Concurrency

Mitchell Chapter 14

Concurrent Programs

Two or more sequences of events occur in parallel

- Multiprogramming
  - A single computer runs several programs at the same time
  - Each program proceeds sequentially
  - Actions of one program may occur between two steps of another
- Multiprocessors
  - Two or more processors may be connected
  - Programs on one processor communicate with programs on another
  - Actions may happen simultaneously

process, thread, task:
sequential program running on a processor

Concurrency

- Allows different tasks to proceed at different speeds.
- Multiprogramming
  - Allows one program to do useful work while another is waiting for input
  - More efficient use of a single processor
- Multiprocessing
  - More raw processing power available
  - Introduces new issues
    - Reliability of network
    - Some processor(s) proceeding while another crashes
- Interaction between separate sequential programs raises programming challenges (in both multiprogramming and multiprocessing)

Most of the concepts discussed apply to either kind of concurrency.

Concurrent Programming Languages

- Provide abstractions and control structures defined specifically for concurrent programming.
- Can provide "light-weight" processes that are less costly than operating system processes.
- Can provide portability across operating systems.
  - Historically, concurrent systems written in languages that did not support concurrency. For example, using system calls specific to a particular operating system.
- We study basic concepts using particular example languages.
  - Concurrent ML (as an extension of Standard ML)
  - Java

Basic Concepts in Concurrency

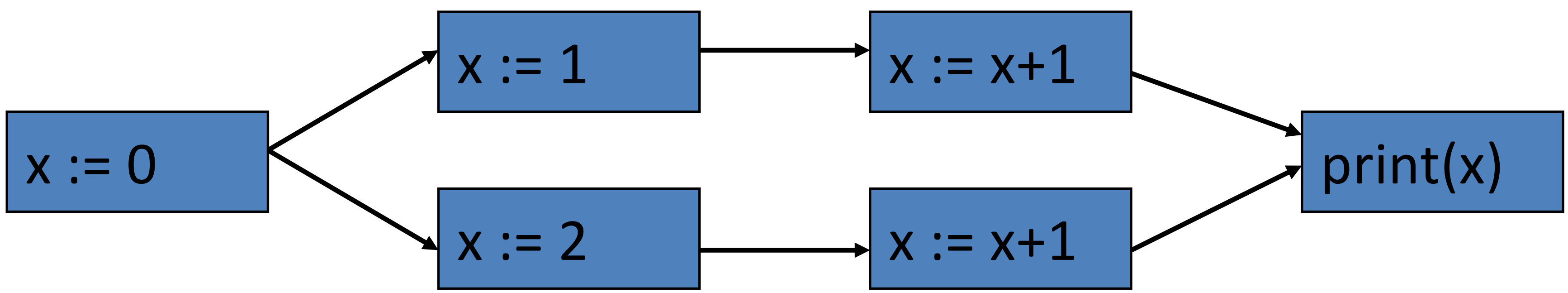
- Execution Order and Nondeterminism
- Communication, Coordination, and Atomicity
- Mutual Exclusion and Locking
- Semaphores
- Monitors

Execution Order and Nondeterminism

- An early limited concurrency primitive in Concurrent Pascal: cobegin…coend statement
- Example

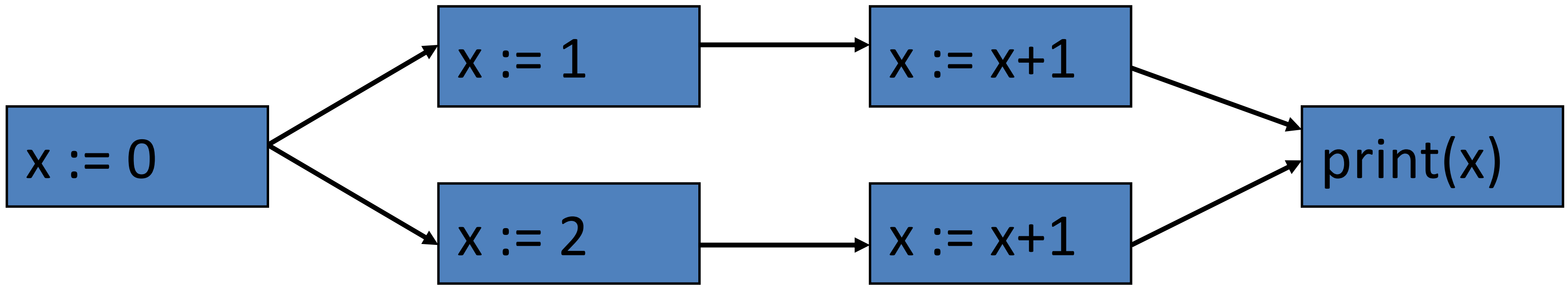
```
x := 0;
cobegin
  begin x := 1; x := x+1 end;
  begin x := 2; x := x+1 end;
coend;
print(x);
```

execute sequential blocks in parallel



Example showing execution on a single processor

Execution Order and Nondeterminism



- Each assignment statement executes *atomically*, which means that there may be more than one low-level machine step, but one assignment is fully completed before allowing another process to execute.
- Some possible orderings *between* statements:

x := 0; x := 1; x := 2; x := x+1; x := x+1; print(x)   Output: 4
x := 0; x := 2; x := 1; x := x+1; x := x+1; print(x)   Output: 3
x := 0; x := 1; x := x+1; x := 2; x := x+1; print(x)   Output: 3
x := 0; x := 2; x := x+1; x := 1; x := x+1; print(x)   Output: 2

Illustrates problem of nondeterminism

Nondeterminism

- A program is *deterministic* if, for each sequence of program inputs, there is one sequence of program actions and resulting outputs.
- A program is *nondeterministic* if there is more than one possible sequence of actions corresponding to a given input sequence.

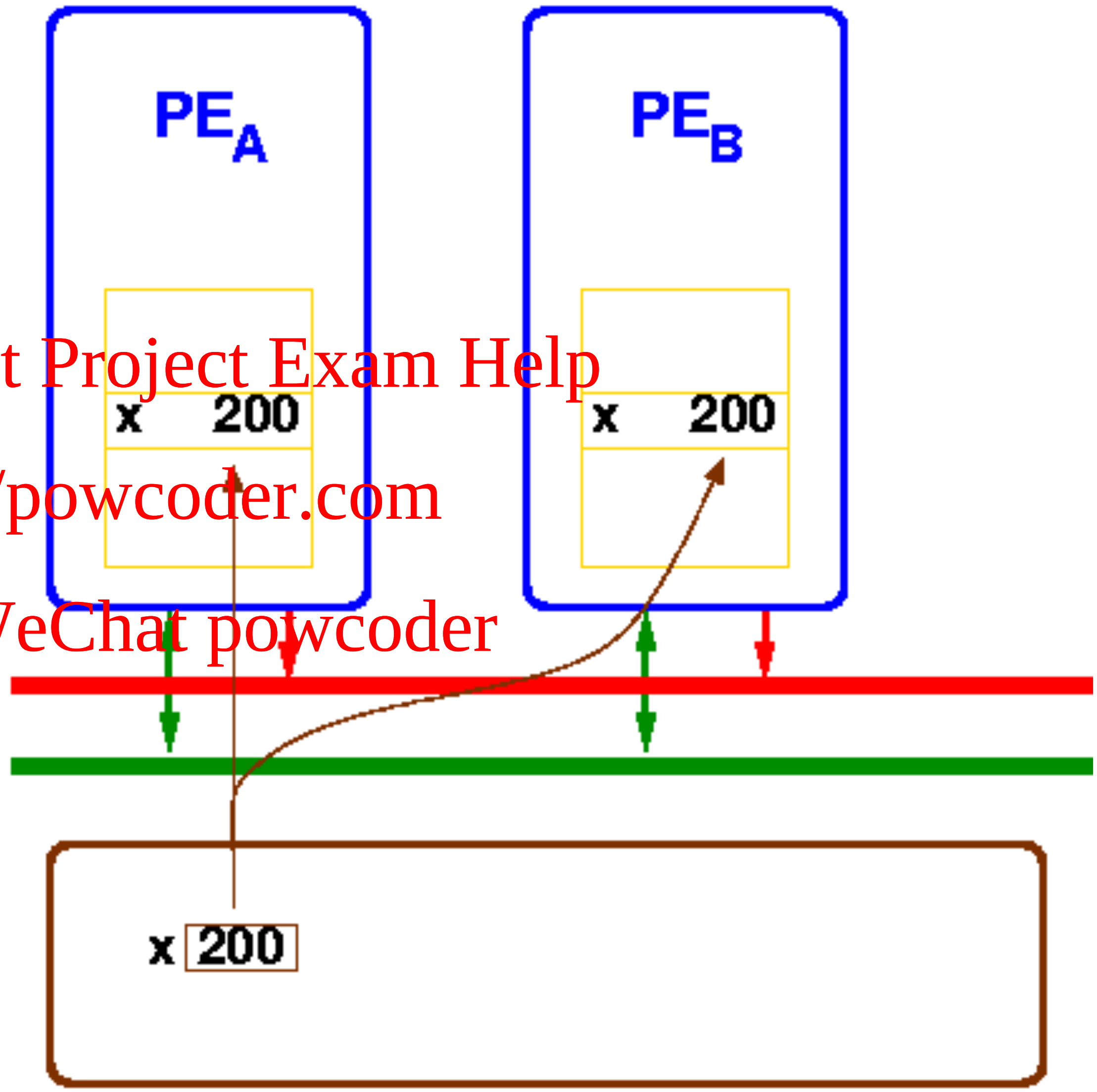
  - Several possible execution orders
  - Different results for different runs, even on the same computer
  - Difficult to design and debug programs

Example Illustrating Problem of Nondeterminism

- Cache coherence protocols in multiprocessors
  - A set of processors share memory
  - Access to memory is slow, can be bottleneck
  - Each processor maintains a memory cache
  - The job of the cache coherence protocol is to maintain the processor caches, and to guarantee that the values returned by every load/store sequence generated by the multiprocessor are consistent with the memory model.

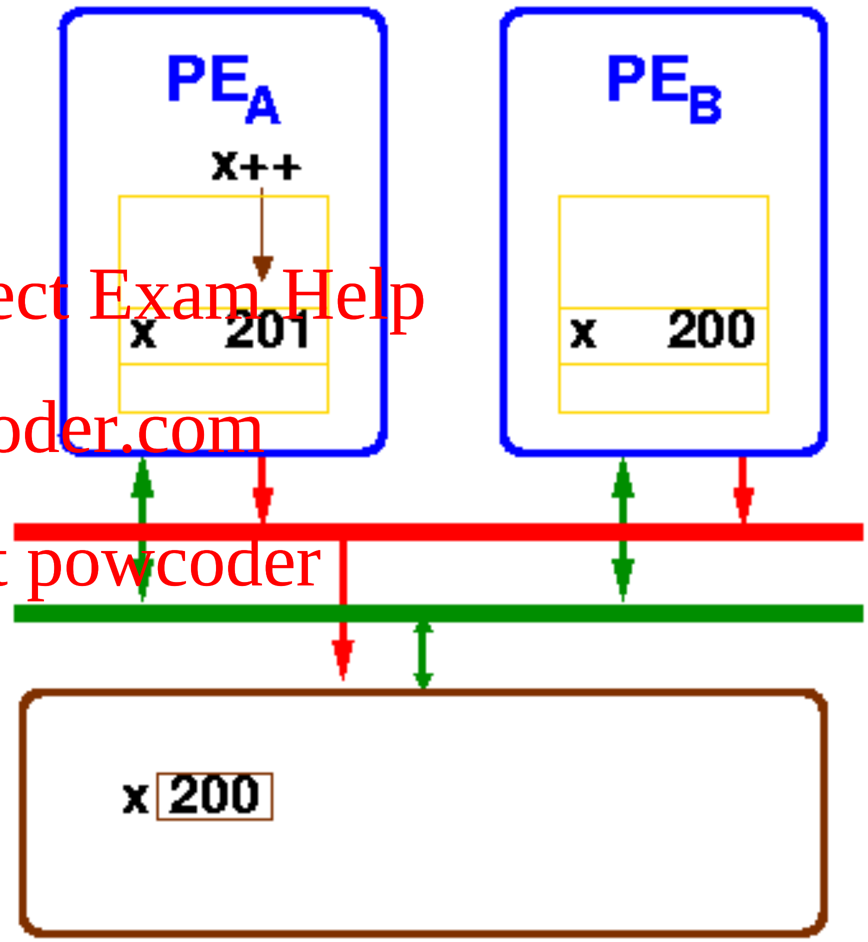
Cache filled by read

- PE_A reads location x
  - Copy of x put in PE_A's cache.
- PE_B also reads x
  - Copy of x put in PE_B's cache too.



Cache modified by write

- PE_A adds 1 to x
  - x is in PE_A's cache, so there's a cache hit
- If PE_B reads x from cache, *may* be wrong
  - OK if program semantics allows PE_B read before PE_A write
- Need protocol to avoid using stale values



Communication, Coordination, and Atomicity

- Mechanisms in programming languages for *explicit concurrency*:
  - Mechanism to initiate and terminate individual sequential processes (all concurrent programming languages provide this capability)
  - *Communication* between processes
    - Buffered communication channels
    - Synchronous communication channels
    - Broadcast
    - Shared variables or objects (as in Concurrent Pascal example)
  - *Coordination* between processes
    - May explicitly or implicitly cause one process to wait for another before continuing
  - *Atomicity* (mentioned earlier)
    - Affects both interaction between processes and handling of errors

Interprocess Communication

- Shared variables is most elementary form
- Can also have shared data structures or files
- Another form is *message passing* with a variety of mechanisms (see next page)

Message Passing

- Buffering
  - If communication is *buffered*, then every data item that is sent remains available until it is received
  - In *unbuffered* communication, a data item sent before the receiver is ready to accept may be lost

- Synchronicity
  - In *synchronous* communication, the sender cannot transmit data unless the receiver is ready to receive it.
  - In *asynchronous* communication, the sending process may transmit a data item and continue executing even if the receiver is not ready to receive the data.
- Message Order
  - A communication mechanism may preserve order of transmission of messages, or it may not.
  - If so, messages will be received in the order they are sent.

Coordination

- Coordination mechanisms allow one process to wait for another or notify a waiting process that it may proceed.
  - Concurrent Pascal cobegin…coend: all processes started at the same cobegin must finish before the statement following coend may proceed.
  - Locking
  - Semaphores

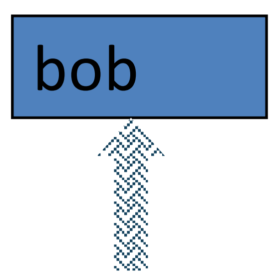
Mutual Exclusion and Locking

- Issue: maintaining consistency of shared data
- Sample action

```
procedure sign_up(person)
   begin
     number := number + 1;
     list[number] := person;
end;
```


- Problem with parallel execution


```
cobegin
   sign_up(fred);
   sign_up(julie);
end;
```



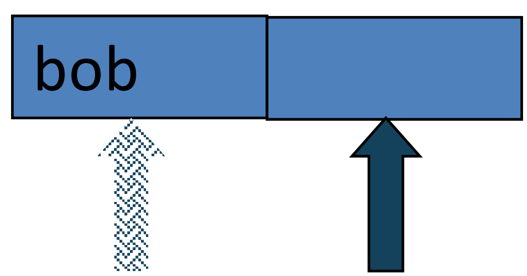
Mutual Exclusion and Locking

- Issue: maintaining consistency of shared data
- Sample action

```
procedure sign_up(person)
    begin
      number := number + 1;
      list[number] := person;
end;
```


- Problem with parallel execution

```
cobegin
    sign_up(fred);
    sign_up(julie);
end;
```



Mutual Exclusion and Locking

- Issue: maintaining consistency of shared data
- Sample action

```
procedure sign_up(person)
    begin
      number := number + 1;
      list[number] := person;
end;
```


- Problem with parallel execution

```
cobegin
    sign_up(fred);
    sign_up(julie);
end;
```

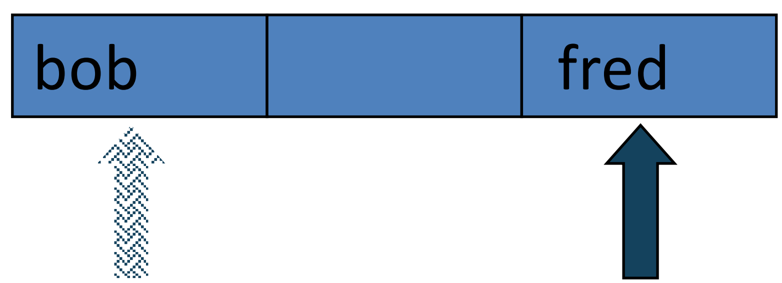
Mutual Exclusion and Locking

- Issue: maintaining consistency of shared data
- Sample action

```
procedure sign_up(person)
    begin
      number := number + 1;
      list[number] := person;
end;
```


- Problem with parallel execution

```
cobegin
    sign_up(fred);
    sign_up(julie);
end;
```



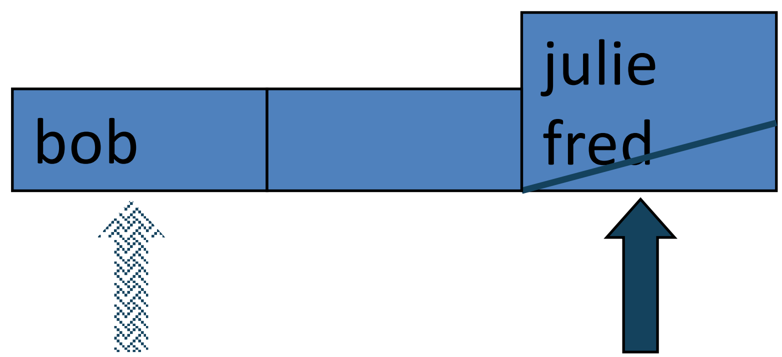
Mutual Exclusion and Locking

- Issue: maintaining consistency of shared data
- Sample action

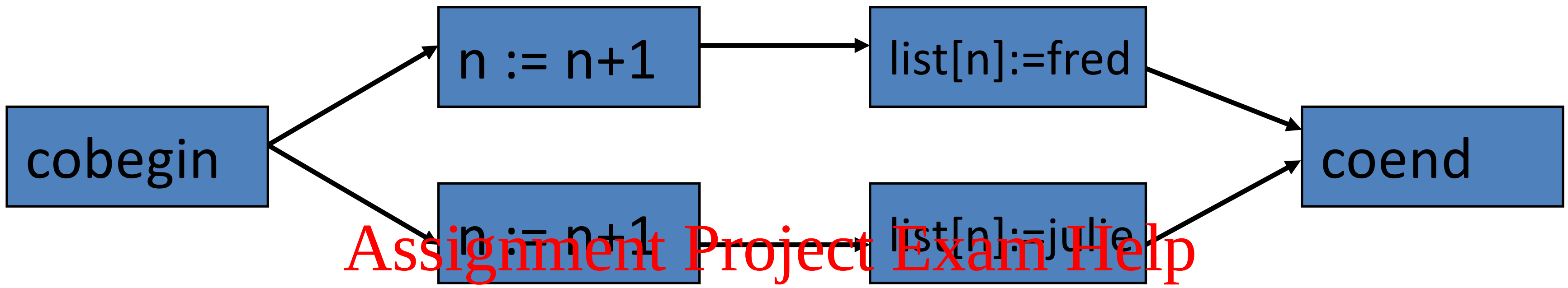
```
procedure sign_up(person)
    begin
      number := number + 1;
      list[number] := person;
end;
```


- Problem with parallel execution

```
cobegin
    sign_up(fred);
    sign_up(julie);
end;
```



Incorrect Ordering



n := n+1; n := n+1; list[n] := fred; list[n] := julie;

Results in an *inconsistent state* because it is not consistent with intended behaviour.

Mutual Exclusion

- Critical Section—a section of a program that accesses shared resources
  - Two processes may access shared resource
  - Inconsistent behavior if two actions are interleaved
- Mutual Exclusion
  - One process at a time may be in its critical section
  - Progress: If no processes are in their critical section and some process wants to enter a critical section, it becomes possible for one waiting process to enter its critical section.
  - Bounded waiting: If one process is waiting, there must be a bound on the number of times that other processes are allowed to enter their critical sections before this one.

Locks and Busy Waiting

- Example using wait and signal actions

```
<initialze concurrency control>
cobegin
    begin
        <wait>
        sign_up(fred);  // critical section
        <signal>
    end;
    begin
        <wait>
        sign_up(julie);   // critical section
        <signal>
    end;
end;
```


Need atomic operations to implement wait

Lock and Signal Implemented as Integers

```
lock := 0;
cobegin
   begin
        while lock=1 do end;  // wait until lock is 0
        lock := 1;  // set lock to enter critical section
        sign_up(fred);   // critical section
        lock := 0;  // release lock
   end;
   begin
        while lock=1 do end;  // wait until lock is 0
        lock := 1;  // set lock to enter critical section
        sign_up(julie);    // critical section
         lock := 0;  // release lock
   end;
end;
```

Lock and Signal Implemented as Integers

```
while lock=1 do end;  // wait until lock is 0
lock := 1;  // set lock to enter critical section
sign_up(…);  // critical section
```

- Using a loop to wait is called *busy waiting*.

- Problem with using a shared variable for mutual exclusion: operation that reads the value of the variable is different from the operation that sets it.

- It is possible for one process to test the variable and see that the lock is open (lock=0), but before the process can lock other processes out by setting it.   (lock := 1), another process can also see the lock is open and set the variable first.
- Two processes can then call signup at once.

Atomic Test-and-Set Lock (TSL)

- Instruction atomically reads and writes some location
- Common hardware instruction
- Combine with busy-waiting loop to implement mutual exclusion.

Deadlock

- Process may hold some locks while awaiting others
- *Deadlock* occurs when no process can proceed
- Example:
  - Process 1 sets Lock 1 and waits for Lock 2
  - Process 2 sets Lock 2 and waits for Lock 1
- Possible solution: 2-phase locking
  - A process is viewed as a sequence of independent tasks.
  - *Locking phase*: For each task, the process must acquire all the locks that could be needed.
  - *Release phase*: A process must release all locks before proceeding to the next task.
  - There must be an ordering on locks.

Semaphore

- Avoid busy-waiting loop
- Keep queue of waiting processes
- A standard *semaphore* is represented by an integer value: the maximum number of processes that may enter a critical section at the same time
- Scheduler has access to semaphore; process sleeps
- Disable interrupts during semaphore operations
  - OK since operations are short
- Processes call **`wait`**, and then must call **`signal`** when finished, otherwise deadlock could occur.

Semaphore Wait

```
// This procedure must be executed atomically.

procedure wait (s:Semaphore)
    begin
        if s.value > 0 then
            s.value := s.value – 1;  // Enter section and
                                     // decrement counter
        else
            suspend_on (s.queue); // Wait for other
                                  // processes to finish
    end;
```

Semaphore Signal

```
// This procedure must be executed atomically.

procedure signal (s:Semaphore)
    begin
        if length(s.queue) = 0 then
                s.value := s.value + 1;  // Increase count
                                         // allowing other
                                         // processes to enter
        else
                allow_one_process (s.queue);
                                         // Wake up one
                                         // suspended process
    end;
```

Monitor

- Synchronized access to private data
- Responsibility for synchronization placed on operations that access data
- Combines:
  - private data
  - set of procedures (methods)
  - synchronization policy
    - At most one process may execute a monitor procedure at a time; this process is said to be *in* the monitor.
    - If one process is in the monitor, any other process that calls a monitor procedure will be delayed.
- Terminology: *synchronized object*

Concurrent Language Examples

- Language Examples
  - Cobegin/coend
  - Actors (not covered in this class)
  - Concurrent ML
  - Java
- Main Features to Compare
  - Threads
  - Communication
  - Synchronization
  - Atomicity

Properties of cobegin/coend

Concurrent Pascal

- Advantages
  - Create concurrent processes
  - Communication: shared variables
- Limitations
  - Mutual exclusion: none
  - Atomicity: not much (an assignment statement is atomic)
  - Number of processes is fixed by program structure
  - Cannot abort processes
    - All must complete before parent process can go on

Concurrent ML

An Extension to Standard ML

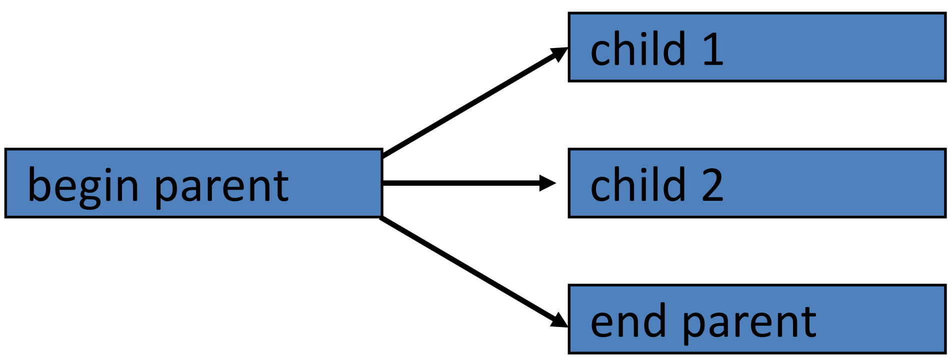
- Threads
  - New type of entity
- Communication
  - Synchronous channels
  - Communication and coordination combined
- Synchronization
  - Channels
  - Events (allows users to define their own communication and synchronization abstractions)
- Atomicity
  - No specific language support

Threads

- Thread creation
  - spawn : (unit → unit) → thread_id
- Example code

```
CIO.print  "begin parent\n";
    spawn (fn () => (CIO.print "child 1\n";));
    spawn (fn () => (CIO.print "child 2\n";));
CIO.print "end parent\n"
```


- Result



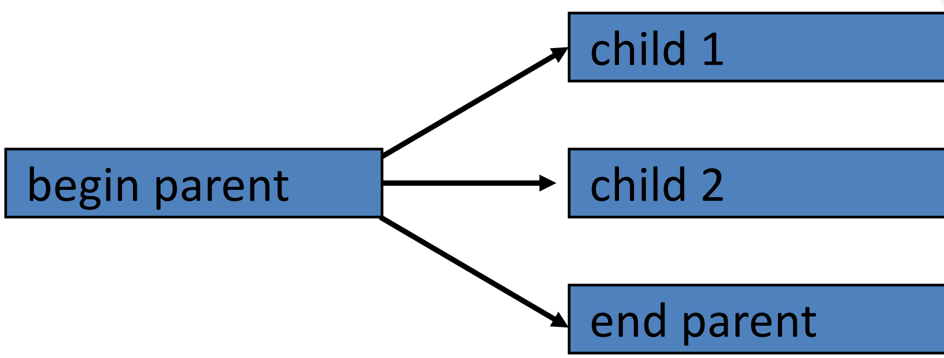
Threads

- Thread creation
  - spawn : (unit → unit) → thread_id
- Example code

```
CIO.print  "begin parent\n";
    spawn (fn () => (CIO.print "child 1
    spawn (fn () => (CIO.print "child 2
CIO.print "end parent\n"
```

- Result

No restriction on when to terminate parent or child. Either can terminate before the other without affecting the other. Prints “begin parent” first, and then prints the other 3 in any order.



Infinite Threads

forever: 'a $\rightarrow$ ('a $\rightarrow$ 'a) $\rightarrow$ unit

- (forever x_0 f) computes:

  $x_1 = (f\ x_0)$ $x_2 = (f\ x_1)$ $x_3 = (f\ x_2)$ …

- The values $x_1, x_2, x_3, \ldots$ are discarded

- f may communicate with other threads

- This thread can be terminated by other CML primitives, otherwise it loops forever.

Channels

- Channel creation (for communicating values of type `'a`)
  - `channel : unit → 'a chan`
- Communication
  - `recv : 'a chan → 'a`
  - `send : ('a chan * 'a) → unit`

- Message passing in synchronous
  - Both sender and receiver must be ready to communicate.

  - If one thread executes a `send` and no thread is ready to execute `recv` on the same channel, the sending thread *blocks* (stops and waits) for a thread to execute `recv`.

  - Similarly, if `recv` is executed, it can block if there is no `send`.

- Example
  - If `c` is an `int chan`, then `send(c,3)` sends the integer `3` on channel `c`. Result type is `unit` like an assignment.

Channels

- Example

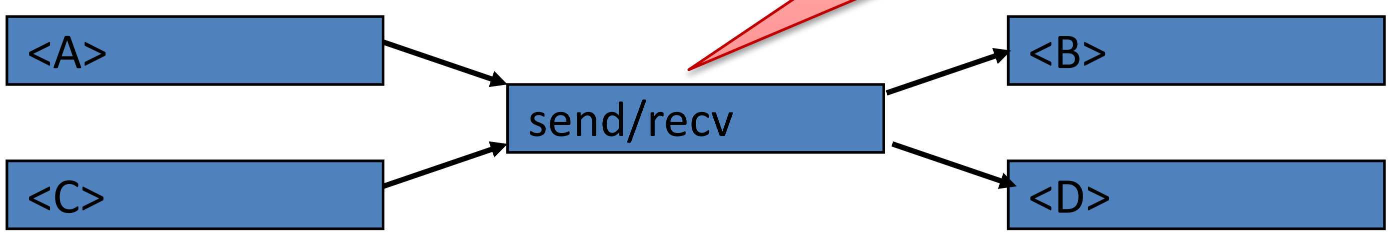
  ch = channel();

  spawn (fn()=> … <A> … send(ch,0); … <B> …);

  spawn (fn()=> … <C> … recv ch; … <D> …);

The synchronous communication causes A and C to execute before B and D.

- Result



Sample CML Program

- Function to create squaring process

```
fun square (inCh, outCh) =
    forever ()  (fn () =>  send (outCh, compute_square (recv inCh)));
```

- Put processes together (assuming that numbers creates a thread that outputs numbers on its argument channel)

```
fun mkSquares () =
let
    val outCh = channel()
    and c1 = channel()
in
    numbers(c1);
    square(c1, outCh);
    outCh
end;
```

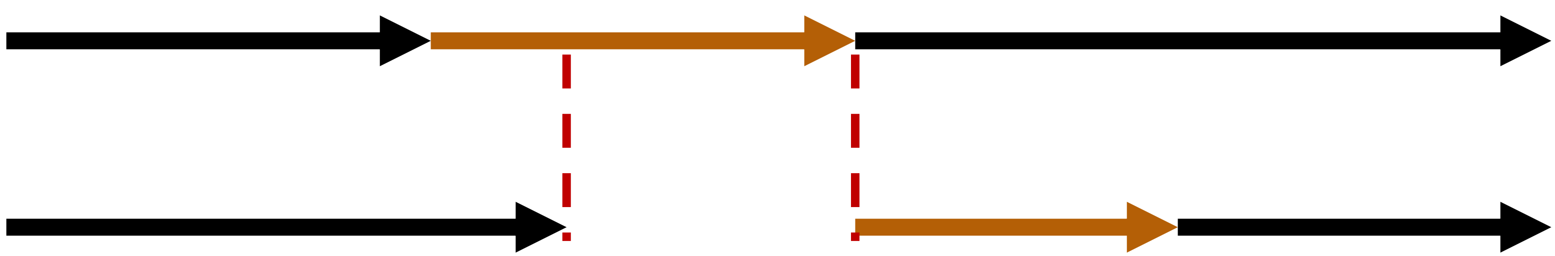

- If a thread has the name of a channel, it can send messages, receive messages, or both.
- If a channel is passed to more than one thread, each can send and receive messages on the channel.

Java Concurrency

- Threads
  - Create process by creating thread object
- Communication
  - shared variables
  - method calls, e.g., one object calls an enqueue method and another calls a dequeue method
- Mutual exclusion and synchronization
  - Not provided by communication through a shared object
  - Has a semaphore primitive (maintains a queue of waiting processes)
  - Supports monitors directly in the form of synchronized objects
    - *synchronized object*: objects that allow only one thread to invoke a method at a time

Synchronization Example

- Objects may have *synchronized* methods
- Can be used for mutual exclusion
  - Two threads may share an object.
  - If one calls a synchronized method, this locks object.
  - If the other calls a synchronized method on same object, this thread blocks until object is unlocked.



Synchronized Methods

- Marked by keyword

  public synchronized void commitTransaction(…) {…}

- Provides mutual exclusion
  - At most one synchronized method can be active
  - Unsynchronized methods can still be called
    - Programmer must be careful because this allows interaction through shared variables

Example

```
class LinkedCell {                  // Lisp-style cons cell containing
    protected double value;   // value and link to next cell
    protected LinkedCell next;
    public LinkedCell (double v, LinkedCell t) {
            value = v; next = t;
    }

    public synchronized double getValue() {
            return value;
    }

    public synchronized void setValue(double v) {
            value = v;          // assignment not atomic
    }

    public LinkedCell next() {   // no synchronization needed
            return next;
    }
```