# Chapter 12

*Concurrency can occur at four levels:*

1. Machine instruction level
2. High-level language statement level
3. Unit level
4. Program level

Because there are no language issues in instruction- and program-level concurrency, they are not addressed here

*The Evolution of Multiprocessor Architectures*

1. Late 1950s - One general-purpose processor and one or more special-purpose processors for input and output operations

2. Early 1960s - Multiple complete processors, used for program-level concurrency

3. Mid-1960s - Multiple partial processors, used for instruction-level concurrency

4. Single-Instruction Multiple-Data (SIMD) machines The same instruction goes to all processors, each with different data - e.g., *vector processors*

Copyright © 1998 by Addison Wesley Longman, Inc.

# Chapter 12

5. Multiple-Instruction Multiple-Data (MIMD) machines
   - Independent processors that can be synchronized (unit-level concurrency)

---

Def: A *thread of control* in a program is the sequence of program points reached as control flows through the program

## Categories of Concurrency:

1. *Physical* concurrency - Multiple independent processors
   ( multiple threads of control)

2. *Logical concurrency* - The appearance of physical concurrency is presented by time-sharing one processor
   (software can be designed as if there were multiple threads of control)

*- Coroutines provide only quasiconcurrency*

Copyright © 1998 by Addison Wesley Longman, Inc. 

# Chapter 12

## Reasons to Study Concurrency

1. It involves a new way of designing software that can be very useful--many real-world situations involve concurrency

2. Computers capable of physical concurrency are now widely used

## Fundamentals (for stmt-level concurrency)

Def: A *task* is a program unit that can be in concurrent execution with other program units

- Tasks differ from ordinary subprograms in that:

    1. A task may be implicitly started
    2. When a program unit starts the execution of a task, it is not necessarily suspended
    3. When a task s execution is completed, control may not return to the caller

Def: A task is *disjoint* if it does not communicate with or affect the execution of any other task in the program in any way

Copyright © 1998 by Addison Wesley Longman, Inc. 

# Chapter 12

**Task communication is necessary for synchronization**

**- *Task communication can be through:***

1. **Shared nonlocal variables**
2. **Parameters**
3. **Message passing**

**- *Kinds of synchronization:***

**1. *Cooperation***

- **Task A must wait for task B to complete some specific activity before task A can continue its execution**
  **e.g., the producer-consumer problem**

**2. *Competition***

- **When two or more tasks must use some resource that cannot be simultaneously used**
  **e.g., a shared counter**

  - **A problem because operations are not atomic**

Copyright © 1998 by Addison Wesley Longman, Inc.

# Chapter 12

- Competition is usually provided by *mutually exclusive access* (methods are discussed later

- Providing synchronization requires a mechanism for delaying task execution

- Task execution control is maintained by a program called the scheduler, which maps task execution onto available processors

- Tasks can be in one of several different execution states:

  1. New - created but not yet started

  2. Runnable or ready - ready to run but not currently running (no available processor)

  3. Running

  4. Blocked - has been running, but cannot now continue (usually waiting for some event to occur)

  5. Dead - no longer active in any sense

Copyright © 1998 by Addison Wesley Longman, Inc.

# Chapter 12

- *Liveness* is a characteristic that a program unit may or may not have

  - In sequential code, it means the unit will eventually complete its execution

  - In a concurrent environment, a task can easily lose its liveness

- If all tasks in a concurrent environment lose their liveness, it is called *deadlock*

- *Design Issues for Concurrency:*

  1. How is cooperation synchronization provided?

  2. How is competition synchronization provided?

  3. How and when do tasks begin and end execution?

  4. Are tasks statically or dynamically created?

Copyright © 1998 by Addison Wesley Longman, Inc. 

# Chapter 12

***Methods of Providing Synchronization:***

1. Semaphores
2. Monitors
3. Message Passing

## 1. Semaphores (Dijkstra - 1965)

- A *semaphore* is a data structure consisting of a counter and a queue for storing task descriptors

- Semaphores can be used to implement guards on the code that accesses shared data structures

- Semaphores have only two operations, wait and release (originally called P and V by Dijkstra)

- Semaphores can be used to provide both competition and cooperation synchronization

Copyright © 1998 by Addison Wesley Longman, Inc.

# Chapter 12

- *Cooperation Synchronization with Semaphores:*

  - *Example: A shared buffer*

    - **The buffer is implemented as an ADT with the operations `DEPOSIT` and `FETCH` as the only ways to access the buffer**

    - **Use two semaphores for cooperation:** `emptyspots` **and** `fullspots`

      - **The semaphore counters are used to store the numbers of empty spots and full spots in the buffer**

    - `DEPOSIT` **must first check** `emptyspots` **to see if there is room in the buffer**

      - **If there is room, the counter of** `emptyspots` **is decremented and the value is inserted**

      - **If there is no room, the caller is stored in the queue of** `emptyspots`

      - **When** `DEPOSIT` **is finished, it must increment the counter of** `fullspots`

Copyright © 1998 by Addison Wesley Longman, Inc. 

# Chapter 12

- FETCH **must first check** fullspots **to see if there is a value**

  - **If there is a full spot, the counter of** fullspots **is decremented and the value is removed**

  - **If there are no values in the buffer, the caller must be placed in the queue of** fullspots

  - **When** FETCH **is finished, it increments the counter of** emptyspots

- **The operations of** FETCH **and** DEPOSIT **on the semaphores are accomplished through two semaphore operations named wait and release**

**wait(**aSemaphore**)**
  if aSemaphore **s counter > 0** then
    **Decrement** aSemaphore **s counter**
  else
    **Put the caller in** aSemaphore **s queue**
    **Attempt to transfer control to some ready task**
    **(If the task ready queue is empty, deadlock occurs)**
  end

Copyright © 1998 by Addison Wesley Longman, Inc.

# Chapter 12

**release(**`aSemaphore`**)**
`if aSemaphore` **s queue is empty** `then`
**Increment** `aSemaphore` **s counter**
`else`
**Put the calling task in the task ready queue**
**Transfer control to a task from** `aSemaphore` **s queue**
`end`

`--->` **SHOW Program (p. 500)**

***- Competition Synchronization with Semaphores***

**- A third semaphore, named** `access`**, is used to control access (competition synchronization)**

**- The counter of** `access` **will only have the values 0 and 1**

**- Such a semphore is called a *binary semaphore***

`--->` **SHOW the complete shared buffer example program (p. 501-502)**

**- Note that wait and release must be atomic!**

Copyright © 1998 by Addison Wesley Longman, Inc.

# Chapter 12

***Evaluation of Semaphores:***

1. **Misuse of semaphores can cause failures in cooperation synchronization**
   **e.g., the buffer will overflow if the wait of `fullspots` is left out**

2. **Misuse of semaphores can cause failures in competition synchronization**
   **e.g., The program will deadlock if the release of `access` is left out**

## 2. Monitors (Concurrent Pascal, Modula, Mesa)

**The idea: encapsulate the shared data and its operations to restrict access**

**A *monitor* is an abstract data type for shared data**

**---> SHOW the diagram of monitor buffer operation, Figure 11.2 (p. 505)**

- ***Example language:* Concurrent Pascal**

  - **Concurrent Pascal is Pascal + classes, processes (tasks), monitors, and the `queue` data type (for semaphores)**

Copyright © 1998 by Addison Wesley Longman, Inc. 

# Chapter 12

- *Example language:* **Concurrent Pascal (continued)**

  - **Processes are types**

    - **Instances are statically created by declarations**

    - **An instance is started by `init`, which allocates its local data and begins its execution**

  - **Monitors are also types**
    **Form:**

```
type some_name = monitor (formal parameters)
  shared variables
  local procedures
  exported procedures (have entry in definition)
  initialization code
```

  - ***Competition Synchronization with Monitors:***

    - **Access to the shared data in the monitor is limited by the implementation to a single process at a time; therefore, mutually exclusive access is inherent in the semantic definition of the monitor**

    - **Multiple calls are queued**

Copyright © 1998 by Addison Wesley Longman, Inc. 

# Chapter 12

***- Cooperation Synchronization with Monitors:***

**- Cooperation is still required - done with semaphores, using the** `queue` **data type and the built-in operations,** `delay` **(similar to** `wait`**) and** `continue` **(similar to** `release`**)**

**-** `delay` **takes a queue type parameter; it puts the process that calls it in the specified queue and removes its exclusive access rights to the monitor s data structure**

**- Differs from** `send` **because** `delay` **always blocks the caller**

**-** `continue` **takes a queue type parameter; it disconnects the caller from the monitor, thus freeing the monitor for use by another process. It also takes a process from the parameter queue (if the queue isn t empty) and starts it**

**- Differs from** `release` **because it always has some effect (**`release` **does nothing if the queue is empty)**

**---> SHOW** `databuf` **monitor (p. 506),** `producer` **and** `consumer` **processes and the program that uses the buffer (p. 506-507)**

Copyright © 1998 by Addison Wesley Longman, Inc.

# Chapter 12

*Evaluation of monitors:*

- Support for competition synchronization is great!
- Support for cooperation synchronization is very similar as with semaphores, so it has the same problems

## 3. Message Passing

- Message passing is a general model for concurrency
  - It can model both semaphores and monitors
  - It is not just for competition synchronization

- *Central idea:* task communication is like seeing a doctor--most of the time he waits for you or you wait for him, but when you are both ready, you get together, or rendezvous

- In terms of tasks, we need:

  a. A mechanism to allow a task to indicate when it is willing to accept messages

  b. Tasks need a way to remember who is waiting to have its message accepted and some fair way of choosing the next message

Copyright © 1998 by Addison Wesley Longman, Inc.

# Chapter 12

***Def:*** **When a sender task s message is accepted by a receiver task, the actual message transmission is called a *rendezvous***

***- The Ada 83 Message-Passing Model***

**- Ada tasks have specification and body parts, like packages; the spec has the interface, which is the collection of entry points.**

**e.g.**
```
task EX is
    entry ENTRY_1 (STUFF : in FLOAT);
end EX;
```

**- The** `body` **task describes the action that takes place when a rendezvous occurs**

**- A task that sends a message is suspended while waiting for the message to be accepted *and* during the rendezvous**

**- Entry points in the spec are described with** `accept` **clauses (message sockets) in the body**

Copyright © 1998 by Addison Wesley Longman, Inc. 

# Chapter 12

***- Example of a task body:***

```
task body EX is
  begin
  loop
  accept ENTRY_1 (ITEM: in FLOAT) do
  ...
  end;
  end loop;
end EX;
```

***- Semantics:***

**a. The task executes to the top of the** `accept` **clause and waits for a message**

**b. During execution of the** `accept` **clause, the sender is suspended**

**c.** `accept` **parameters can transmit information in either or both directions**

**d. Every** `accept` **clause has an associated queue to store waiting messages**

---> **SHOW rendezvous time lines for the example task (Figure 12.3, p. 511)**

**Copyright © 1998 by Addison Wesley Longman, Inc.** 

# Chapter 12

- A task that has `accept` clauses, but no other code is called a ***server task*** (the example above is a server task)

- A task without accept clauses is called an ***actor task***

- Example actor task:

```
task WATER_MONITOR; -- specification
task body WATER_MONITOR is -- body
  begin
    loop
    if WATER_LEVEL > MAX_LEVEL
      then SOUND_ALARM;
    end if;
    delay 1.0; -- No further execution
               -- for at least 1 second
    end loop;
  end WATER_MONITOR;
```

- An actor task can send messages to other tasks

- Note: A sender must know the `entry` name of the receiver, but not vice versa

- Tasks can be either statically or dynamically allocated

Copyright © 1998 by Addison Wesley Longman, Inc. 

# Chapter 12

- ***Example:***

```
task type TASK_TYPE_1 is ... end;
type TASK_PTR is access TASK_TYPE_1;
TASK1 : TASK_TYPE_1;          -- stack dynamic
TASK_PTR := new TASK_TYPE_1;  -- heap dynamic
```

- **Tasks can have more than one** `entry` **point**

  - **The specification task has an** `entry` **clause for each**

  - **The task body has an** `accept` **clause for each** `entry` **clause, placed in a** `select` ***clause*, which is in a loop**

- ***Example task with multiple entries:***

```
task body TASK_EXAMPLE is
  loop
    select
      accept ENTRY_1 (formal params) do
      ...
      end ENTRY_1;
      ...
    or
      accept ENTRY_2 (formal params) do
      ...
      end ENTRY_2;
      ...
    end select;
  end loop;
end TASK_EXAMPLE;
```

Copyright © 1998 by Addison Wesley Longman, Inc.

# Chapter 12

- ***Semantics of tasks with*** `select` ***clauses:***

  - **If exactly one** `entry` **queue is nonempty, choose a message from it**

  - **If more than one** `entry` **queue is nonempty, choose one, nondeterministically, from which to accept a message**

  - **If all are empty, wait**

- **Extended** `accept` **clause - code following the clause, but before the next clause**
  - **Executed concurrently with the caller**

- ***Cooperation Synchronization with Message Passing***

  - **Provided by *Guarded*** `accept` ***clauses***

    - ***Example:***

```
when not FULL(BUFFER) =>
  accept DEPOSIT (NEW_VALUE) do
  ...
  end DEPOSIT;
```

**Copyright © 1998 by Addison Wesley Longman, Inc.**

# Chapter 12

**Def: A clause whose guard is true is called *open*.**

**Def: A clause whose guard is false is called *closed*.**

**Def: A clause without a guard is always open.**

***- Semantics of select with guarded*** `accept` ***clauses:***

`select` **first checks the guards on all clauses**

**If exactly one is open, its queue is checked for messages**

**If more than one are open, nondeterministically choose a queue among them to check for messages**

**If all are closed, it is a runtime error**

- **A** `select` **clause can include an** `else` **clause to avoid the error**
  - **When the** `else` **clause completes, the loop repeats**

Copyright © 1998 by Addison Wesley Longman, Inc. 

# Chapter 12

***Example of a task with guarded*** `accept` ***clauses:***

```
task GAS_STATION_ATTENDANT is
  entry SERVICE_ISLAND (CAR : CAR_TYPE);
  entry GARAGE (CAR : CAR_TYPE);
end GAS_STATION_ATTENDANT;

task body GAS_STATION_ATTENDANT is
  begin
    loop
      select
        when GAS_AVAILABLE =>
           accept SERVICE_ISLAND (
                       CAR : CAR_TYPE) do
             FILL_WITH_GAS (CAR);
           end SERVICE_ISLAND;
      or
         when GARAGE_AVAILABLE =>
           accept GARAGE (
                       CAR : CAR_TYPE) do
             FIX (CAR);
           end GARAGE;
      else
          SLEEP;
      end select;
    end loop;
  end GAS_STATION_ATTENDANT;
```

Copyright © 1998 by Addison Wesley Longman, Inc. 

# Chapter 12

- *Competition Synchronization with Message Passing:*

  - *Example--a shared buffer*

  - **Encapsulate the buffer and its operations in a task**

  - **Competition synchronization is implicit in the semantics of** `accept` **clauses**
    - **Only one** `accept` **clause in a task can be active at any given time**

**---> SHOW** `BUF_TASK` **task and the** `PRODUCER` **and** `CONSUMER` **tasks that use it (p. 514-515)**

## Task Termination

**Def: The execution of a task is *completed* if control has reached the end of its code body**

- **If a task has created no dependent tasks and is completed, it is terminated**

- **If a task has created dependent tasks and is completed, it is not terminated until all its dependent tasks are terminated**

Copyright © 1998 by Addison Wesley Longman, Inc.

# Chapter 12

- A `terminate` clause in a `select` is just a `terminate` statement

- A `terminate` clause is selected when no `accept` clause is open

- When a `terminate` is selected in a task, the task is terminated only when its master and all of the dependents of its master are either completed or are waiting at a `terminate`

- A block or subprogram is not left until all of its dependent tasks are terminated

- *Priorities*

  - The priority of any task can be set with the the pragma `priority`

  - The priority of a task applies to it only when it is in the task ready queue

- *Evaluation of the Ada 83 Tasking Model*

- If there are no distributed processors with independent memories, monitors and message passing are equally suitable. Otherwise, message passing is clearly superior

Copyright © 1998 by Addison Wesley Longman, Inc. 

# Chapter 12

## Concurrency in Ada 95

- Ada 95 includes Ada 83 features for concurrency, plus two new features

1. *Protected Objects*
    - A more efficient way of implementing shared data
    - The idea is to allow access to a shared data structure to be done without rendezvous

- A protected object is similar to an abstract data type

- Access to a protected object is either through messages passed to entries, as with a task, or through protected subprograms

- A *protected procedure* provides mutually exclusive read-write access to protected objects

- A *protected function* provides concurrent read-only access to protected objects

---> SHOW the protected buffer code (pp. 518-519)

Copyright © 1998 by Addison Wesley Longman, Inc. 

# Chapter 12

2. ***Asynchronous Communication***

   **- Provided through asynchronous** `select` **structures**

   **- An asynchronous select has two triggering alternatives, and entry clause or a delay**

   **- The entry clause is triggered when sent a message; the delay clause is triggered when its time limit is reached**

`--->` **SHOW examples (p. 519-520)**

**Copyright © 1998 by Addison Wesley Longman, Inc.**

# Chapter 12

## Java Threads

- The concurrent units in Java are `run` methods

- The `run` method is inherited and overriden in subclasses of the predefined `Thread` class

- *The Thread Class*

  - Includes several methods (besides `run`)

    - `start`, which calls `run` , after which control returns immediately to `start`

    - `yield`, which stops execution of the thread and puts it in the task ready queue

    - `sleep`, which stops execution of the thread and blocks it from execution for the amount of time specified in its parameter

    - `suspend`, which stops execution of the thread until it is restarted with `resume`

    - `resume`, which restarts a suspended thread

    - `stop`, which kills the thread

Copyright © 1998 by Addison Wesley Longman, Inc.

# Chapter 12

*- Competition Synchronization with Java Threads*

- A method that includes the `synchronized` modifier disallows any other method from running on the object while it is in execution

- If only a part of a method must be run without interference, it can be synchronized

*- Cooperation Synchronization with Java Threads*

- The `wait` and `notify` methods are defined in `Object`, which is the root class in Java, so all objects inherit them

- The wait method must be called in a loop

*- Example - the queue*

---> SHOW `Queue` class (p. 524) and the `Producer` and `Consumer` classes (p. 525)

Copyright © 1998 by Addison Wesley Longman, Inc. 