Concurrency

470410-1
Spring 2017
4/13/2017
Kyoung Shin Park
Multimedia Engineering
Dankook University

Criteria for Evaluating Concurrent Programming Constructs

- With advanced mechanisms for concurrency control, we should consider the following criteria:
  - Applicability to Centralized and Distributed system
  - Expressive Power
  - Modularity
  - Ease of Use
  - Program Structure
  - Real-Time Systems
  - Process Failure & Timeouts
  - Unanticipated Faults

Applicability

- Applicability to Centralized and Distributed system
  - Since there are times when both centralized systems and distributed systems need to interact, it is best if such constructs can work in both directions and both environments.
  - **Centralized system (the shared memory model)**
  - **Distributed system (the loosely-coupled model)**

Expressive Power

- Exclusion constraints
  - Does the construct provide for mutual exclusion?
- Priority constraints
  - Is the construct able to express priority between processes?
- Conditions
  - Does the construct permit that certain conditions must be satisfied before a process can execute? Such conditions would include the following:
    - Type of request (e.g. readers versus writers)
    - Time of request (e.g. timestamps)
    - Request parameters (e.g. filename)
    - Process information (e.g. for load balancing)
    - Priority relations (static)
    - Local state of resources (e.g. to prevent overloading)
    - History information (e.g. for aging)

Modularity

- We should consider two differing viewpoints
  - The operating system should regulate access to all shared resources
  - The operating system should regulate interaction between processes (shared memory versus message passing)
- This provides two orthogonal modularization criteria
  - Resources should be separated from each other – Each may contain synchronization and scheduling information and operations
  - Synchronization and scheduling should be separated from operation and state – We may need to allow for some global control

Ease of Use

- How difficult or complex is it to construct a solution using the given construct?
- Can a problem be broken into single parts?
- Is it easy to modify a solution? (e.g. add or change a constraint)

Program Structure

- Does the structure of the mechanism fit well with the overall program structures?
- Does the structure help the programmer avoid problems? (e.g. nested monito calls)

Real-Time Systems

- Concurrent programming techniques are not used much in real-time programming languages
  - They would need to include facilities for
    - Time-out
    - Time-of-day
    - Delay for a certain length of time
    - Etc
  - They would need run-time error handling
    - Even for unrecoverable errors

Failures

- Process Failures and Timeouts
  - We want to keep the failure of one process from affecting other processes
  - We need to be able to detect a failure and know
    - If it was caused by a timeout
    - If it was caused by another exception
  - It would be best if we can define exception handling procedures as part of the structure
    - Such procedures need to leave the state consistent
    - Such procedures may cost some efficiency
    - Such procedures should try to avoid mutual exclusion, if only synchronization is needed – If mutual exclusion is needed, it can be done more efficiently in hardware or firmware

Faults

- Unanticipated Faults
  - Assuming no exception handler provided
  - We can provide a recovery block of code
    - That allows backtracking to a state before the error
    - That is able to detect an error
    - That could permit a retry with a different algorithm
  - This concept is fairly untried
    - It may not be feasible for complex situations
    - It may be too expensive

Semaphores

- Semaphore
  - A semaphore is "an integer variable that apart from initialization, is accessed only through two standard atomic operations: **wait** and **signal**" (SilGal98)
  - "These operations were originally termed **P** (for **wait**; from the Dutch ***proberen***, to test) and **V** (for **signal**; from ***verhogen***, to increment)" (SilGal98)
  - Dijkstra introduced these terms and used these operations in the operating system
- For semaphore s

```
wait(s): while s <= 0
                do no-op;
         s--;
signal(s): s++;
```

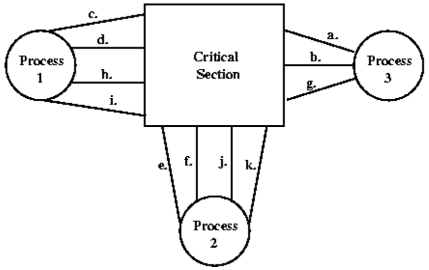
  - Where wait(s) is the same as P(s) and signal(s) is the same as V(s)

Semaphores

- Semaphore actions
  - Must be atomic actions
  - Must be indivisible
  - Must be uninterruptible
- Further, both the test of the semaphore and the change of the value of the semaphore must happen together
- Note that s can be any integer
- There are two types of semaphores:
  - Two-valued (could be represented as boolean or int)
  - Integer (could be multi-valued)

Semaphore Example (SS)

- Semaphore use in a sequential system
  - Consider a sequential system with 3 running processes, Process 1, 2, 3
  - Each of the processes has access to a shared semaphore, s
  - Each process has a critical section controlled by s



Semaphore Example (SS)

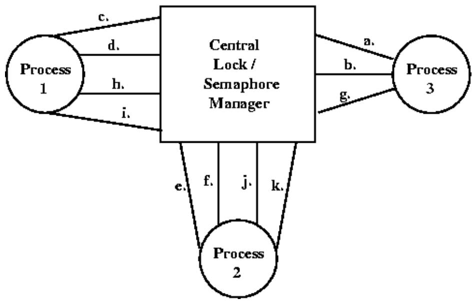
- Initially s = 1
  - With no processes in their critical section
- Steps
  a. Process3 requests its critical sections
     - Since s = 1, s <= 0, so s is decremented by 1, making s = 0
  b. Process3 is granted access to its critical section
  c. Process1 requests its critical section,
     - But s <= 0, as s = 0
  d. Process1 starts a busy wait,
     - Continually retesting s until s > 0
  e. Process2 requests its critical section
     - But s <= 0, as s = 0
  f. Process2 starts a busy wait,
     - Continually retesting s until s > 0

Semaphore Example (SS)

g. Process3 finishes its critical section,
   - So s is incremented by 1, making s = 1, releasing access to the critical section
h. Process1 checks the value of s
   - Since s = 1, s ! <= 0, So s is decremented by 1, making s = 0, and Process1 is granted access to its critical section
i. Process1 finishes its critical section
   - So s is incremented by 1, making s = 1, releasing access to the critical section
j. Process2 checks the value of s
   - Since s = 1, s ! <= 0, So s is decremented by 1, making s = 0, and Process2 is granted access to its critical section
k. Process2 finishes its critical section
   - So s is incremented by 1, making s = 1, releasing access to the critical section

Semaphore Example (DS)

- Semaphore use in a distributed system
  - Consider a distributed system with 3 running processes, Process 1, 2, 3
  - Each of the processes has access to a shared semaphore, s
  - Each process has a critical section controlled by s
  - Assume the semaphore/lock is controlled by a centralized lock manager



Semaphore Example (DS)

- Initially s = 1
  - With no processes in their critical section
- Steps
  a. Process3 requests its critical sections
     - Since s = 1, s <= 0, so s is decremented by 1, making s = 0
  b. Process3 is granted access to its critical section
  c. Process1 requests its critical section,
     - But s <= 0, as s = 0
  d. Process1 is placed on a **queue**,
     - Until s > 0
  e. Process2 requests its critical section
     - But s <= 0, as s = 0
  f. Process2 is **queued**,
     - Until s > 0

Semaphore Example (DS)

g. Process3 finishes its critical section,
   - So s is incremented by 1, making s = 1, releasing access to the critical section
h. The central manager checks the semaphore value s
   - Since s = 1, s ! <= 0, So s is decremented by 1, making s = 0, and Process1 is granted access to its critical section by the central manger
i. Process1 finishes its critical section
   - So s is incremented by 1, making s = 1, releasing access to the critical section
j. The central manager checks the semaphore value s
   - Since s = 1, s ! <= 0, So s is decremented by 1, making s = 0, and Process2 is granted access to its critical section by the central manager
k. Process2 finishes its critical section
   - So s is incremented by 1, making s = 1, releasing access to the critical section

Semaphore for Producer/Consumer Problem

```
sem nfull = 0;
sem nempty = N;
sem mutexP, mutexC = 1;
info buffer[N]; int in, out = 0;
producer() {
  begin
    create one unit of type info, U;
    P(mutexP); //one producer
    P(nempty); //wait for empty
    buffer[in] = U;
    in = (in++) % N;
    V(nfull); //signal full
    V(mutexP);
  end;
}
```

```
consumer() {
  begin
    P(mutexC); //one consumer
    P(nfull); //wait for full
    U= buffer[out];
    out = (out++) % N;
    V(nempty); //signal empty
    V(mutexC);
    consume one unit of type info, U;
  end;
}
```

Semaphore for Reader/Writer Problem

```
int nreaders = 0;
sem mutex, wmutex, srmutex = 1;
reader() {
   P(mutex);
   nreaders++; //#reader++
   if (nreaders == 0)
     P(wmutex) ; //wait until no writer
   V(mutex);

   ... read ... ;

   P(mutex);
   nreaders --; //#reader--
   if (nreaders == 0)
      V(wmutex); //signal
   V(mutex);
}
```

```
writer () {
    P(srmutex);
    P(wmutex);

    ... write ... ;

    V(wmutex);
    V(srmutex);
}
```

mutex protects modifications to **nreaders**
wmutex protects makes sure that only readers or just **one** writer is active
V(wmutex) should unblock a waiting reader before **V(srmutex)** can release a waiting writer

Disadvantage of Semaphores

- Simple algorithms require more than one semaphore
  - This increases the complexity of semaphore solutions to such algorithms
- Semaphore are too low level.
  - It is easy to make programming mistakes
- The programmer must keep track of all calls to wait and to signal the semaphore.
  - If this is not done in the correct order, programmer error can cause deadlock.
- Semaphores are used for both condition synchronization and mutual exclusion.
  - These are distinct and different events, and it is difficult to know which meaning any given semaphore may have.
- What happens if system crashes when one process is in the critical sections?

Monitors

- A monitor is a high-level synchronization primitive
  - Developed by Hoare and Brinch Hansen
  - A programming language construct
  - A compiler-supported data structure with
    - Procedures
    - Variables
    - Data structures
  - Similar to today's classes and objects, e.g. Concurrent Pascal, Java
- Outside processes may
  - Call monitor procedure
  - Not access monitor data structures
- Only one process is active in monitor at once
  - Ensuring mutual exclusion
  - Blocking other processes are blocked
- It may be implemented using binary semaphores

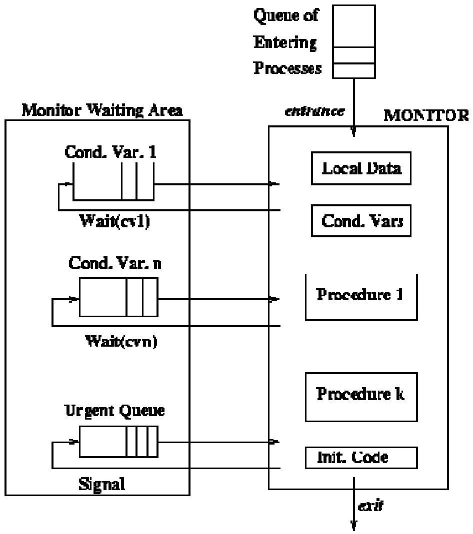
Monitor Definitions

- A monitor is an abstract mechanism which
  - Encapsulates abstract resources, and
  - Provides functions to manipulate those resources
- Can be though of as an object(or ADT) containing
  - A data structure, and operations (methods) for manipulating that data structure, where only one process can execute an operation at a time.
  - It other words, it is an object with synchronization.
- Only allows the resources to be accessed through the monitor operations:
  - Only the procedure names of the monitor operations are visible outside the monitor.
  - Monitor procedures may only access monitor variables within the monitor itself.
  - All shared variables declared within the monitor are initialized before execution begins.
- Provides mutual exclusion:
  - Only one process may be executing within a monitor at any given time.
  - Concurrent processes can use the monitor resources.

Advantages of Monitors

- A process calling a monitor procedure (or method) can ignore the actual implementation (as in any abstract data type).
- Once a monitor is correctly programmed, it remains correct, despite the number of processes executing (as in object-oriented programming).
- The implementation of a monitor can be changed without affecting the application or the user's view of the monitor resources (as in object-oriented programming).
- Monitors provide mutual exclusion on a higher level than semaphores or conditional critical regions.

Representation of a Monitor



Condition Variables

- Condition variables allow a process executing within the monitor to be put to sleep to wait for some condition to be set (signaled).
  - They are used to delay a process that cannot safely proceed until there is a change in the state of the monitor.
  - This avoids deadlock within the monitor.
- Condition variables can also awaken a sleeping process to let it be actively executing again within the monitor.
  - Condition variables wake up delayed or suspended processes within the monitor.
- A condition variable is just a data structure (or class) consisting of
  - A boolean value
  - A queue of delayed processes
- A condition variable is a shared data variable within the monitor.

Condition Variables

- Commands related to condition variables include:
  - Wait(c):
    - The process currently active in the monitor suspends execution and gives up mutual exclusion to the monitor until the condition variable c is signaled. It is placed on the end of the queue of delayed processes waiting for c to be signaled.
  - Signal(c):
    - The process at the front of the queue is awakened and resumes execution within the monitor. If the queue connected to the condition variable c is empty, nothing happens; this is equivalent to a skip operation.
- A drawback of condition variables is that compilers for monitor-supporting languages usually rely on shared memory.

Monitors vs. Semaphores

- Wait versus P(s) and Signal versus V(s)
  - The signal command has no effect if there is no suspended process. V(s) always increments s.
  - The wait command always delays until there is a signal command. P(s) only delays if s is not positive.
  - The process that executes the signal command is currently executing within the monitor. V(s) and P(s) may be used outside the critical section.

Disadvantages of Monitors

- Monitors can exhibit an absence of concurrency, when a monitor encapsulates a resource since only one process can be active at a time within the monitor.
- When using nested monitor calls, there is a possibility of deadlock.

Implementation Issues for Monitors

- Suppose process Q is waiting on the condition variable c in a monitor.
  - Further suppose that process P is active in the monitor and executes c. signal, waking up Q.
  - Now which process continues to be active in the monitor?
- This turns out to be an implementation issue
  - (i.e., how the monitors are implemented).
- When P signals Q, there are three choice of actions:
  1. P may continue to execute in the monitor. However, if it does so, P may alter the condition that awakened Q.
  2. P may wait (suspend) while Q executes in the monitor until Q is done or some other condition becomes true. This is the method preferred by Hoare.
  3. P executes the signal command and immediately leaves the monitors. In other words, the signal command is the last line of the procedure P executes. This is the method preferred by Brinch Hansen.

Monitor for Producer/Consumer Problem

```
monitor ProducerConsumer {
  int itemCount;
  condition full;
  condition empty;
  procedure add(item) {
    while (itemCount == BUFFER_SIZE) {
      wait(full);
    }
    putItemIntoBuffer(item);
    itemCount = itemCount + 1;
    if (itemCount == 1) {
      notify(empty);
    }
  }
  procedure remove() {
    while (itemCount == 0) {
      wait(empty);
    }
    item = removeItemFromBuffer();
    itemCount = itemCount – 1;
    if (itemCount == BUFFER_SIZE - 1) {
      notify(full);
    }
    return item;
  }
}
```

Monitor for Producer/Consumer Problem

```
procedure producer() {
  while (true) {
    item = produceItem();
    ProducerConsumer.add(item);
  }
}
procedure consumer() {
  while (true) {
    item = ProducerConsumer.remove();
    consumeItem(item);
  }
}
```

References

- http://www.cs.colostate.edu/~cs551/CourseNotes/ConcurrentConstructs/ConcurrentTOC.html