Chapter 13 Topics

• Introduction
• Introduction to Subprogram-Level Concurrency
• Semaphores
• Monitors
• Message Passing
• Java Threads
• C# Threads
• Statement-Level Concurrency

Introduction

• 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

Introduction

• 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
  5. Multiple-Instruction Multiple-Data (MIMD) machines

  Independent processors that can be synchronized (unit-level concurrency)
Introduction

- **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 quasi-concurrency.

Introduction to Subprogram-Level Concurrency

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

- Tasks differ from ordinary subprograms in that:
  - A task may be implicitly started.
  - When a program starts the execution of a task, it is not necessarily suspended.
  - When a task's execution is completed, control may not return to the caller.

- Tasks usually work together.

- Two general categories of tasks:
  - **Heavyweight tasks:** Execute in their own address space and have their own run-time stacks.
  - **Lightweight tasks:** All run in the same address space and use the same run-time stack.

- **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.

- Task communication is necessary for synchronization:
  - Task communication can be through:
    1. Shared nonlocal variables;
    2. Parameters;

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.
   - Competition is usually provided by mutually exclusive access (approaches are discussed later).

<table>
<thead>
<tr>
<th>Task A</th>
<th>Task B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1</td>
<td>Task 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task A</th>
<th>Task B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 3</td>
<td>Task 4</td>
</tr>
</tbody>
</table>
Introduction to Subprogram-Level Concurrency

- 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

Introduction to Subprogram-Level Concurrency

- 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

Introduction to Subprogram-Level Concurrency

- 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?
- Methods of Providing Synchronization:
  1. Semaphores
  2. Monitors
  3. Message Passing
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

Semaphores

- Cooperation Synchronization 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 first checks 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
  - 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

Semaphores

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

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

Producer Consumer Code

```cpp
semaphore fullspots, emptyspots;
fullspots.count = 0;
emptyspots.count = BUFLEN;
task producer;
loop
    -- produce VALUE --
    wait(emptyspots);  {wait for space}
    DEPOSIT(VALUE);
    release(fullspots); {increase filled}
end loop;
end producer;

task consumer;
loop
    wait(fullspots); {wait till not empty} // FETCH(VALUE)
    release(emptyspots); {increase empty}
    -- consume VALUE --
end loop;
end consumer;
```

Semaphores

- 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 semaphore is called a binary semaphore
  - Note that `wait` and `release` must be atomic!
Semaphores

• 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

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

Monitors

• Example language: Concurrent Pascal
  – Concurrent Pascal is Pascal + classes, processes (tasks), monitors, and the queue data type (for semaphores)
  – Processes are types
    • Instances are statically created by declarations (the declaration does not start its execution)
    • An instance is “started” by init, which allocates its local data and begins its execution
Monitors

- Monitors are also types
- Form:
  
  ```
type some_name = monitor (formal parameters)
  shared variables
  local procedures
  exported procedures (have entry in definition)
  initialization code
  ```

Monitors

- 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
- 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`)

Monitors

- Cooperation Synchronization with Monitors:
  - `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)
Monitors

- 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

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 (don’t let tasks interrupt each other)

Message Passing

- 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
- Def: When a sender task’s message is accepted by a receiver task, the actual message transmission is called a rendezvous
Message Passing

- 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 in the body

- Example of a task body:

```
task body TASK EXAMPLE is
  begin
    loop
      accept ENTRY_1 (ITEM: in FLOAT)
      do
        ...
      end ENTRY_1;
    end loop;
    end TASK EXAMPLE;
```
Message Passing

- 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

Rendezvous Time Lines

Message Passing

- 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
A Rendezvous

Message Passing

- 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;
  ```

Message Passing

- An actor task can send messages to other tasks
- Note: A sender must know the `entry` name of the receiver, but not vice versa (asymmetric)
Message Passing

• Tasks can be either statically or dynamically allocated
• Example:

```plaintext
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
```

Message Passing

• 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

```plaintext
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;
```

Message Passing

• Example task with multiple entries:

```plaintext
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;
```
Message Passing

• 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
  – The construct is often called a selective wait
• Extended `accept` clause - code following the clause, but before the next clause
  – Executed concurrently with the caller

Message Passing

• Cooperation Synchronization with Message Passing
  – Provided by Guarded `accept` clauses
  – Example:
    ```
    when not FULL(BUFFER) =>
      accept DEPOSIT (NEW_VALUE) do ...
      end DEPOSIT;
    ```

Message Passing

• 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`
Message Passing

• 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

Message Passing

• Example of a task with guarded accept clauses:
  • Note: The station may be out of gas and there
    may or may not be a position available in the
    garage

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

Message Passing

```plaintext
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;
```
Message Passing

• 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

Message Passing

• 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

Message Passing

• 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
Message Passing

• Priorities
  – The priority of any task can be set with 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

Concurrency in Ada 95

• Ada 95: Ada 83 concurrency + 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
  2. Asynchronous Communication
     • Provided through asynchronous select structures
     • An asynchronous select has two triggering alternatives, an 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

Java Threads

• The Thread class
  – Methods: run and start
  – yield, sleep, join, interrupt
  – getPriority/setPriority
• Declare as subclass of Thread
• runnable interface
Java Threads

- Competition Synchronization with Java Threads
  - A method that includes the \texttt{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 \texttt{wait}, \texttt{notify} and \texttt{notifyAll} methods are defined in \texttt{Object}, which is the root class in Java, so all objects inherit them
  - The \texttt{wait} method must be called in a loop

Statement-Level Concurrency

- High-Performance FORTRAN (HPF)
  - Extensions that allow the programmer to provide information to the compiler to help it optimize code for multiprocessor computers
  - Primary HPF specifications:
    1. Number of processors
       \texttt{!HPF$ PROCESSORS procs(n)}
    2. Distribution of data
       \texttt{!HPF$ DISTRIBUTE (kind)}
       \texttt{ONTO procs :: identifier_list}
       - \texttt{kind} can be \texttt{BLOCK} (distribute data to processors in blocks) or \texttt{CYCLIC} (distribute data to processors one element at a time)
    3. Relate the distribution of one array with that of another
       \texttt{ALIGN array1\_element WITH array2\_element}

- Code example
  \begin{verbatim}
  REAL list_1(1000), list_2(1000)
  INTEGER list_3(500), list_4(501)
  !HPF$ PROCESSORS proc (10)
  !HPF$ DISTRIBUTE (BLOCK) ONTO procs ::
    list_1, list_2
  !HPF$ ALIGN list_1(index) WITH
      list_4 (index+1)
  …
    list_1 (index) = list_2(index)
    list_3(index) = list_4(index+1)
  FORALL (index = 1:1000)
    list_1(index) = list_2(index)
  \end{verbatim}
  - Specifies that all 1000 RHSs of the assignments can be evaluated before any assignment takes place