Concurrency

In recent times ubiquity has not always been understood — not even by Sir Boyle Roche, for example, who held that a man cannot be in two places at once unless he is a bird.
-- Ambrose Bierce

- Concurrency 12.1, 12.2
- Communication 12.4, 12.5
  - semaphores
  - scheduling
  - message passing

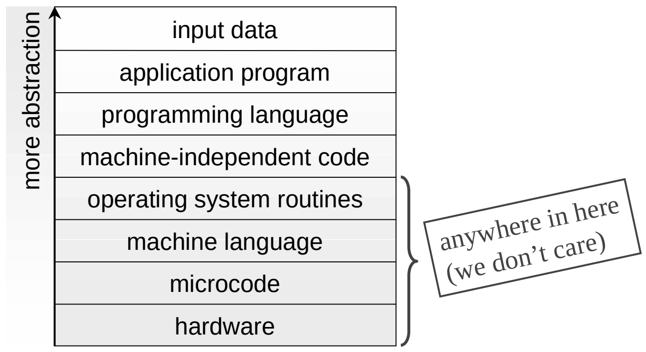
Concurrency

- *concurrency*
  - doing more than one thing at a time
- hardware concurrency ← *may not require special programming techniques*
  - multiple processors
  - distributed processing
- software concurrency ← *requires special programming techniques and languages*
  - quasi-parallel (coroutines)
  - simulated parallel (multiprogramming on a single processor)
  - parallel (multiprocessing on multiple processors)
  - massively parallel (multiprocessing on hardware with thousands of primitive processors)
  - distributed (on networks)

Hardware vs. Software Concurrency

We’re interested in the changes required to programming techniques and programming languages

- whether the concurrency is actually carried out by the hardware or simulated on a lower-level virtual machine is unimportant



Levels of Concurrency

Concurrency can take place on several levels:

- ~~instruction level~~
  - ~~simultaneous execution of machine level instructions~~
- statement level
  - simultaneous execution of high-level programming statements
- subprogram level
  - simultaneous execution of subprograms
- ~~program level~~
  - ~~simulataneous execution of programs~~

Tasks

- *task*
  - a sequence of operations independent of other such sequences
  - independent of = can be executed concurrently with
  - a.k.a. process
  - a.k.a. thread
- *disjoint tasks*
  - tasks that share no data and have no need to communicate

We're more concerned with tasks that *do* share data and *do* need to communicate.

414

Communication

Let this example motivate communication among tasks:

```
main program:
  Person Me;
  Me.married := false;
  loop concur:
    handle request(FromPerson, Me);
  rucnoc;
  ...
```

Me.married:

```
task_1(JuliaO, Me):
  get request type;
  if type = marriage then
    if not Me.married then
      write "I do.";
      Me.married := true;
    fi
  fi
end task;
```

```
task_2(KellyM, Me):
  get request type;
  if type = marriage then
    if not Me.married then
      write "I do.";
      Me.married := true;
    fi
  fi
end task;
```

415

Synchronization

Tasks that need to communicate must be synchronized:

- *synchronization*
  - control of the order in which statements within concurrent tasks are executed
- *cooperation*
  - task A needs the results of task B
  - task A waits for task B to finish, then takes B's results and continues execution
- *competition*
  - two concurrent tasks need to use shared data but neither task depends on the results of the other
  - who gets to use the data first? who has to wait?
    - first come first served
    - priority

Methods of Communicating

There are different ways for tasks to communicate:

- through shared data
  - two tasks have access to the same data
  - one task can "see" what the other is doing by looking at its modifications to the shared data area
- through mailboxes
  - each task has a public mailbox
  - other tasks can leave messages in the mailbox
  - a tasks checks its mailbox as often as it wants
- through message passing
  - a task sends a message directly to another task
  - the second task must respond

Accessing Shared Data

If two tasks share data, there must be a mechanism to prevent inconsistencies (as in our marriage example).

- *exclusive access*
  - only one task at a time is allowed to access shared data
- *critical region*
  - part of a task that cannot be interrupted
  - part of a task that needs exclusive access to shared data

Where is the critical region in our marriage tasks?

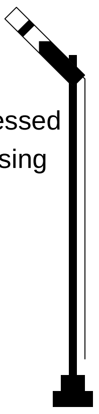
```
task_1(JuliaO, Me):
  get request type;
  if type = marriage then
    if not Me.married then
      write "I do.";
      Me.married := true;
    fi
  fi
end task;
```

Semaphores

- *semaphore*
  - a “do not disturb” sign

If a task wants to use some piece of data, it must check the semaphore first.

- if the semaphore is “up” (proceed)
  - the data is not being used by any other task and can be accessed
  - put the semaphore “down” to prevent other tasks from accessing the data
- if the semaphore is “down” (stop)
  - the data is in use by some other task
  - the task wanting to use the data must wait in a queue



Operations on Semaphores

The basic operations on semaphores are `wait` and `release`.

- wait
  - ▲
```
wait(Sem):
  if Sem = UP then
    Sem := DOWN
  else
    place task in Sem’s queue
```
- release
  - ▲
```
release(Sem):
  if Sem’s queue is empty then
    Sem := UP
  else
    remove task t from the head of Sem’s queue
    execute t
```

Semaphore Example

```
main program:
  Person Me;
  Me.married := false;
  loop concur:
    handle request(FromPerson, Me);
  rucnoc;
  ...
```

`Me.married:`

`MarSem:`

```
task_1(JuliaO, Me):
  get request type;
  if type = marriage then
    wait(MarSem);
    if not Me.married then
      write "I do.";
      Me.married := true;
      release(MarSem);
    else write "boohoo";
    fi
  fi
end task;
```

```
task_2(KellyM, Me):
  get request type;
  if type = marriage then
    wait(MarSem);
    if not Me.married then
      write "I do.";
      Me.married := true;
      release(MarSem);
    else write "boohoo";
    fi
  fi
end task;
```

Message Passing

The semaphore is one way to allow synchronization for tasks that share data in the same memory space. But sometimes, we have to synchronize tasks that do not share memory space.

- two tasks running concurrently: task A and task B
  - task A wishes to communicate with task B
  - task A waits until task B is willing to accept messages
  - task B declares that it is willing to accept messages
  - task A sends a message to task B

This allows each task to specify when it is willing to be interrupted (when it is not in a critical region).

422

Message Passing in Ada

In Ada, tasks are called “tasks”:

- tasks are activated (begin executing) automatically when the unit in which they are declared is activated
- tasks are executed concurrently until they need to communicate with each other, at which time they must be synchronized
- synchronization occurs between two tasks at a time

423

Ada Task Units

The basic form of the task unit in Ada is as follows:

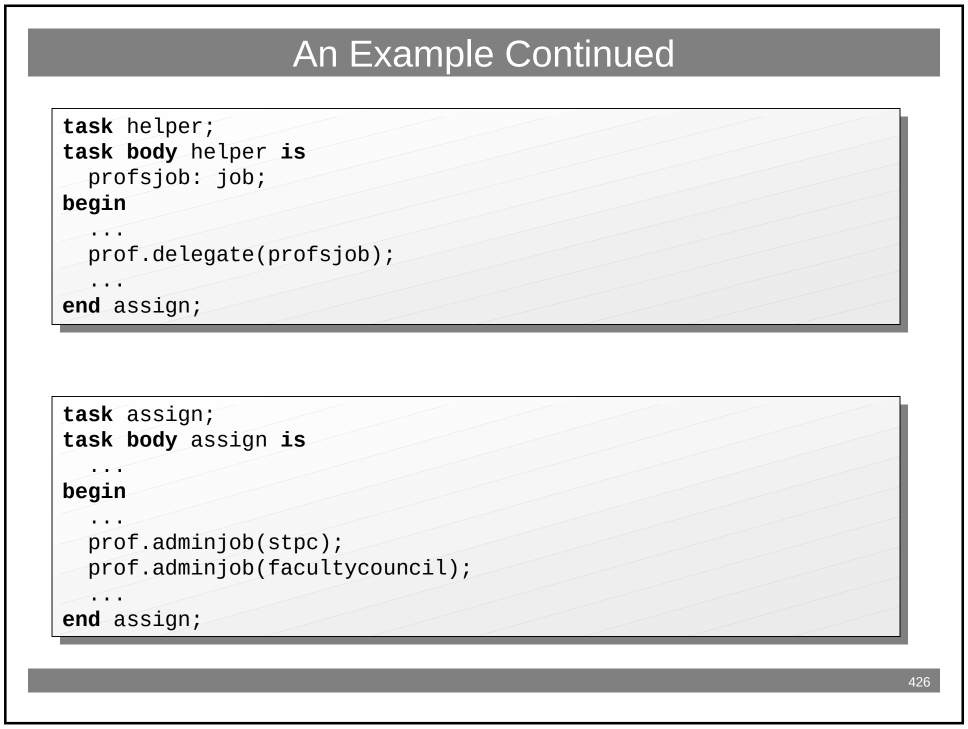
```
task T is
  entry E (formalparameters);
  ...
end T;
task body T is
  ...
  accept E (formalparameters) do
    ...
  end E;
  ...
end T;
```

And here's how to send a message to task T:

```
...
T.E(actualparameters);
...
```

An Example of Ada Tasks

```
task body prof is
  myadminjobs: array(1..10) of job;
  numjobs: integer := 0;   lastjob: integer := 0;
begin
  loop
    select
      when numjobs < 10 =>
        accept adminjob(J: in job) do
          myadminjobs(lastjob+1) := J;
        end adminjob;
        lastjob := lastjob + 1;  numjobs := numjobs + 1;
    or
      when numjobs > 0 =>
        accept delegate(J: out job) do
          J := myadminjobs(lastjob);
        end delegate;
        lastjob := lastjob - 1;  numjobs := numjobs - 1;
    end select;
  end loop;
end prof;
```



An Example Continued

```
task helper;
task body helper is
  profsjob: job;
begin
  ...
  prof.delegate(profsjob);
  ...
end assign;
```

```
task assign;
task body assign is
  ...
begin
  ...
  prof.adminjob(stpc);
  prof.adminjob(facultycouncil);
  ...
end assign;
```

426