[3\]
    \{I\[s-1/s]\} \[4\] s = s - 1; \{I\}
\}
```

```
procedure V(C) {
    \{I\}
    if (empty(\text{pos})) \{I \land \#\text{pos} == 0\} \[1\]
    \{I[s+1/s]\} \[2\] s = s + 1; \{I\}
    else \{I[\#\text{pos} - 1/\#\text{pos}]\} \[4\] \text{signal}(\text{pos}); \{I\}
\}
```

- This solution avoids a check when a process
  is woken up, and a signal when no processes waiting
  compared to: P, while (S==0) wait(pos), V: s=s+1, \text{signal}(\text{pos});
- Verifiably has the same invariant
- May not be correct in all systems
- C++ standard: A thread can be
  woken up for no reason!
  Even if signal was
  not called
Software transactional memory

- Idea: declare blocks of code that should execute atomically
- no locking, execute optimistically, keep log of memory reads and writes
- at the end of an atomic block, compare memory with logs
  - if discrepancies, roll back and try again

- Good model for programmer
  - no need to keep track of which lock to use
  - no deadlock
  - compositional
  - just declare a block atomic

\[
\text{lock}(k) \quad \Rightarrow \quad \text{atomic \&
\text{dostuff} \\
\text{unlock}(k) \quad \Rightarrow \quad \text{dostuff}
\]

- STM coming to /is in
  - C++, Haskell, HPC languages Chapel, X10
  - Clojure, Erlang
- Most languages provide libraries for STM
  - Java, Scala, Python, Perl, Javascript, C#
Language support needed to translate memory accesses:

```c
atomic {
    a.x = x1;
    a.y = x2;
    if (a.z == 0) {
        a.x = 0;
        a.z = x3;
    }
}
```

```c
⇒

+X Begin();
+X W (&a.x, x1);
+X W (&a.y, x2);
if (X R(&a.z) != 0) {
    +X W (&a.x, 0);
    +X W (&a.z, x3);
} +X Commit();
```

Language support needed to make STM use safe(r). Type system can help.

Benefits
- Composable
- Fine-grained ⇒ possible performance benefits
- Performance good if no congestion
- Performance in particular good in distributed environments
  - Logs are local and thus cheap in comparison to communication costs

(Examples elsewhere)
Example
atomic {  
\[ x = x + y \]
\[ z = x + y \]
}

3

1. begin transaction
   - get global version number and increment
   - read version == 7 == rv

2. read x and its version atomically
   \[ x = 10, \quad x.v = 7 \]
   - ok, since \( x.v \leq rv \) (otherwise rollback)
   - add &x to read log

3. read y and its version atomically
   \[ y = 40, \quad y.v = 3 \]
   - ok
   - add y to Rlog

4. add x to Wlog

5. read x
   - since x is in wlog, use that value \( x = 50 \)

6. read y and its version atomically
   \[ y = 40, \quad y.v = 3 \]

7. add z to wlog

8. Try to commit
   - lock wlog (x, z)
   - rollback if fail
   - get write version, increment \( gv \) atomically
     \[ \text{write version (wv)} = 12 \]
   - validate Rlog
     - x in wlog, so already locked, check \( x.v \leq rv \)
     - lock y
     - if fail, rollback
     - check \( y.v \leq rv \)
     - unlock y

9. Write wlog to memory: - write x, unlock x
    - write z, unlock z
Rollback

- happens on any read if
  - variable locked
  - version greater than \( RV \)
- on commit
  - if any wlog variable cannot be locked
  - if any wlog variable locked or version higher than \( RV \)

Early rollback crucial

Consider:

\[
\text{atomic}\{ \\
\text{atomic}\{ \\
\quad \text{if } (x != y) \quad \text{crash}(); \quad \text{if } (x != y) \quad \text{crash}(); \\
\}
\]

3

3

- assume invariant \( x == y \)
- both transactions maintain invariant
- without early rollback, can invoke crash
Relaxed memory order

```
x. store(1)
x. store(2)
x. load()
x. loadC()
x. store(4)
```

Modification order

```
x. store(3)
```

Thread 1

Example

```
atomic <bool> f = False, g = False;
```

Thread 1

```
f = true
```

Thread 2

```
while (!g.load())
    f.load();
```

Q: can f be false?
A: yes
Acquire (release)

\( a = 1 \)
\n\( x = \text{false} \)

\( \text{while}(\neg x \text{load}()) \)
\n\( \text{assert}(a = 1); \quad \text{// ok} \)

\[
\text{atomic} < \text{bool} > \ f = \text{false}, \ j = \text{false} > \quad \text{int} \ n;
\]
\[
[n = 1] \quad f.\text{store} (\text{true, memory-order-release});
\]
\[
[\neg f.\text{load} (\text{memory-order-acquire})]; \quad g.\text{store} (\text{true, memory-order-release});
\]
\[
[\neg g.\text{load} (\text{memory-order-acquire})]; \quad \text{assert} (n = 1);
\]

\text{Consume?}
- \text{currently semantics unclear}
 Dekker's algorithm

T₁
f = true
if (!g.load()) {
    // CS
    T
}

T₂
f = true
if (!f.load()) {
    // CS
    T
}

- no acquire/release between same atomic
- both threads can enter CS!
- need memory-order seq-cst

f = true
if (!g.load()) {
    // CS
    T
}

f = true
if (!f.load()) {
    // CS
    T
}
Message Passing Concurrency - Distributed Computing

- Part II in book

- So far we have concurrent programs running on a single machine
  - shared (physical) memory

- Now: distributed memory architectures
  - Processor has private memory
  - Communicates with other processors via an "interconnect", network

- Examples:
  - Clusters (PCs on one local network)
  - Grid systems (machines communicating via Internet, sharing computing resources)
  - Cloud computing

- Memory distributed
  - No shared variables

- Processes communicate by
  - Sending messages (that contain data) and receiving messages via shared channels
  - Via RPC or rendezvous
Big Picture

busy waiting

semaphores

monitors

implicit mutual exclusion

each semaphore count carries data

message passing

RPC/rendezvous
Asynchronous message passing via channels

- **Channel**
  - communication path between processes
  - abstraction over physical communication network
  - or even shared memory

- **Channel declaration**
  
  \[
  \text{chan } c \langle \text{type}_1 \text{ id}_1, ..., \text{type}_n \text{ id}_n \rangle;
  \]

- messages are n-tuples of type \( \langle \text{type}_1, \text{type}_2, ..., \text{type}_n \rangle \)

- examples:
  
  \[
  \begin{aligned}
  \text{chan } c_1 \langle \text{int acc id}, \text{int cmd}, \text{int amount} \rangle; \\
  \text{chan } c_2 \langle \text{int size}, \text{char* str} \rangle; \\
  \text{chan } c_3 \langle \text{char* str} \rangle; \\
  \text{chan } c_4 \langle \text{variant} \langle \text{deposit, withdraw, query} \rangle \rangle; \\
  \text{chan } c_5 \langle \text{int c id}, \text{int data} \rangle;
  \end{aligned}
  \]

- **Communication primitives**

  \[
  \begin{aligned}
  \text{send } c \langle e_1, e_2, ..., e_n \rangle; \\
  \text{receive } c \langle \text{var}_1, \text{var}_2, ..., \text{var}_n \rangle; \\
  \text{empty } c; \quad \text{//true if channel empty}
  \end{aligned}
  \]

- all topologies allowed
  - one-to-one
  - one-to-many
  - many-to-one
  - many-to-many
Sending
- if channel c has space,
  deposit message to c and continue
- if channel full, block

Receiving
- if c has at least one message,
  remove it and continue
- if c empty, block

Channel size
- 0: unbuffered channel,
  synchronous communication
- \( n \geq 1 \): buffered channel,
  asynchronous communication
- \( \infty \): unbounded,
  asynchronous communication

Properties of the channel abstraction
- one way: from sender to receiver
- (unbounded) FIFO queue of messages
  - preserves message order
- atomic access to queue
  - sends and receives are atomic
- error-free
  - messages do not get lost or corrupted
- typed
  - send can only send messages that receiver understands
Example

```plaintext
chan c (int);

process A {
  send c (1);
  send c (2);
}

⇒ (x, y) = (1, 2)

process B {
  int x, y;
  receive c (x);
  receive c (y);
}

A → B

process A1 {
  send c (1);
}

⇒ (x1, y) = (1, 2) \lor (x, y) = (2, 1)

process A2 {
  send c (2);
}

A1 → c → B

A2

A → B
```
Filters one-way interaction pattern
- a process that receives messages from input channels
- sends messages to output channels
- output is function of the input and initial state

- Example: sorting filter
  - Process sort specification as a predicate
    \[ \text{SORT: } (\forall i, 1 \leq i < n. \text{ sent } [i] \leq \text{ sent } [i+1]) \land \text{ values sent to output are a permutation of those received from output) } \]

- Implementation sketch
  
  process Sort {
    receive all elements from channel input
    sort elements
    send sorted elements to output
  }

- Often desire is a network of processes that interact to solve a problem

  \Rightarrow \text{ merge network}
- Merge two sorted lists

\[ \text{MERGE:} \quad \text{in1, in2 empty} \land \]
\[ \text{Sent}[n+1] = \text{Eos} \land \]
\[ \forall i, 1 \leq i < n. \text{sent}[i] \leq \text{sent}[i+1] \land \]
\[ \text{values in out a permutation of those in in1, in2} \]

\[ \text{chan in1 (int), in2 (int), out (int);} \]

\[ \text{process Merge \{ \}
\]
\[ \text{int v1, v2;} \]
\[ \text{receive in1 (v1);} \]
\[ \text{receive in2 (v2);} \]
\[ \text{while (v1 != Eos \& v2 != Eos) \{ \}
\]
\[ \text{if (v1 <= v2) \{ send out (v1); receive in1 (v1); \}} \]
\[ \text{else \{ send out (v2); receive in2 (v2); \}} \]
\[ \} \]
\[ \text{if (v1 == Eos) while (v2 != Eos) \{ send out (v2); receive in2 (v2); \}} \]
\[ \text{else while (v1 != Eos) \{ send out (v1); receive in1 (v1); \}} \]
\[ \} \]
\[ \text{Send out (Eos);} \]
\[ \} \]

Sorting network

value 1 \[\rightarrow\] Merge \[\rightarrow\] ...
value 2
... ...
value n-1 \[\rightarrow\] Merge \[\rightarrow\] ...
value n

Sorted stream
Client - Server with message passing

```
chan request (int clientID, typesOfInput);  
chan reply [n] (types of results);

process server {
    int clientID;
    // initialization, permanent variables
    while (true) {  // Loop invariant = monitor invariant
        receive request (clientID, input values);
        // body
        send reply [clientID, result values];
    }
}

process client [i = 0 to n-1] {
    send request (i, args);
    receive reply [i] (res_args);
}
```

Monitor Server {
    permanent variables
    initialization code;
    procedure op (input_args) { //body; }  
}

Correspondence between (passive) monitors and (active) servers

Monitors
- Permanent variables --> local server variables
- procedure identifier --> request channel + operation kind
- procedure call --> send request(); receive reply();
- monitor entry --> receive request();
- procedure return --> send reply();
- wait statement --> save pending request
- signal statement --> retrieve and process pending request
- procedure bodies --> arms of a case statement on operation kind
Monitor vs. Server

```c
monitor resource_allocator {
    int avail = MAX_UNITS;
    set units = initial_values;
    cond free;
    procedure acquire (int id) {
        while (avail == 0) wait (free);
        avail --;
        remove (units, id);
    }
    procedure release (int id) {
        insert (units, id);
        avail ++;
        signal_all (free);
    }
}
```

type op_kind = enum (ACQ, REL);
chan request (int clientID, op_kind kind, int unit_id);
chan reply [n] (int unitID);

Process Allocator {
    int avail = MAX_UNITS;
    set units = ... // init
    queue pending;
    int clientID, unitID, op_kind kind;
    while (true) {
        receive request (clientID, kind, unitID);
        if (kind == ACQ) {
            if (avail > 0) { // serve request now
                avail --;
                remove (units, unitID);
                send reply [clientID] (unitID);
            } else {
                insert (pending, clientID); // save request to pending queue
            }
        } else { // kind == REL
            if (empty (pending)) { // no pending, restore resource
                avail ++;
                insert (units, unitID);
            } else {
                remove (pending, clientID); // give the released
                send reply [clientID] (unitID); // resource unit to
                // the first pending
            }
        }
    }
}
```
Conversational continuity - Sessions

- Often client/server communications not just simple request/reply, but longer discussions
  - a sequence of messages that follow a protocol
  - one discussion == Session

Example

```
client

open file req.
filehandle f
f.rex

data
f.rex

data
f.wtr

close

file server
```

Two options.

1. Server maintains Session, gives session ID to client
   - must have timeout for closing dangling sessions
2. Stateless server
   - client maintains session data, resends all relevant data with each request
Interacting Peers (some distributed programming)

- Various consensus algorithms often needed in distributed algorithms
- Many communication pattern choices
- Simple but characteristic example:
  - every process has a value
  - compute min and max
  - communicate min and max to all processes
- Three solutions

```
Centralized

chan values(int), results[n][int mi, int ma];
process P[0] {
    int v; // assume initialized
    int new, mi = v, ma = v;
    for [i=1 to n-1] {
        receive values (new);
        mi = min (mi, new);
        ma = max (ma, new);
    }
    for [i=1 to n-1] {
        send results [i] (mi, ma);
    }
}

process P[i=1 to n-1] {
    int v;
    int mi, ma;
    send values(v);
    receive results [i] (mi, ma);
}
```

```
Symmetric

channel values [n] (int);

process P[i = 0 to n-1] {
    int v;
    int new, mi = v, ma = v;
    for [j = 0 to n-1] send values[i] (v);
    for [j = 0 to n-1] {
        receive values[i] (new);
        mi = min (mi, new);
        ma = max (ma, new);
    }
}

Ring

channel results [n] (int mi, int ma);

process P[0] {
    int v;
    int mi = v, ma = v;
    send results[1] (mi, ma);
    receive results[0] (mi, ma);
    send results[1] (mi, ma);
}

process P[i = 1 to n-1] {
    int v;
    int mi, ma;
    receive results[i] (mi, ma);
    mi = min (mi, v);
    ma = max (ma, v);
    send results [(i+1) % n] (mi, ma);
    receive results[i] (mi, ma);
    if (i < n-1) send results[i+1] (mi, ma);
}
Synchronous message passing

- New primitive for sending:
  sync-send \( c(e_1, e_2, \ldots, e_n) \)
  - Blocks until message is received by some process
  - Effect: Sender & receiver synchronize on sending & receiving a message

- Benefits
  - Channels have fixed size (needs no memory for message data)
    - Receiver has at most 1 pending message
    - Sender has at most 1 undelivered message

- Drawbacks
  - Reduced parallelism
  - Deadlock risk higher
    - Example of deadlock

```
chan in1(int), in2(int);
process P1 {
    int v1 = 1, v2;
    sync-send in2(v1);
    receive in1(v2);
    3
}
process P2 {
    int v1, v2 = 2;
    sync-send in1(v2);
    receive in2(v1);
    3
}
```

- With async send, no deadlock
Communicating Sequential Processes (CSP)

- Formalism by Tony Hoare, 1978
- Highly influential
  - Occam
  - Ada
  - Go
  - Clojure async core libraries
- Formal reasoning about concurrent systems
  - Space applications
  - Banking
  - ...

Communication sentences

- **Input statement**
  
  \[ B!e \]
  
  "bang" - send value of expression \( e \) to process \( B \)

- **Output statement**
  
  \[ A?x \]
  
  "query" - receive value to variable \( x \) from process \( A \)

- Communication takes place when an input and output statement match.
- Processes synchronized during communication

- General forms of input/output statements
  
  \[
  \text{Dest}! \text{port}(e_1, \ldots, e_n); \\
  \text{Source} ? \text{port}(x_1, \ldots, x_n);
  \]

  - if process has 1 port
  - if 1 e or v
  - Port names a channel some process pair can use multiple channels
  - Source[++] ? port \((x_1, \ldots, x_n)\); - read from any process in an array of processes
Example
- copy characters from East process to West process

process copy {
  char c;
  do true \rightarrow West? c; East! c; od

3
 delays until West ready to send
 delays until East ready to receive

Guarded commands notation (Dijkstra)

B \rightarrow S
- B guard, a boolean expression
- S a list of statements
- Used with if and do statements

if B_1 \rightarrow S_1
... B_n \rightarrow S_n
fi
- if all guards B_i are false, skip
- otherwise, execute one S_i whose guard B_i is true

do B_1 \rightarrow S_1
... B_n \rightarrow S_n
od
- if all guards B_i are false, exit loop
- otherwise, execute one S_i whose guard B_i is true, and repeat
Example (do)
- GCD Server
  process GCD {
    int id, x, y;
    do
      true →
      client [id] ? args (id, x, y);
      do x > y → x = x - y;
        [|] x < y → y = y - x;
        od
      client [id] ! result (x);
    od
  } od

- GCD Client
  int i = client.id, r;
  GCD ! args (i, 56, 47);
  GCD ! result (r);

Guarded Communication Statements
- Input and output statements are blocking.
- Guarded communication statements allow for choosing ready statements among many.

B; C → S
- B boolean expression - if omitted, assumed true
- C communication statement
- S list of statements

B; C is the guard:
- succeeds if B true and C would not delay
- fails if B false
- blocks if B false but C would delay
Guarded communication statements also appear in if and do

- **if-statement**

```plaintext
if B_1; C_1 \rightarrow S_1
[ ] B_2; C_2 \rightarrow S_2
\vdots
[ ] B_n; C_n \rightarrow S_n
fi
```

- **semantics**:
  - if all guards fail, skip
  - if at least one guard succeeds, execute S_i
    - non-deterministic choice
  - otherwise, wait until one of the blocking guards succeeds, then execute its S_i

- **do-statement**

```plaintext
do B_1; C_1 \rightarrow S_1;
[ ] B_2; C_2 \rightarrow S_2;
\vdots
[ ] B_n; C_n \rightarrow S_n;
```

- same as if, but repeat until all guards fail

**Example**:  

process Copy_1 \{  
  char \ c;  
  do West! \ c \rightarrow East! \ c \ od
\}

```
process Copy_2 \{  
  char \ c_1,c_2;  
  West? \ c_1;  
  do West? \ c_2 \rightarrow East! \ c_1 ; \ c_1 = C_2;  
  [ ] East? \ c_1 \rightarrow West? \ c_1;  
```

- unbuffered copy (or 1 char buffer)
- 2 char buffer

Loop

Invariant: \ c_1 received from west
Buffered Copy

\[
\text{process } \text{Copy}(N) \{ \\
\quad \text{char buffer }[0..N]; \\
\quad \text{int front } = 0, \text{rear } = 0, \text{count } = 0; \\
\quad \text{do } \text{count } < N \text{; } \text{while } \text{? buffer }[\text{rear}] \\
\quad \quad \rightarrow \text{count } \rightarrow \text{; } \text{rear } = (\text{rear } + 1) \% N; \\
\quad \} \quad \text{count } > 0 \text{; } \text{Fast } ? \text{ buffer }[\text{front}] \\
\quad \quad \rightarrow \text{count } \rightarrow \text{; } \text{front } = (\text{front } + 1) \% N; \\
\}
\]

Example (multiple ports)

\[
\text{process } \text{Allocator} \{ \\
\quad \text{int avail } = \text{MAX}\text{UNITS}; \\
\quad \text{set units } = \ldots \text{/initialize} \ldots ; \\
\quad \text{int index, unit id; } \\
\quad \text{do } \text{avail } > 0 \text{; } \text{clients }[\text{x}] \text{? acquire }[\text{index}] \rightarrow \\
\quad \quad \text{avail } \rightarrow \text{; } \\
\quad \quad \text{remove }(\text{units}, \text{unit id}); \\
\quad \quad \text{clients }[\text{index}] \text{! reply }[\text{unit id}]; \\
\quad \} \quad \text{clients }[\text{x}] \text{? release }[\text{index, unit id}] \rightarrow \\
\quad \quad \text{avail } \rightarrow \text{; } \\
\quad \quad \text{insert }[\text{units, unit id}]; \\
\}
\]

- nicely concise
- no need to merge acquire and release messages to one channel
- no need to store pending requests
  - acquire messages are received only when resources are available
  - clients block
About deadlocks

Guarded communication statements can (in many cases) naturally avoid deadlocks.
Consider the example of 2 processes exchanging information that deadlocked with synchronous message passing; in CSP version there is no deadlock

```
process P1 {
    int v1 = 1, v2;
    if P2!v1 → P2?v2;
    [ ] P2?v2 → P2!v1;
    fi
}
```

```
process P2 {
    int v1, v2 = 2;
    if P1!v2 → P1?v1;
    [ ] P1?v1 → P1!v2;
    fi
}
```

Example (Sieve of Eratosthenes)

```
process Sieve [1] {
    int P = 2;
    for [i = 3 to N by 2] Sieve [2] ! i;
}
```

```
process Sieve[ i = 2 to L ] {
    int p, next;
    Sieve[ i - 1 ] ? p;
    do Sieve[ i - 1 ] ? next → Sieve[ i ] ! next;
        if (next % p != 0) → Sieve[ i ] ! next;
    od
}
```
Formal CSP (modern version, 1985)
- formal language for modeling concurrent systems and verifying properties
- features (in addition to commands):
  - prefixing (sequencing)
  - guarded alternatives (non-deterministic choice)
  - recursion
- functional, rather than imperative
- each process characterized by its communication events with environment or other processes

Example traffic signal communication patterns
red → green → STOP

LIGHT = green → red → LIGHT
- many possible implementations for the same specification
- above events sent to the environment

COPY_1 = West ? c: char → East! c → COPY_1
- here communication events with another process

COPY = West ? c1: char → COPY_2(c1)
COPY(c1) = West ? c2: char → East! c1 → COPY_2(c2)
  East! c1 → West ? c1: char → COPY_2(c1)
- 2 character buffer
GCD = Input (? id, x, y) -> GCD (id, x, y);
GCD (id, x, y) =
  if (x = y) then Output (! id, x) -> GCD
  else if (x > y) then GCD (id, x-y, y)
  else GCD (id, x, y-x)
Coroutines

Stack and Coroutine Frames

Assume function \( f() \) calls coroutine \( g(\text{int} \ a) \).

- Prior to call
  - Stack
  - Registers
  - Heap

- Call \( g(1) \)
  - Stack
  - Registers
  - Heap

\( f \): \n\begin{align*}
&x = 1 \\
&\text{ret} = f(x) \text{ return} \\
&f
\end{align*}

\( g \): \n\begin{align*}
&x = 1 \\
&\text{ret} = g(x) \text{ return} \\
&g
\end{align*}
- Coroutine g calls another function h

- h returns
  - pop h's frame, restore g's
  - assume h's return stored in g's local variable y

- h's return
  - pop h's frame, restore g's
  - assume h's return stored in g's local variable y
- `g` suspended

- stack
- registers
- `handle` ...

- heap

- `handle` is a first-class value
  - can be passed to other functions, even threads, say to function `m`
  - `m` resumes coroutine pointed to by `handle`

- `resume(handle)`
Distributed systems

Diagram showing communication between processes P1, P2, P3, and P4, with arrows indicating input and output directions.
Happens before

\[ P_0 \rightarrow a \rightarrow d \rightarrow P_1 \]
\[ P_1 \rightarrow b \rightarrow c \rightarrow P_2 \]
\[ a \rightarrow d \]
\[ a \rightarrow c \]
\[ b \rightarrow a \]
\[ a \rightarrow b \]
\[ b \rightarrow c \]
\[ a \rightarrow c \]
\[ b \rightarrow d \]
\[ c \rightarrow d \]
\[ b \rightarrow e \]

Logical clocks

\[ P_0 \]
\[ a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f \]
\[ P_1 \]
\[ g \rightarrow h \rightarrow i \]

Vector clocks

\[ P_0 \]
\[ a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f \]
\[ P_1 \]
\[ g \rightarrow h \rightarrow i \]
\[ P_2 \]
Consistent cuts

- \((3,6)\) is not consistent.
  Step 4 in \(P_0\) happens before step 6 in \(P_1\),
  but \(4 > 3\). (Step 4's message delivered after cut)