Process Synchronization – Part II

CS 1550 – Introduction to Operating Systems

Prof. Taieb Znati
Department Computer Science
University of Pittsbrugh

Outline

- Classic Synchronization Problems
  - Bounded Buffer
  - Readers-Writers
  - Dinning Philsophers
- Monitor Abstract Data Type
  - Monitor structure
  - Condition construct
  - Monitor implementation
  - Dinning Philosophers implementation
- Conclusion

Classic Synchronization Problems

- A number of synchronization problems have been used to represent a larger class of concurrency control problems
  - There are typically used for testing newly proposed synchronization schemes
- **Bounded Buffer Problem** – used to illustrate the power of synchronization primitives
- **Readers-Writers Problem** – used to illustrate synchronization between two classes of processes sharing access to a database
- **Dinning Philosophers Problem –** used to illustrate a simple representation of the need to allocate resources among several processes in a dead-lock free and starvation free-manner

Bounded-Buffer Problem

CLASSIC SYNCHRONIZATION PROBLEMS

Bounded-Buffer Problem

- The solution uses both aspects of a counting semaphore:
  - Mutual exclusion to CS – the mutex semaphore
  - Limited resource control – **full** and **empty** semaphores
- Shared data – semaphore **full**, **empty**, and **mutex**
  - full represents the number of used entries in the buffer
  - empty represents the number unused entries in the buffer
  - mutex is used for CS mutual exclusion
- Initially, buffer is empty
  - full = 0, empty = N, mutex = 1

Producer-Consumer – Bounded Buffer

- A producer adds new items into a shared buffer
- A consumer consumes items from the shared buffer
- **Problem Constraints**
  - The consumer must wait if buffers are empty – **scheduling constraint**
  - The producer must wait if buffers are full – **scheduling constraint**
  - Only one thread can manipulate the buffer at a time – **mutual exclusion**
- The solution involves both scheduling and mutual exclusion

Producer-Consumer Solution

- A semaphore is need for each constraint

```
semaphore mutex = 1;
semaphore nFreeBuffers = N;
semaphore nLoadedBuffers = 0;
```

```
Producer() {

   wait(mutex);
   // add 1 item in
   the buffer //
   signal(mutex);
}
```

```
Consumer() {

   wait(mutex);
   // consume one
   item from the
   buffer //
   signal(mutex);
}
```

Producer-Consumer Solution

- Each constraint needs a semaphore

```
semaphore mutex = 1;
semaphore nFreeBuffers = N;
semaphore nLoadedBuffers = 0;
```

```
Producer() {
  wait(nFreeBuffers);
   wait(mutex);
   // add an item to
   the buffer //
   signal(mutex);
}
```

```
Consumer() {
 wait(nLoadedBuffers);
  wait(mutex);
   // consume one item
   from the buffer //
  signal(mutex);
}
```

Producer-Consumer Solution

- Each constraint needs a semaphore

```
semaphore mutex = 1;
semaphore nFreeBuffers = N;
semaphore nLoadedBuffers = 0;
```

```
Producer() {
 wait(nFreeBuffers);
   wait(mutex);
   // add one item in
   the buffer //
   signal(mutex);
 signal(nLoadedBuffers);
}
```

```
Consumer() {
 wait(nLoadedBuffers);
   wait(mutex);
   // consume one item
   from the buffer //
   signal(mutex);
 signal(nFreeBuffers);
}
```

Readers and Writers Problem

CLASSIC SYNCHRONIZATION PROBLEMS

Readers-Writers Problem

- The readers-writers problem consists of two types of processes
  - **Readers** – are permitted to only read the resource, concurrently with a number of other readers
  - **Writers** – are permitted to update, read and write, the resource and, thus, must have exclusive access to the resource
- Depending on the fairness policy required, there are different variations of the basic readers-writers problem
  - When implemented, the fairness policy aims to prevent indefinite exclusion of readers or writers or both

Readers-Writers Variants

- **Readers-over-Writers Policy** – readers have priority over writers
  - **No** reader should be kept waiting, unless a writer has already obtained permission to update the resource
  - This policy may result in **writer starvation** – indefinite postponement of write requests, if there is a continuous stream of read requests
- **Writer-Priority Policy** – When a writer arrives, only those readers already granted permission to read are allowed to complete their operation
  - All new readers arriving after the writer are postponed until completion of the update operation
  - This policy may result in **reader starvation** – indefinite postponement of read requests, when there is an interrupted stream of writers arriving at the resource

Readers-over-Writers Variant – Single writer and multiple readers

- Problem constraints
  - Multiple readers allowed in CS
  - Readers have priority over writers
    - As long as there is a reader(s) in the CS, no writer may enter the CS
    - CS must be empty for a writer to get in
- Semaphores are required purely for mutual exclusion
  - No limits on writing or reading the buffer
- Shared data
  - semaphore mutex, wrt;
- Initially no items to be read, until writer creates one
  - mutex = 1, wrt = 1, readcount = 0

Readers-over-Writers – Writer Process

```
Writer Process
do {
        …
        wait(wrt);
          …
          Perform writing
          …
        signal(wrt);
} while (true)
```

- When a writer is in its CS, readers are waiting
- When a writer executes signal(wrt), the execution of either the waiting readers or a single waiting writer is resumed
  - The selection is made by the scheduler

Readers-over-Writers – Reader Process

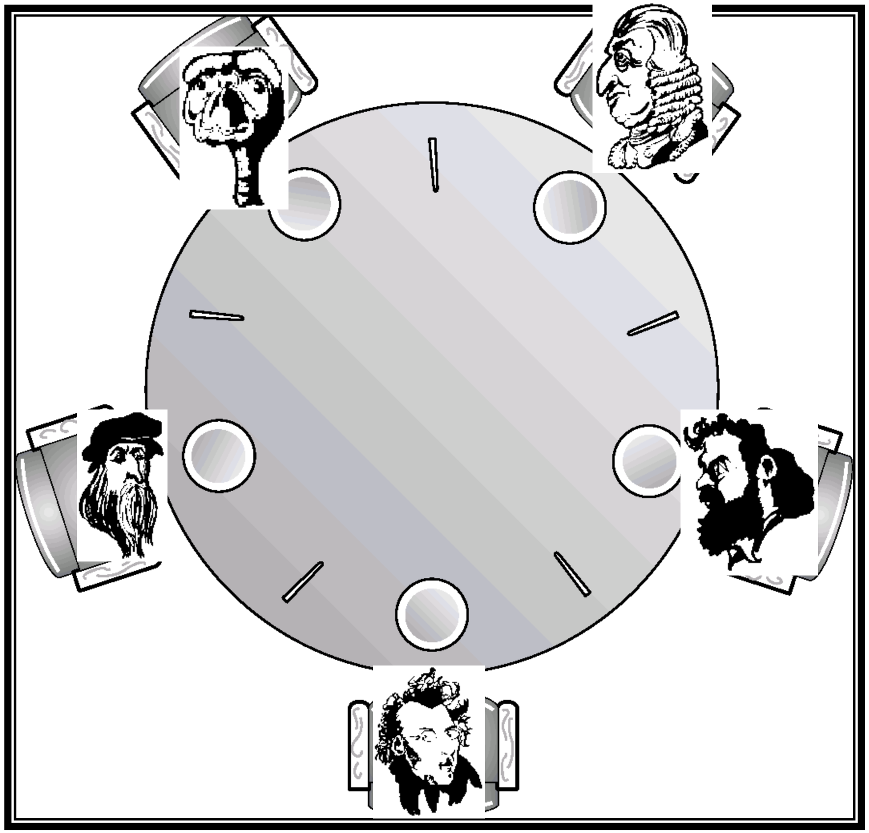
Reader Process

```
Do{ wait(mutex);
      readcount++;
      if (readcount == 1)
       wait(wrt);
      signal(mutex);
        Reading is performed
      wait(mutex);
      readcount--;
      if (readcount == 0)
       signal(wrt);
      signal(mutex):
} while (true)
```

- The 2nd, 3rd, … queue on mutex if 1st reader is blocked on **wrt**, waiting for a writer to leave.
  - Access to **readcount** is guarded
- 1st reader blocks if any writers in CS
- After it gets in CS, 1st reader opens the "floodgates" for all readers.
  - Access to readcount is guarded
- Last reader out lets any queued writers in

Dinning Philosopher Problem

- Five philosophers, p_i ($0 \leq i \leq 4$), sit around a table in a middle of which a bowl of rice
  - There is a plate in front of each philosopher and five chopsticks around the table
  - At unspecified times, each of the philosophers may wish to eat
    - To do so, philosopher must pick left and right chopsticks, one at a time.
      - Only then can the philosopher eat



Dining-Philosophers Problem

- Philosophers are considered to be processes and chopsticks are considered to be resources
  - The problem is to design the philosopher processes so that none of them starve because of the unavailability of chopsticks
- Solution requirements
  - **Deadlock Free** – prevent a situation where each philosopher acquires one of the chopstick and waits indefinitely for the other to be released
  - **Conspiracy Free** – prevent a situation where two or more philosophers conspire in such a way that one of the remaining philosophers could be prevented indefinitely from acquiring two chopsticks

Dining-Philosophers – Simple Solution

- **Philosopher(i)**

```
do {
      wait(chopstick[i])                  // Left chopstick
      wait(chopstick[(i+1) % 5])          // Right chopstick
        Eat for a while
          …
      signal(chopstick[i]);
      signal(chopstick[(i+1) % 5]);
          …
        Think for a while

} while (true)
```

Dinning Philosophers – Deadlock and Stravation Problem

- **Deadlock:** If all philosophers decide to eat at the same time and pick up the chopstick on their left
  - None will be able to pick up a chopstick on their right.
- **Possible Solutions**:
  - **1**. Allows at most four philosophers to be sitting simultaneously at the table.
  - **2**. Allows a philosopher to pick up his/her chospstick only if both of them are available.
  - **3.** Allows only an “odd” philosopher to pick up left chopstick first and then right chopstick, whereas an “even” philosopher picks up right chopstick first and then left chopstick.
- **Starvation:** A philosopher may never gain access to two chopsticks if the philosophers next to him are always eating.

Synchronization Mechanisms

MONITOR ADT

Semaphore Limitations

- Although useful as a general-purpose mechanism for controlling access to critical sections, their use does not guarantee that access will be mutually exclusive or deadlock will be avoided
  - Programming errors, such omitting a wait() before a process enters its critical section, or inverting a wait() and a signal(), or replacing a wait() with a signal, or ..
  - Programming errors manifest themselves only if a particularly execution timing pattern occurs
    - Difficult to detect error, further compounded by the inability to repeat the error in order to locate it
- There is a need for **higher level construct** that guarantee appropriate access to critical sections

Monitors

- The idea of monitor is based on the principles of abstract data types
  - For any distinct data type there should be a well-defined set of operations through which any instance of that data type must be manipulated
- A monitor is defined as a collection of data, which represent the resource to be controlled by the monitor, and a set of functions to manipulate that resource
  - Monitor is a high-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.

Monitor Implementation

- The implementation of the monitor construct must guarantee
  1. Access to resource is possible only via one of the monitor functions
  2. Functions are mutually exclusive
     - At any given time, only one process may be executing inside the monitor
     - Other processes calling a monitor function are delayed until the process leaves the monitor

Monitor Syntax

```
monitor monitor_name
{ Shared variable declarations
    function P1 (...) {
      . . . }
    function P2 (...) {
      . . . }

   ⋮

    function Pn (...) {
      . . . }
  Initialization code ( . . . ){
          . . . }
}
```

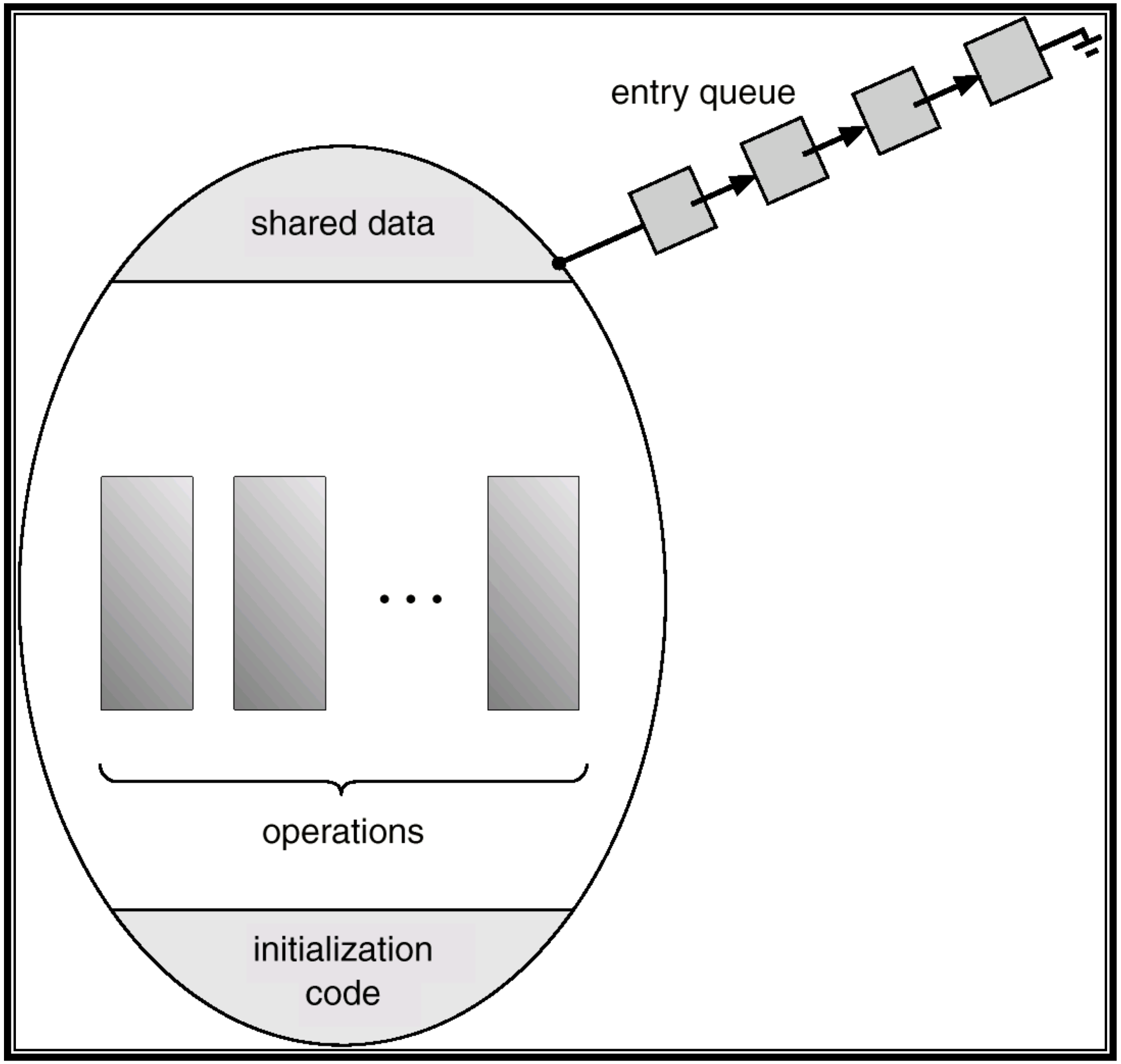
Monitor Usage

- In its basic definition, a monitor is **NOT** sufficient to implement a critical section by preventing access to a resource
  - It does not provide any means for processes to communicate with one another
    - Additional synchronization mechanisms are needed to allow a process to leave a monitor and to suspend itself while waiting for some condition to be satisfied
    - When the condition becomes true, the suspended process must be reactivated and allowed to resume execution inside the monitor
- The mechanisms are provided by **condition construct** and **two operations** to be invoked on a condition variable

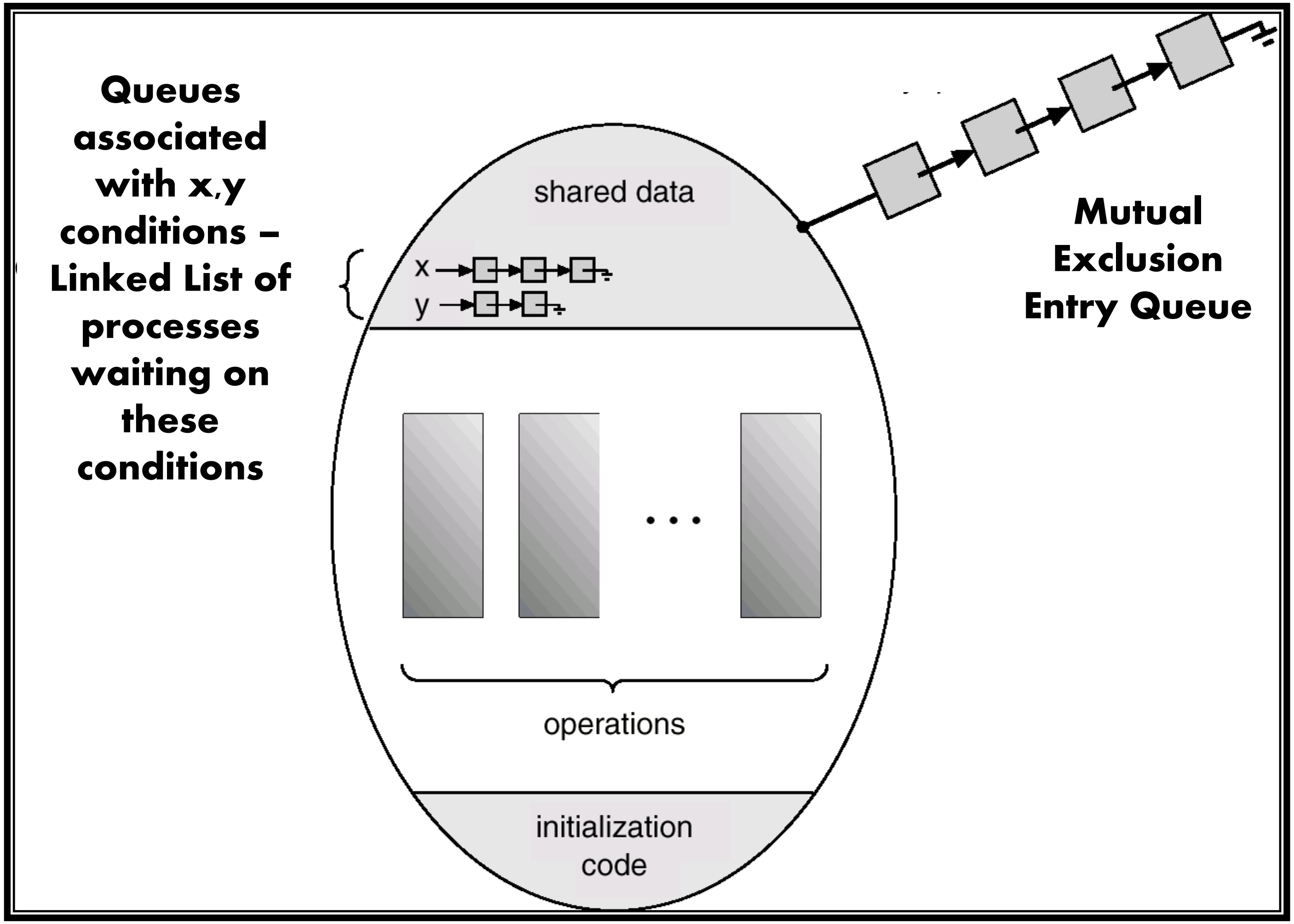
Monitors Condition Construct And Operation

- To allow a process to wait within the monitor, a variable of type **condition** is used
  - **condition** x, y;
- Condition variable can only be used with the operations wait() and signal()
  - The operation c.wait(), wait on condition c, causes the executing process to be unconditionally suspended (blocked) and placed on a queue associated with the condition c
    - Process invoking c.wait() operation remains suspended until another process invokes the x.signal() operation
  - The x.signal operation resumes exactly one suspended process
    - If no process is suspended, then the signal operation has no effect

Schematic View of a Monitor: Entry Queue



Monitor With Condition Variables



Signal Operation Semantics

- Signal operation allows a waiting process to reenter the monitor at the point its execution was suspended
  - Since only one process may execute in the monitor at a time, a decision has to be made with respect to **which** of the **signaling process** or the **signaled process** can execute
- Two options are possible
  - **Signal-and-Wait** – The signaling process either waits until the signaled process leaves the monitor or waits for another condition
  - **Signal-and-Continue** – The signaled process either waits until the signaling process leaves the monitor or waits for another condition

Dining Philosophers Example

```
monitor DiningPhilosophers
{
 enum {THINKING, HUNGRY, EATING} state[5];
 condition self[5];
 void pickup(int i)
 void putdown(int i)
 void test(int i)
 void init() {
  for (int i = 0; i < 5; i++){
      state[i] = THINKING;
   }
}
```

```
do{
        Think for a while

    DiningPhilosopher.pickup(i)

        Eat for a while

    DiningPhilosopher.putdown(i)

}while(true)
```

Dining Philosophers

```
void pickup(int i) {
  state[i] = HUNGRY;
  test[i];
  if (state[i] != EATING)
   self[i].wait();
}
```

```
void putdown(int i) {

   state[i] = THINKING;
   test((i+4) % 5);
   test((i+1) % 5);
}
```

```
void test(int i) {
   if ( (state[(i + 4) % 5] != EATING)
      && (state[i] == HUNGRY) &&
      (state[(i + 1) % 5] != EATING))
       {
         state[i] = EATING;
         self[i].signal();
        }
}
```

Monitor Implementation Using Semaphores

- **semaphore mutex; // (initially = 1)**
  semaphore next; // (initially = 0)
  int next-count = 0;

```
wait(mutex);
  Body of F;
 if (next-count > 0)
    signal(next)
 else
    signal(mutex
```

- Mutual exclusion within a monitor is ensured.

Monitor Implementation

- For each condition variable ***x***, we have:
  semaphore x-sem;
  int x-count = 0;

- The operation **x.wait()** can be implemented as:

```
x-count++;
if (next-count > 0)
     signal(next);
else
     signal(mutex);
wait(x-sem);
x-count--;
```

Monitor Implementation

- The operation **x.signal()** can be implemented as:

```
if (x-count > 0) {
   next-count++;
   signal(x-sem);
   wait(next);
   next-count--;
}
```

Monitor Implementation – Resuming Order of Processes within Monitor

- Conditional-wait construct: x.wait(c);
  - c – integer expression evaluated when the wait operation is executed.
  - value of c (a priority number) stored with the name of the process that is suspended.
  - when x.signal() is executed, process with smallest associated priority number is resumed next.
- Check two conditions to establish correctness of system:
  - User processes must always make their calls on the monitor in a correct sequence.
  - Must ensure that an uncooperative process does not ignore the mutual-exclusion gateway provided by the monitor, and try to access the shared resource directly, without using the access protocols.

Conclusion

- Classic Synchronization Problems
  - Bounded Buffer
  - Readers-Writers
  - Dinning Philosophers
- Monitor Abstract Data Type
  - Monitor structure
  - Condition construct
  - Monitor implementation
  - Dinning Philosophers implementation
- Conclusion