uOttawa

L'Université canadienne
Canada's university

Concurrency

Contents

- Concept of concurrency
- Subprogram-level concurrency
- Semaphores
- Monitors
- Message passing
- Java thread

Concept of Concurrency

- Concurrency: mean working in parallel
- It is divided into instruction level, statement level, unit level, and program level.
- Concurrent execution of program units can be either physically on separate processors or logically in some time-sliced fashion on a single processor computer system.

Why Study Concurrency?

- It provides a method of conceptualizing program solutions to problems.
- Multiple processor computers are now being widely used, and creating the need for software to make effective use of that hardware capability.

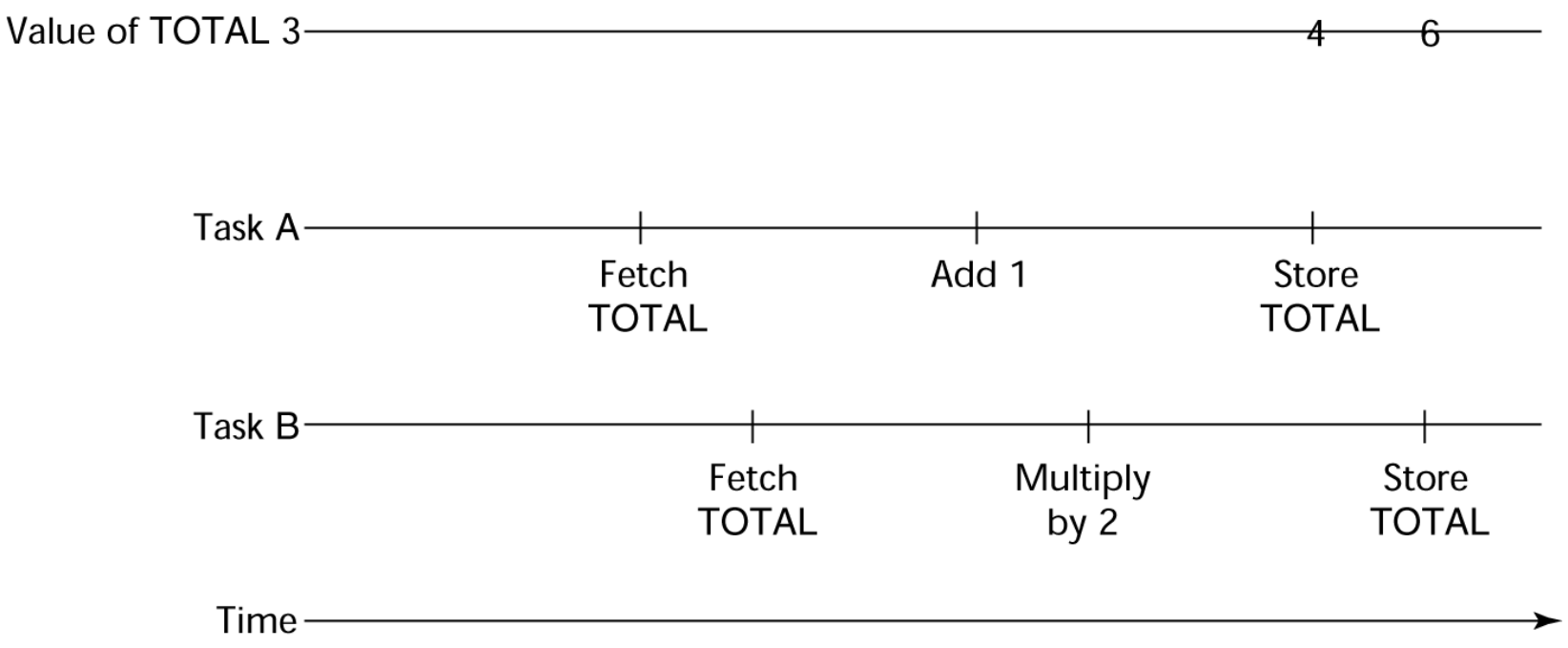
Subprogram-Level Concurrency

- A task is a unit of program that can be in concurrent execution with other units of the same program.
- Each task in a program can provide one thread of control.
- A task can communicate with other tasks through shared nonlocal variables[global], through message passing, or through parameters.

Synchronization

- A mechanism to control the order in which tasks execute.
- 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 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. e.g., a shared counter

The Need for Competition Synchronization



Critical Section

- A segment of code, in which the thread may be changing common variables, updating a table, writing a file, and so on.
- The execution of critical sections by the threads is mutually exclusive in time.

Task States

- New: it has been created, but has not yet begun its execution.
- Runnable or ready: it is ready to run, but is not currently running.
- Running: it is currently executing, it has a processor and its code is being executed.
- Blocked: it has been running, but its execution was interrupted by one of several different events.
- Dead: no longer active in any sense.

Methods of Providing Synchronization

- Semaphores
- Monitors
- Message Passing
- Java thread

Semaphores

- Dijkstra - 1965
- A *semaphore* is a data structure consisting of a counter and a queue for storing 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
Producer consumer problems

- 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 Operation

```
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
```

Semaphores: Release Operation

```
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
```

Producer Consumer Code

```
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;
```

Producer Consumer Code

```
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 Consumer Code

```
semaphore access, fullspots, emptyspots;
access.count = 0;
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); {hand over access}
   release(fullspots); {increase filled}
   end loop;
end producer;
```

Producer Consumer Code

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

Monitors

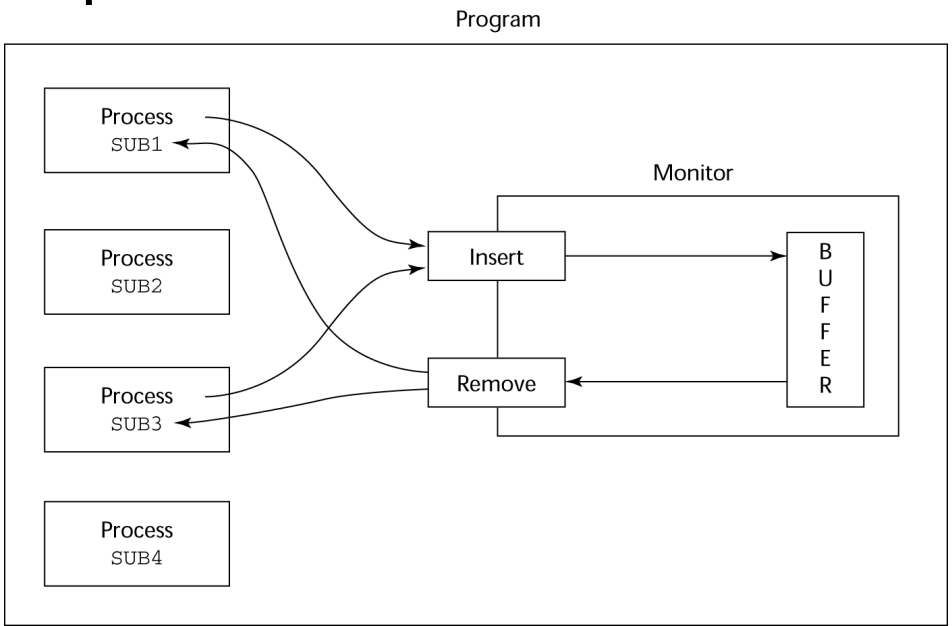
- Concurrent Pascal, Modula, Mesa, Ada, Java, C#
- The idea: encapsulate the shared data and its operations to restrict access
- A monitor is an abstract data type for shared data

Competition Synchronization

- Shared data is resident in the monitor
- 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



Message Passing

- Message passing means that one process sends a message to another process and then continues its local processing.
- The message may take some time to get to the other process and may be stored in the input queue of the destination process if the latter is not immediately ready to receive the message.
- Then the message is received by the destination process, when the latter arrives at a point in its local processing where it is ready to receive messages.
- This is called asynchronous message passing (because sending and receiving is not at the same time).

Asynchronous Message Passing

- Blocking send and receive operations:
  - A receiver will be blocked if it arrives at the point where it may receive messages and further no message can wait.{means message queue full}
  - A sender may get blocked if there is no room in the message queue between the sender and the receiver; however, in many cases, one assumes arbitrary long queues, which means that the sender will never be blocked.
- Non-blocking send and receive operations:
  - Send and receive operations always return immediately, returning a status value which could indicate that no message has arrived at the receiver.
  - The receiver may test whether a message is waiting and possibly do some other processing.

Synchronous Message Passing

- One assumes that sending and receiving takes place at the same time (there is no need for an intermediate buffer).
- This is also called rendezvous [To bring together at a certain place] and implies closer synchronization: the combined send-and-receive operation can only occur if both parties (the sending and receiving processes) are ready to do their part.
- The sending process may have to wait for the receiving process, or the receiving process may have to wait for the sending one.

Message Passing

- A mechanism to allow a task to indicate when it is willing to accept messages
- Tasks need a way to remember who is waiting to have its message accepted and some "fair" way of choosing the next message

Ada Support for Concurrency

- The Ada 83 Message-Passing Model
  - Ada tasks have specification and body parts; the spec has the interface, which is the collection of entry points:

```
task Task_Example is
                entry ENTRY_1 (Item : in Integer);
end Task_Example;
```

Task Body

- 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

```
accept entry_name (formal parameters)
  do…
end entry_name
```

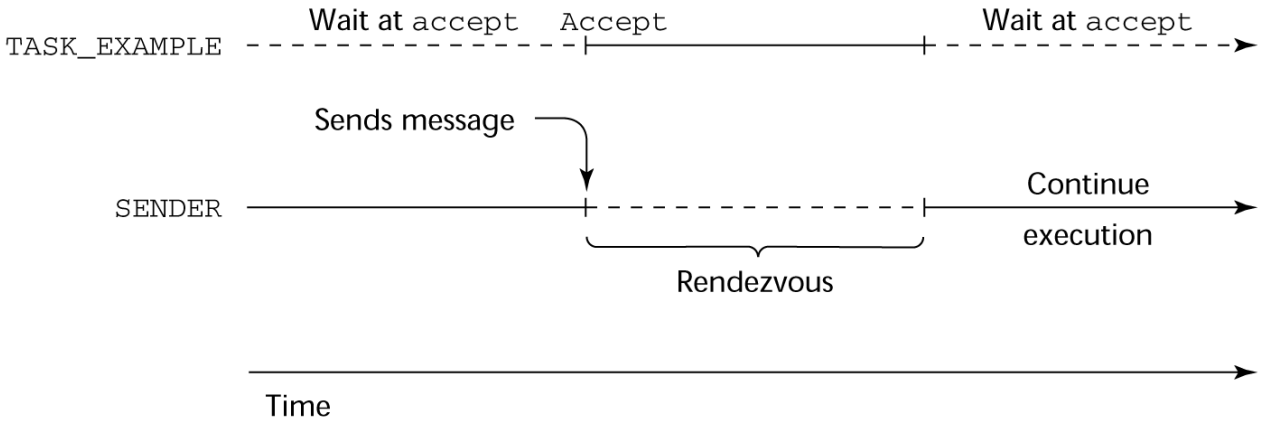
Example of a Task Body

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

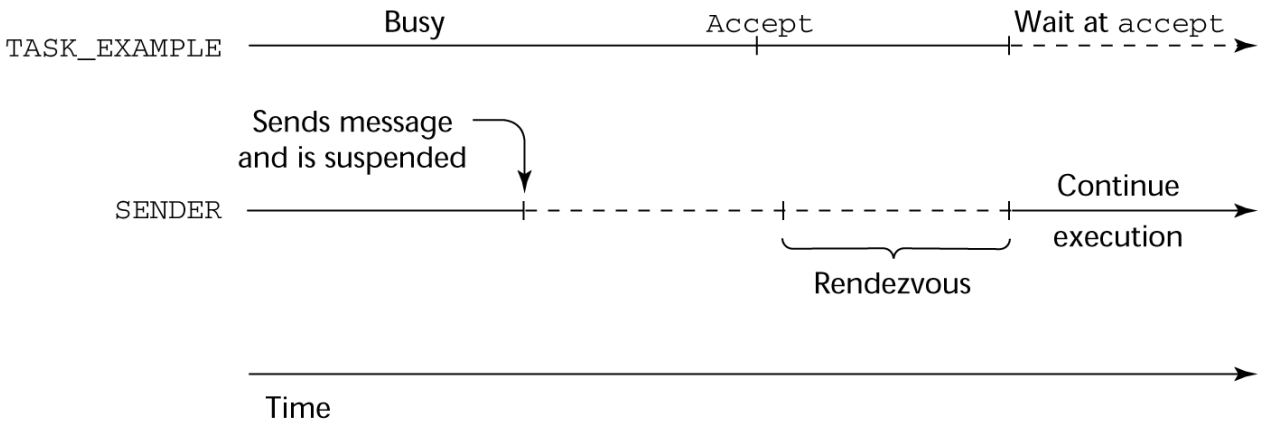
Ada Message Passing Semantics

- The task executes to the top of the `accept` clause and waits for a message
- During execution of the `accept` clause, the sender is suspended
- `accept` parameters can transmit information in either or both directions
- Every `accept` clause has an associated queue to store waiting messages

Rendezvous Time Lines



(a) TASK_EXAMPLE waits for SENDER



(b) SENDER waits for TASK_EXAMPLE

Message Passing: Server/Actor Tasks

- 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*
  - 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)

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;
```

Multiple Entry Points

- 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

A 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;
```

Semantics of Tasks with Multiple `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

Cooperation Synchronization with Message Passing

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

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

- An `accept` clause with a `when` clause is either *open* or *closed*
  - A clause whose guard is true is called *open*
  - A clause whose guard is false is called *closed*
  - 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, non-deterministically 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

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

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

Example of a Task with Guarded `accept` Clauses

```
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;
```

The `terminate` Clause

- 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

Java Threads

- The concurrent units in Java are methods named `run`
  - A `run` method code can be in concurrent execution with other such methods
  - The process in which the `run` methods execute is called a *thread*

```
Class myThread extends Thread
  public void run () {…}
}
…
Thread myTh = new MyThread ();
myTh.start();
```

Controlling Thread Execution

- The `Thread` class has several methods to control the execution of threads
  - The `yield` is a request from the running thread to voluntarily submit the processor
  - The `sleep` method can be used by the caller of the method to block the thread
  - The `join` method is used to force a method to delay its execution until the run method of another thread has completed its execution

Thread Priorities

- A thread's default priority is the same as the thread that create it
  - If `main` creates a thread, its default priority is `NORM_PRIORITY`
- Threads defined two other priority constants, `MAX_PRIORITY` and `MIN_PRIORITY`
- The priority of a thread can be changed with the methods `setPriority`

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

```
…
public synchrnoized void deposit( int i) {…}
public synchrnoized int fetch() {…}
…
```

- The above two methods are synchrnoized which prevents them from interfering with each other
- If only a part of a method must be run without interference, it can be synchronized thru synchrnoized statement

```
Synchrnoized (expression) statement
```

Cooperation Synchronization with Java Threads

- Cooperation synchronization in Java is achieved via `wait`, `notify`, and `notifyAll` methods
  - All 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
- The `notify` method is called to tell one waiting thread that the event it was waiting has happened
- The `notifyAll` method awakens all of the threads on the object's wait list

Java's Thread Evaluation

- Java's support for concurrency is relatively simple but effective
- Not as powerful as Ada's tasks

Deadlocks

- A law passed by the Kansas legislature early in 20th century:

  "…… When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until other has gone. ……"

Deadlock

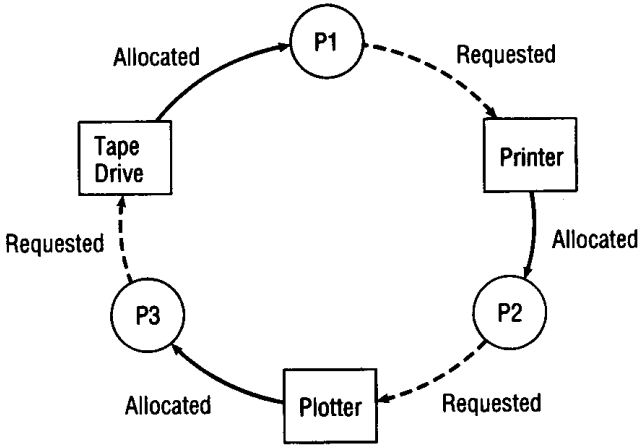


FIGURE 5.4
Case 4. Three processes, shown as circles, are each waiting for a device that has already been allocated to another process thus creating a deadlock.

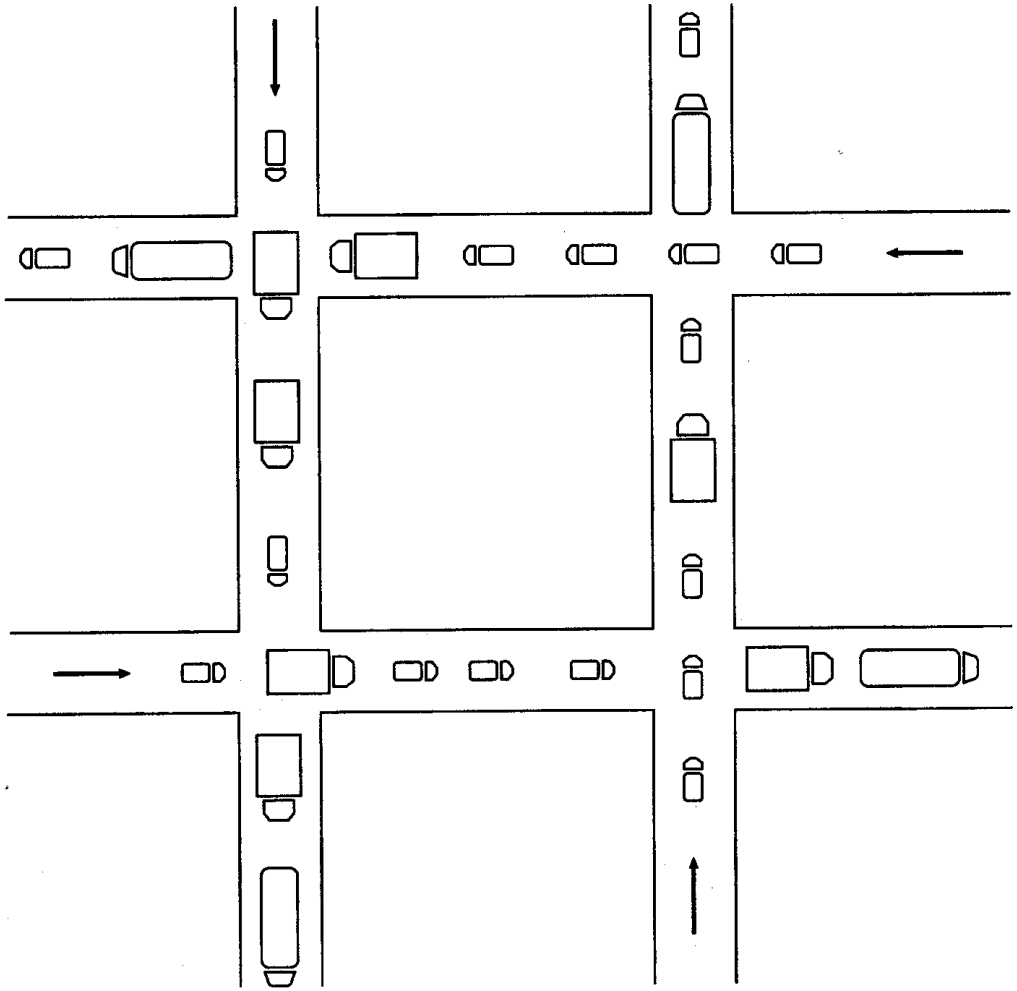


FIGURE 5.1
A classic case of traffic deadlock. This is gridlock, where no vehicles can move forward to clear the traffic jam.

Conditions for Deadlock

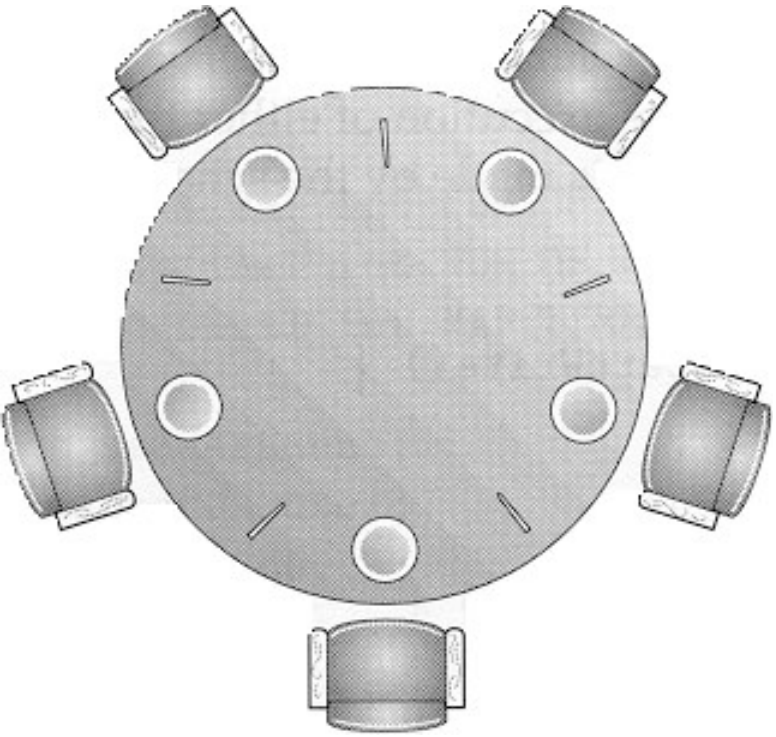
- Mutual exclusion: the act of allowing only one process to have access to a dedicated resource
- Hold-and-wait: there must exist a process that is holding at least one resource and is waiting to acquire additional resources that are currently being held by other processes.
- No preemption: the lack of temporary reallocation of resources, can be released only voluntarily.
- Circular waiting: result of above three conditions, each process involved in the impasse is waiting for another to voluntarily release the resource.

Strategy for Handling Deadlocks

- Prevention: eliminate one of the necessary conditions
- Avoidance: avoid if the system knows ahead of time the sequence of resource quests associated with each active processes
- Detection: detect by building directed resource graphs and looking for circles
- Recovery: once detected, it must be untangled and the system returned to normal as quickly as possible
  - Process termination
  - Resource preemption

uOttawa
L'Université canadienne
Canada's university

The Dining-Philosophers Problem



```
Semaphore chopStick[] = new Semaphore[5];

while (true){
    // get left chopstick
    chopStick[i].P();
    // get right chopstick
    chopStick[(i + 1) % 5].P();
    eating();
    // return left chopstick
    chopStick[i].V();
    // return right chopstick
    chopStick[(i + 1) % 5].V();
    thinking();
```

uOttawa

L'Université canadienne
Canada's university

Glossary for sample code
Java

uOttawa
L'Université canadienne
Canada's university

Creating threads by extending the thread class

```
// Custom thread class
public class CustomThread extends Thread
{
  ...
  public CustomThread(...)
  {
    ...
  }

  // Override the run method in Thread
  public void run()
  {
    // Tell system how to run custom thread
    ...
  }
  ...
}
```

```
// Client class
public class Client
{
  ...
  public someMethod()
  {
    ...
    // Create a thread
    CustomThread thread = new CustomThread(...);

    // Start a thread
    thread.start();
    ...
  }

  ...
}
```

uOttawa
L'Université canadienne
Canada's university

Creating threads by implementing the runnable interface

```
// Custom thread class
public class CustomThread
  implements Runnable
{
  ...
  public CustomThread(...)
  {
    ...
  }

  // Implement the run method in Runnable
  public void run()
  {
    // Tell system how to run custom thread
    ...
  }

  ...
}
```

```
// Client class
public class Client
{
  ...
  public someMethod()
  {
    ...
    // Create an instance of CustomThread
    CustomThread customThread
     = new CustomThread(...);

    // Create a thread
    Thread  thread = new  Thread(customThread);

    // Start a thread
    thread.start();
    ...
  }

  ...
}
```

A Circular Queue

```
// Queue
// This class implements a circular queue for storing int
// values. It includes a constructor for allocating and
// initializing the queue to a specified size. It has
// synchronized methods for inserting values into and
// removing values from the queue.

class Queue {
  private int [] que;
  private int nextIn,
              nextOut,
              filled,
              queSize;

  public Queue(int size) {
    que = new int [size];
    filled = 0;
    nextIn = 1;
    nextOut = 1;
    queSize = size;
  }  //** end of Queue constructor

  public synchronized void deposit (int item) {
    try {
      while (filled == queSize)
        wait();
      que [nextIn] = item;
      nextIn = (nextIn % queSize) + 1;
      filled++;
      notify();
    }  //** end of try clause
    catch(InterruptedException e) {}
  }  //** end of deposit method

  public synchronized int fetch() {
    int item = 0;
    try {
      while (filled == 0)
        wait();
      item = que [nextOut];
      nextOut = (nextOut % queSize) + 1;
      filled--;
      notify();
    }  //** end of try clause
    catch(InterruptedException e) {}
    return item;
  }  //** end of fetch method
}  //** end of Queue class
```

```
class Producer extends Thread {
  private Queue buffer;
  public Producer(Queue que) {
    buffer = que;
  }
  public void run() {
    int new_item;
    while (true) {
      //-- Create a new_item
      buffer.deposit(new_item);
    }
  }
}

class Consumer extends Thread {
  private Queue buffer;
  public Consumer(Queue que) {
    buffer = que;
  }
  public void run() {
    int stored_item;
    while (true) {
      buffer.fetch(stored_item);
      //-- Consume the stored_item
    }
  }
}
```

```
Queue buff1 = new Queue(100);
Producer producer1 = new Producer(buff1);
Consumer consumer1 = new Consumer(buff1);
producer1.start();
consumer1.start();
```