Topics

- Subprogram-Level Concurrency
- Types of Synchronization
  - *Cooperation* synchronization (or simply synchronization)
  - *Competition* synchronization (aka mutual exclusion)
- Support for Synchronization
  - Semaphores
  - Monitors
  - Message Passing

Introduction

- Concurrency can occur at four levels:
  - Machine instruction level
  - High-level language statement level
  - Unit/subprogram level
  - Program level
- Because there are no language issues in instruction- and program-level concurrency, they are not addressed here

Categories of Concurrency

- Categories of Concurrency:
  - *Physical concurrency* – Multiple independent processors (multiple threads of control)
  - *Logical concurrency* – The appearance of physical concurrency is presented by time-sharing one processor
- A *thread of control* in a program is the sequence of program points reached as control flows through the program
- A program designed to have more than one thread of control is said to be *multi-threaded*

Motivations for the Use of Concurrency

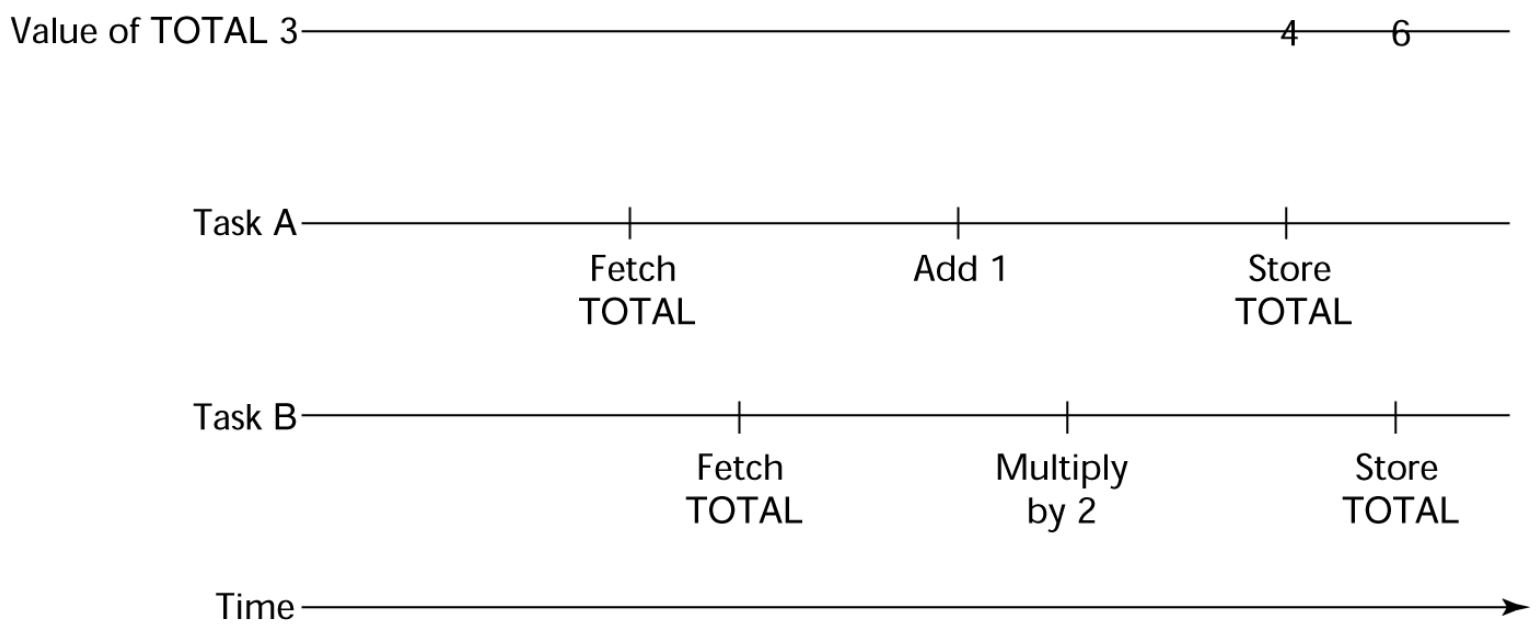
- Multiprocessor computers capable of physical concurrency are now widely used
- Even if a machine has just one processor, a program written to use concurrent execution can be faster than the same program written for nonconcurrent execution
- Many real-world situations involve concurrency - a different way of designing software can be very useful
- Many program applications are now spread over multiple machines, either locally or over a network

Task Synchronization

- *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
- *Competition*: 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)

Need for Competition Synchronization

Task A: `TOTAL = TOTAL + 1`
Task B: `TOTAL = 2 * TOTAL`

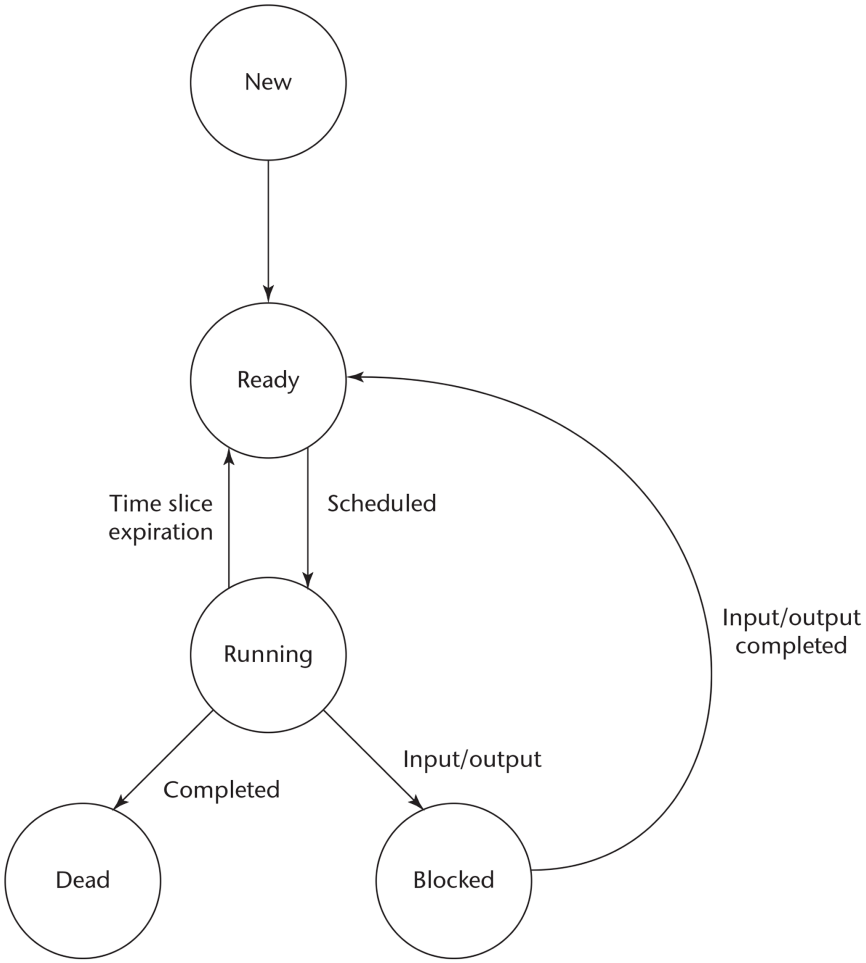


- Depending on order, there could be four different results

Task Execution States

- *New* – created but not yet started
- *R*eady – ready to run but not currently running (no available processor)
- *Running*
- *Blocked* – has been running, but cannot now continue (usually waiting for some event to occur)
- *Dead* – no longer active in any sense

Flow diagram of task states



Liveness and Deadlock

- *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*

Methods of Providing Synchronization

- Semaphores
- Monitors
- Message Passing

Semaphores

- Dijkstra – 1965
- A *semaphore* is a data structure consisting of a counter and a queue for storing task descriptors
  - A task descriptor is a data structure that stores all of the relevant information about the execution state of the task
- 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

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

Cooperation Synchronization with Semaphores (continued)

- `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`

Cooperation Synchronization with Semaphores (continued)

- `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: Wait and Release Operations

```
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 a ready task
end

release(aSemaphore)
if aSemaphore's queue is empty then
   increment aSemaphore's counter
else
   move one task from aSemaphore's queue to the ready
  queue
end
```

Producer and Consumer Tasks

```
semaphore fullspots, emptyspots;
fullstops.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;
```

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!

Producer Code for Semaphores

```
semaphore access, fullspots, emptyspots;
access.count = 1;
fullstops.count = 0;
emptyspots.count = BUFLEN;
task producer;
   loop
      -- produce VALUE --
      wait(emptyspots); {wait for space}
      wait(access);     {wait for access)
      DEPOSIT(VALUE);
      release(access); {relinquish access}
      release(fullspots); {increase filled}
   end loop;
end producer;
```

Consumer Code for Semaphores

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

Evaluation of Semaphores

- Misuse of semaphores can cause failures in cooperation synchronization, e.g., the buffer will overflow if the wait of `fullspots` is left out
- Misuse of semaphores can cause failures in competition synchronization, e.g., the program will deadlock if the release of `access` is left out

Monitors

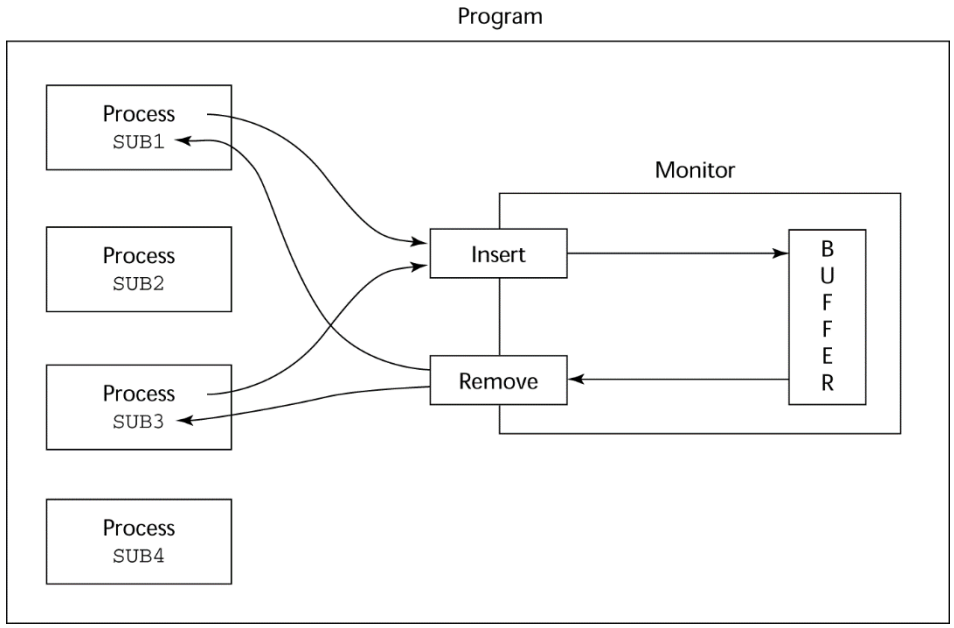
- Ada, Java, C#
- The idea: encapsulate the shared data and its operations to restrict access
- A monitor is an abstract data type for shared data

Competition Synchronization

- Shared data is resident in the monitor (rather than in the client units)
- All access resident in the monitor
  - Monitor implementation guarantee synchronized access by allowing only one access at a time
  - Calls to monitor procedures are implicitly queued if the monitor is busy at the time of the call

Cooperation Synchronization

- Cooperation between processes is still a programming task
  - Programmer must guarantee that a shared buffer does not experience underflow or overflow



Evaluation of Monitors

- A better way to provide competition synchronization than are semaphores
- Semaphores can be used to implement monitors
- Monitors can be used to implement semaphores
- 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 she waits for you or you wait for her, but when you are both ready, you get together, or *rendezvous*

Message Passing Rendezvous

- To support concurrent tasks with message passing, a language needs:

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

  - A way to remember who is waiting to have its message accepted and some "fair" way of choosing the next message

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

Assignments

- Assignment #7, due ~~April 12~~ April 16
- Assignment #8, due April 16
- It might be a good idea to complete Assignment #8 first