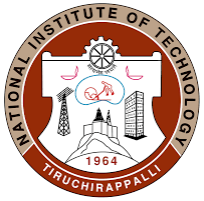


CSPC31: Principles of Programming Languages

Dr. R. Bala Krishnan
Asst. Prof.
Dept. of CSE
NIT, Trichy – 620 015
Ph: 999 470 4853
E-Mail: balakrishnan@nitt.edu

Books

- **Text Books**
  - ✓ **Robert W. Sebesta, *"Concepts of Programming Languages"*, Tenth Edition, Addison Wesley, 2012.**
  - ✓ Michael L. Scott, *"Programming Language Pragmatics"*, Third Edition, Morgan Kaufmann, 2009.
- **Reference Books**
  - ✓ Allen B Tucker, and Robert E Noonan, *"Programming Languages – Principles and Paradigms"*, Second Edition, Tata McGraw Hill, 2007.
  - ✓ R. Kent Dybvig, *"The Scheme Programming Language"*, Fourth Edition, MIT Press, 2009.
  - ✓ Jeffrey D. Ullman, *"Elements of ML Programming"*, Second Edition, Prentice Hall, 1998.
  - ✓ Richard A. O'Keefe, *"The Craft of Prolog"*, MIT Press, 2009.
  - ✓ W. F. Clocksin, C. S. Mellish, *"Programming in Prolog: Using the ISO Standard"*, Fifth Edition, Springer, 2003.

Chapters

Chapter No.	Title
1.	Preliminaries
~~2.~~	~~Evolution of the Major Programming Languages~~
3.	Describing Syntax and Semantics
4.	Lexical and Syntax Analysis
~~5.~~	~~Names, Binding, Type Checking and Scopes~~
6.	Data Types
7.	Expressions and Assignment Statements
8.	Statement-Level Control Structures
9.	Subprograms
10.	Implementing Subprograms
~~11.~~	~~Abstract Data Types and Encapsulation Constructs~~
12.	Support for Object-Oriented Programming
13.	Concurrency
14.	Exception Handling and Event Handling
15.	Functional Programming Languages
16.	Logic Programming Languages

Chapter 13 – Concurrency

Introduction to Subprogram-Level Concurrency

- A task is a unit of a program, similar to a subprogram, that can be in concurrent execution with other units of the same program
- Each task in a program can support one thread of control
- ~~Tasks are sometimes called processes~~
- ~~In some languages, for example Java and C#, certain methods serve as tasks. Such methods are executed in objects called threads~~
- Tasks fall into two general categories: heavyweight and lightweight
  - A heavyweight task executes in its own address space
  - Lightweight tasks all run in the same address space
- It is easier to implement lightweight tasks than heavyweight tasks
- Furthermore, lightweight tasks can be more efficient than heavyweight tasks, because less effort is required to manage their execution
- A task can communicate with other tasks through shared nonlocal variables, through message passing, or through parameters
- If a task does not communicate with or affect the execution of any other task in the program in any way, it is said to be disjoint
- Because tasks often work together to create simulations or solve problems and therefore are not disjoint, they must use some form of communication to either synchronize their executions or share data or both

Introduction to Subprogram-Level Concurrency

- Synchronization is a mechanism that controls the order in which tasks execute
- Two kinds of synchronization are required when tasks share data: cooperation and competition
- Cooperation synchronization is required between task A and task B when task A must wait for task B to complete some specific activity before task A can begin or continue its execution -> **Producer-Consumer Problem**
- Competition synchronization is required between two tasks when both require the use of some resource that cannot be simultaneously used -> **Bank Transaction**

Introduction to Subprogram-Level Concurrency

- Cooperation Synchronization (Producer-Consumer Problem)
  - The consumer unit must not be allowed to take data from the buffer if the buffer is empty
  - Likewise, the producer unit cannot be allowed to place new data in the buffer if the buffer is full
  - This is a problem of cooperation synchronization because the users of the shared data structure must cooperate if the buffer is to be used correctly
- Competition synchronization (Bank Transaction)
  - Prevents two tasks from accessing a shared data structure at exactly the same time—a situation that could destroy the integrity of that shared data
  - To provide competition synchronization, mutually exclusive access to the shared data must be guaranteed

Cooperation Synchronization

- This problem originated in the development of operating systems, in which one program unit produces some data value or resource and another uses it
- Produced data are usually placed in a storage buffer by the producing unit and removed from that buffer by the consuming unit
- The sequence of stores to and removals from the buffer must be synchronized
- The consumer unit must not be allowed to take data from the buffer if the buffer is empty
- Likewise, the producer unit cannot be allowed to place new data in the buffer if the buffer is full
- This is a problem of cooperation synchronization because the users of the shared data structure must cooperate if the buffer is to be used correctly

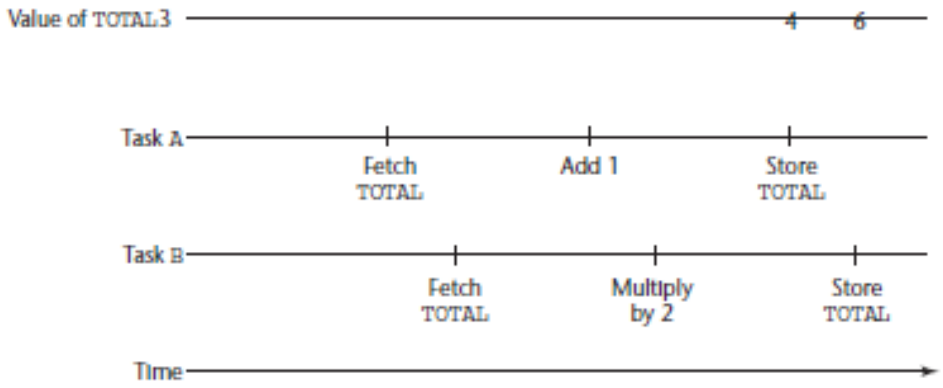
Competition Synchronization

To clarify the competition problem, consider the following scenario: Suppose task A has the statement `TOTAL += 1`, where TOTAL is a shared integer variable. Furthermore, suppose task B has the statement `TOTAL *= 2`. Task A and task B could try to change TOTAL at the same time.

At the machine language level, each task may accomplish its operation on TOTAL with the following three-step process:

1. Fetch the value of TOTAL.
2. Perform the arithmetic operation.
3. Put the new value back in TOTAL.

Without competition synchronization, given the previously described operations performed by tasks A and B on TOTAL, four different values could result, depending on the order of the steps of the operation. Assume TOTAL has the value 3 before either A or B attempts to modify it. If task A completes its operation before task B begins, the value will be 8, which is assumed here to be correct. But if both A and B fetch the value of TOTAL before either task puts its new value back, the result will be incorrect. If A puts its value back first, the value of TOTAL will be 6. This case is shown in Figure 13.1. If B puts its value back first, the value of TOTAL will be 4. Finally, if B completes its operation before task A begins, the value will be 7. A situation that leads to these problems is sometimes called a **race condition**, because two or more tasks are racing to use the shared resource and the behavior of the program depends on which task arrives first (and wins the race). The importance of competition synchronization should now be clear.



Solution

- Three methods of providing for mutually exclusive access to a shared resource are semaphores, monitors, and message passing
- Semaphores:
- Use a variable called Semaphore variable, X

P(s)
Critical Section
V(s)

Wait(P)↓; Signal(V) ↑

- The wait semaphore subprogram is used to test the counter of a given semaphore variable
  - If the value is greater than zero, the caller can carry out its operation. In this case, the counter value of the semaphore variable is decremented to indicate that there is now one fewer of whatever it counts
  - If the value of the counter is zero, the caller must be placed on the waiting queue of the semaphore variable, and the processor must be given to some other ready task
- The release semaphore subprogram is used by a task to allow some other task to have one of whatever the counter of the specified semaphore variable counts
  - If the queue of the specified semaphore variable is empty, which means no task is waiting, release increments its counter (to indicate there is one more of whatever is being controlled that is now available)
  - If one or more tasks are waiting, release moves one of them from the semaphore queue to the ready queue

Solution

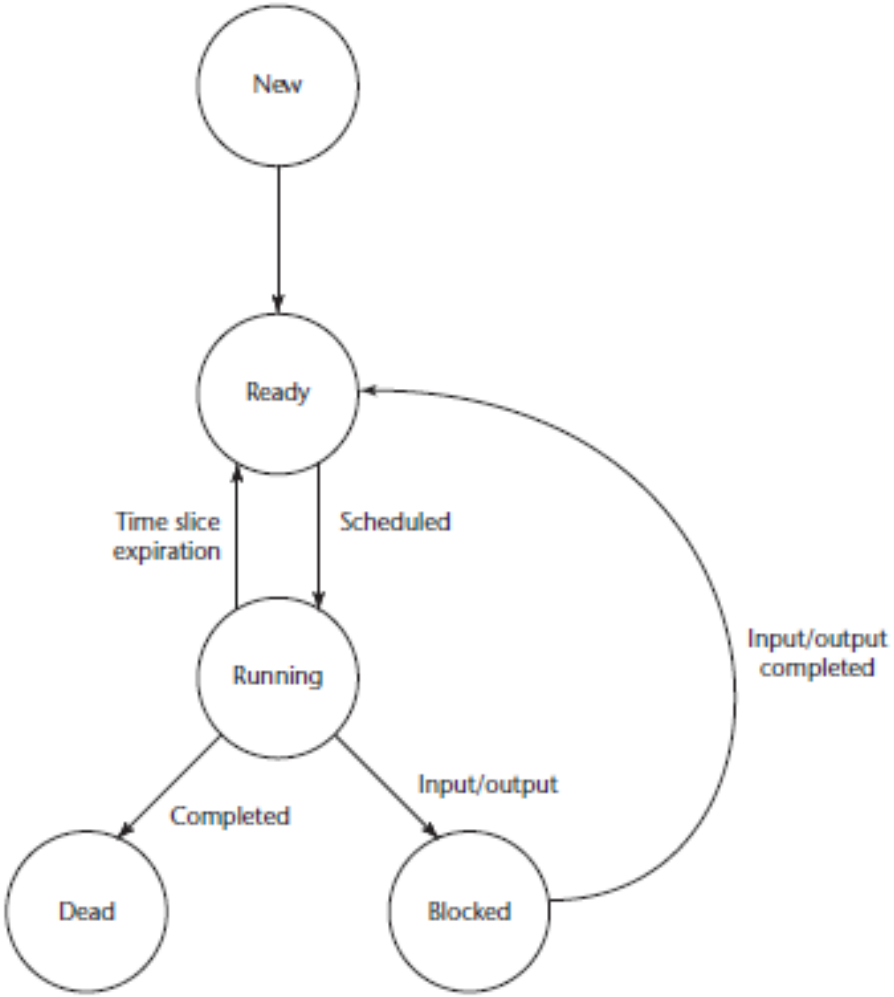
- Semaphore Types
  - Binary Semaphore (Solution for Competition synchronization – Provides mutually exclusive access to shared resource)
    - If the value of the semaphore variable is 0, then wait
    - Else
      - Before entering the critical section, make the semaphore value as 0
      - Execute the code in critical section
      - After executing the code in critical section, make the semaphore value as 1 and then exit from critical section
  - Counting Semaphore (Solution for Competition Semaphore)
    - Check the value of the semaphore variable value
      - **Producer:** If the value has not reached the maximum count
        - ❑ Increment the semaphore variable value by 1
        - ❑ Manufacture a product
      - Else, wait and repeat the checking after sometime
      - **Consumer:** If the value is not 0
        - ❑ Decrement the semaphore variable value by 1
        - ❑ Consume the product
      - Else, wait and repeat the checking after sometime

Various States of Task

- Tasks can be in several different states:

1. **New:** A task is in the new state when it has been created but has not yet begun its execution
2. **Ready:** A ready task is ready to run but is not currently running. Either it has not been given processor time by the scheduler, or it had run previously but was blocked. Tasks that are ready to run are stored in a queue that is often called the task ready queue
3. **Running:** A running task is one that is currently executing; that is, it has a processor and its code is being executed
4. **Blocked:** A task that is blocked has been running, but that execution was interrupted by one of several different events, the most common of which is an input or output operation. In addition to input and output, some languages provide operations for the user program to specify that a task is to be blocked
5. **Dead:** A dead task is no longer active in any sense. A task dies when its execution is completed or it is explicitly killed by the program

Various States of Task



Thank You