Concurrent Programming

A Program

A Thread

Sequential Program

Concurrent Program

Two Threads
Multiprocessors

Shared-Memory Multiprocessor

Local cache has much lower latency
Threads and Processes
OS Space versus User Space

Threads and Processes
Tradeoffs

- **One-process-per-thread** is acceptable in personal computer with a single address space
  - This is too expensive in most OSes, since each operation on them requires a system call
  - Processes are general-purpose, so threads may pay the price of features they do not use
    - Processes are **heavy-weight**

- **All-threads-on-one-process** are acceptable in simple languages for uniprocessors
  - This precludes parallel execution in a multiprocessor machine
  - System call will block the entire set of threads
Threads and Processes
Multiprocessors

- A multiprocessor OS may allocate processes to processors following one of the following main strategies

- **Coscheduling** (a.k.a. *gang scheduling*): attempts to run each process in a different processor
  - Maximize parallelism

- **Space sharing** (a.k.a. *processor partitioning*): give an application exclusive use of some subset of the processors
  - Minimizing context switching cost and/or communication cost

Coroutines

- User-level threads are usually built on top of coroutines
  - Simulate parallel execution in a single processor
    - p(a,b,c)
    - d:=q(e,f); r(d,g,h)
    - s(i,j)

- **Coroutines** are execution contexts that exist concurrently and execute one at a time
- Coroutines transfer control to each other explicitly (by name)
Coroutines

Example

- Screen-saver program

```
loop
  -- update picture on screen
  -- perform next sanity check
```

- Successive updates and sanity checks usually depend on each other
  - Save and restore state of the computation
  - Coroutines are more attractive for this problem

Coroutines

Example

- A coroutine can detach itself from the main program
  - `detach` created the coroutine object

- Control can be transferred from one coroutine to another
  - `transfer` saves the program counter and resumes the coroutines specifies as a parameter
  - Transfers can occur anywhere in the code of the coroutine

```plaintext
us, cfs : coroutine
coroutine update_screen
  -- initialize
detach
  loop
    ...
    transfer (cfs)
    ...

coroutine check_file_system
  -- initialize
detach
  for all files
    ...
    transfer (us)
    ...
    transfer (us)
    ...
begin
  -- main
  us := new update_screen
cfs := new check_file_system
  resume (us)
```
Turning Coroutines Into Threads

- The programming language environment provides a scheduler in charge of transferring control automatically.
- The scheduler chooses which threads to run first after the current thread yields the processors.
- The scheduler may implement preemption mechanism that suspend the current thread on a regular basis:
  - Make processor allocation more fair.
- If the scheduler data structures are shared, threads can run in multiple processors.

Uniprocessor Scheduler

- `current_thread` and `ready_list` are used to manage threads.
- The scheduler manages the order of execution among threads.
- Waiting for conditions can also affect the scheduling process.
Scheduling

- **reschedule** is used to give up the processor
- **yield** is used to give up the processor temporarily
- Threads can wait on specific conditions
  - **sleep_on**
  - Intended for synchronization

```
procedure reschedule
  t : thread := dequeue (ready_list)
  transfer [t]

procedure yield
  enqueue (ready_list, current_thread)
  reschedule

procedure sleep_on (ref Q : queue of thread)
  enqueue (Q, current_thread)
  reschedule

procedure yield
  disable_signals
  enqueue (ready_list, current_thread)
  reschedule
  reenable_signals

disable_signals
if not desired_condition
  sleep_on (condition_queue)
  reenable_signals
```

Scheduling

- In preemptive multithreading, multiplexing does not require to explicitly invoke **yield**
- Switching is driven by signals
  - *Force the current thread to yield*
- **Race conditions** may occur inside yield if signals are not disabled

```
procedure reschedule
  t : thread := dequeue (ready_list)
  transfer [t]

procedure yield
  enqueue (ready_list, current_thread)
  reschedule

procedure sleep_on (ref Q : queue of thread)
  enqueue (Q, current_thread)
  reschedule

procedure yield
  disable_signals
  enqueue (ready_list, current_thread)
  reschedule
  reenable_signals

Preemptive Multithreading

disable_signals
if not desired_condition
  sleep_on (condition_queue)
  reenable_signals
```
Multiprocessor Scheduling

- The goal of most languages that support parallel threads is to that there should be no difference from the programmer point of view.
- This is an increasingly more important issue, since a very significant number of machines will be multiprocessors in the near future.
  - At least, Intel is trying hard to do this…
- Multiprocessor thread scheduling requires additional synchronization mechanism that prevent race conditions.

Concurrent Programming

- The two most crucial issues are:
  - Communication
  - Synchronization
- **Communication** refers to any mechanism that allows one thread to obtain information from another.
  - It is usually based on using *shared memory* or *message passing*.
- **Synchronization** refers to any mechanism that allows the programmer to control the relative order in which operations occur in different threads.
Shared Memory

- Example of synchronization: the semaphore
  - P decrements a counter, and waits till it is non-negative
  - V increments a counter, waking up waiting threads

```plaintext
shared buf : array [1..SIZE] of bdata
shared next_full, next_empty : integer := 1, 1
shared mutex : semaphore := 1
shared empty_slots, full_slots : semaphore := SIZE, 0

procedure insert (d : bdata)
   P (empty_slots)
   [mutex]
   buf[next_empty] := d
   next_empty := next_empty mod SIZE + 1
   V (mutex)
   V (full_slots)

function remove : bdata
   P (full_slots)
   P (mutex)
   d : bdata := buf[next_full]
   next_full := next_full mod SIZE + 1
   V (mutex)
   V (empty_slots)
   return d
```

Message Passing Models

- (a) processes name each other explicitly
- (b) senders name input ports
- (c) a channel abstraction
Ada Example

```ada
task buffer is
  entry insert (d : in bdata);
  entry remove (d : out bdata);
end buffer;

task body buffer is
  SIZE : constant integer := 10;
  subtype index is integer range 1..SIZE;
  buf : array (index) of bdata;
  next_empty, next_full : index := 1;
  full_slots : integer range 0..SIZE := 0;
begin
  loop
    select
      when full_slots < SIZE =>
        accept insert (d : in bdata) do
          buf(next_empty) := d;
          next_empty := next_empty mod SIZE + 1;
          full_slots := full_slots + 1;
        end;
      when full_slots > 0 =>
        accept remove (d : out bdata) do
          d := buf(next_full);
          next_full := next_full mod SIZE + 1;
          full_slots := full_slots - 1;
          end select;
    end loop;
end buffer;
```

Reading Assignment

- Read Scott
  - Sect. 12.1.3
  - Sect. 12.2.4
  - Sect. 8.6
  - Sect. 12.3 intro
  - Sect. 12.4 intro