Operating Systems 2023 Spring Term
Week 6

Dr. Emrah İnan (emrahinan@iyte.edu.tr)

Synchronization II

April 13, 2023

Week 5: Sample Glossary

- **busy waiting:** A practice that allows a thread or process to use CPU time "continuously" while waiting for something. An I/O loop in which an I/O thread continuously reads status information while waiting for I/O to complete. (on Page 1241)
- **critical section:** A section of code responsible for changing data that must only be executed by one thread or process at a time to avoid a race condition. (on Page 1244)
- **spinlock:** A locking mechanism that continuously uses the CPU while waiting for access to the lock. (on Page 1270)
- **starvation:** The situation in which a process or thread "waits indefinitely -> hunger" within a semaphore. Also, a scheduling risk in which a thread that is ready to run is never put onto the CPU due to the scheduling algorithm; it is starved for CPU time. (on Page 1270)
- **semaphore:** An integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal(). (on Page 1268)

Mutual Exclusion vs Progress

- Example: Two people - A and B - want to use the changing room in the clothes department of a department store.
- "mutual exclusion": Person A goes to the changing room to try on the chosen garments. While person A is inside the changing room -> occupied sign (indicating that no one else can come in) Hence, person B has to wait until person A is done using the changing room. (Only one thread in critical section at a time)
- "progress": Person A decides to use the changing room first, but cannot decide how many clothes to take inside. Person B cannot enter the changing room even though it is empty -> blocking
  Progress -> If several simultaneous requests, must allow one to proceed (Must not depend on threads outside critical section)

Mutex (Mutual Exclusion) Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- OBJECTIVE: To protect critical sections and thus prevent race conditions. That is, a process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section -> by first acquire() a lock then release() the lock (Boolean variable indicating if lock is available or not)
- Calls to acquire() and release() must be atomic
  Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  This lock therefore called a spinlock

acquire() and release()

A process that attempts to acquire an unavailable lock is blocked until the lock is released.

```
acquire() {
    while (!available)
        ; /* busy wait */
    available = false;
}
release() {
    available = true;
}
do {
acquire lock
    critical section
release lock
   remainder section
} while (true);
```

Busy waiting - spinlock

- Disadvantage of the implementation: it requires busy waiting. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire().
  This continual looping is clearly a problem in a real multiprogramming system, where a single CPU core is shared among many processes.
- Type of mutex lock -> spinlock because the process “spins” while waiting for the lock to become available.
  (+) Spinlocks -> no context switch is required when a process must wait on a lock, and a context switch may take considerable time. If a lock is to be held for a "short duration", one thread can “spin” on one processing core while another thread performs its critical section on another core.
  "short duration" -> the general rule is to use a spinlock if the lock will be held for a duration of less than two context switches.

Semaphore

- Semaforo (Italian) -> traffic light
- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore **S** – integer variable
- Can only be accessed via two indivisible (atomic) operations
- wait() -> P -> proberen (Dutch) -> to test
  signal() -> V -> verhogen (Dutch) -> to increment

`wait()` and `signal()`

- Originally called `P()` and `V()`

Definition of the `wait()` operation

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

Definition of the `signal()` operation

```
signal(S) {
    S++;
}
```

Semaphore Usage I

- Counting semaphore – integer value can range over an unrestricted domain
  "The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a signal() operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used."
- Binary semaphore – integer value can range only between 0 and 1 (Same as a mutex lock -> on systems that do not provide mutex locks, binary semaphores can be used instead for providing mutual exclusion)

Semaphore Usage II

- Consider two concurrently running processes P_1 and P_2.
  Suppose we require a statement S_1 to happen before S_2
  Let P_1 and P_2 share a common semaphore "synch" initialized to 0
  For P1 insert statements:
  S_1;
  signal(synch);
  For P2 insert statements:
  wait(synch);
  S_2;
- Because synch is initialized to 0, P2 will execute S2 only after P1 has invoked signal(synch), which is after statement S1 has been executed.

Semaphore Implementation

- The implementation of mutex locks discussed suffers from busy waiting. The definitions of the wait() and signal() semaphore operations just described present the same problem.
- Must guarantee that no two processes can execute the wait() and signal() on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the wait and signal code are placed in the critical section
  - Could now have busy waiting in critical section implementation
    But implementation code is short
    Little busy waiting if critical section rarely occupied
  - Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore Implementation with no Busy waiting I

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  value (of type integer)
  pointer to next record in the list
- Two operations:
  block – place the process invoking the operation on the appropriate waiting queue
  wakeup – remove one of processes in the waiting queue and place it in the ready queue
- typedef struct{
  int value;
  struct process *list;
  } semaphore;

Semaphore Implementation with no Busy waiting II

When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process can suspend itself. The suspend operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state.

When a process must wait on a semaphore, it is added to the list of processes. A signal() operation removes one process from the list of waiting processes and awakens that process.

```
wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      add this process to S->list;
      block();
   }
}

signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
   }
}
```

Semaphore Implementation with no Busy waiting III

- Note that in this implementation, semaphore values may be negative, whereas semaphore values are never negative under the classical definition of semaphores with busy waiting. If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore. This fact results from switching the order of the decrement and the test in the implementation of the wait() operation.
- It is important to admit that we have not completely eliminated busy waiting with this definition of the wait() and signal() operations. Rather, we have moved busy waiting from the entry section to the critical sections of application programs. Furthermore, we have limited busy waiting to the critical sections of the wait() and signal() operations, and these sections are short.

Liveness

- Processes may have to wait indefinitely while trying to acquire a synchronization tool such as a mutex lock or semaphore.
- Waiting indefinitely violates the progress and bounded-waiting criteria discussed at the beginning of this chapter.
- Liveness refers to a set of properties that a system must satisfy to ensure processes make progress.
- Indefinite waiting is an example of a liveness failure.
- The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a signal() operation.

Deadlock

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- To illustrate this, consider a system consisting of two processes, P0 and P1, each accessing two semaphores, S and Q, set to the value 1
- Suppose that P0 executes wait(S) and then P1 executes wait(Q). When P0 executes wait(Q), it must wait until P1 executes signal(Q). Similarly, when P1 executes wait(S), it must wait until P0 executes signal(S). Since these signal() operations cannot be executed, P0 and P1 are deadlocked.

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

Other forms of deadlock

We say that a set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set. The “events” with which we are mainly concerned here are the acquisition and release of resources such as mutex locks and semaphores.

- Starvation – indefinite blocking (A process may never be removed from the semaphore queue in which it is suspended)
- **Priority Inversion (it can occur only in systems with more than two priorities)** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process (Solved via priority-inheritance protocol)

Classical Problems of Synchronization

(1) the critical-section problem focused on how race conditions can occur when multiple concurrent processes share data. (2) examining several tools that address the critical-section problem by preventing race conditions from occurring ranged from low-level hardware solutions (compare-and-swap operation) to higher-level tools (from mutex locks to semaphores to monitors) (3) discussions on various challenges in designing applications that are free from race conditions, including liveness hazards such as deadlocks.

- Classical problems (in our solutions to the problems, we use semaphores for synchronization) used to test newly-proposed synchronization schemes:
  Bounded-Buffer Problem
  Readers and Writers Problem
  Dining-Philosophers Problem

Bounded-Buffer Problem I

An illustration of the power of synchronization primitives
In our problem, the producer and consumer processes share the following data structures
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
We assume that the pool consists of n buffers, each capable of holding one item.
The mutex binary semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1.
The empty and full semaphores count the number of empty and full buffers.
The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.

Bounded-Buffer Problem II: Producer

```
do {
      ...
       /* produce an item in next_produced */
      ...
   wait(empty);
   wait(mutex);
      ...
       /* add next produced to the buffer */
      ...
   signal(mutex);
   signal(full);
 } while (true);
```

Bounded-Buffer Problem III: Consumer

```
Do {
    wait(full);
    wait(mutex);
       ...
       /* remove an item from buffer to next_consumed */
       ...
    signal(mutex);
    signal(empty);
       ...
       /* consume the item in next consumed */
       ...
    } while (true);
```

Note the symmetry between the producer and the consumer. We can interpret this code as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

Readers-Writers Problem I

- A data set is shared among a number of concurrent processes
  Readers – only read the data set; they do not perform any updates
  Writers – can both read and write
- If two readers access the shared data simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may ensue.
- Problem – allow multiple readers to read at the same time
  Only one single writer can access the shared data at the same time (to ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared data set)
- Several variations of how readers and writers are considered – all involve some form of priorities

Readers-Writers Problem II

First readers–writers problem, requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting.
Second readers–writers problem requires that, once a writer is ready, that writer perform its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.
A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve.

Readers-Writers Problem: writer

In the solution to the first readers–writers problem, the reader processes share the following data structures:
Shared Data
Data set
Semaphore rw_mutex initialized to 1 and it is common to both reader and writer processes
Semaphore mutex initialized to 1 and is used to ensure mutual exclusion when the variable read_count is updated
Integer read_count initialized to 0 keeps track of how many processes are currently reading the object.

```
do {
      wait(rw_mutex);
        ...
      /* writing is performed */
        ...
   signal(rw_mutex);
} while (true);
```

Readers-Writers Problem: reader

Note that, if a writer is in the critical section and n readers are waiting, then one reader is queued on rw_mutex, and n-1 readers are queued on mutex. Also observe that, when a writer executes signal(rw_mutex), we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler.

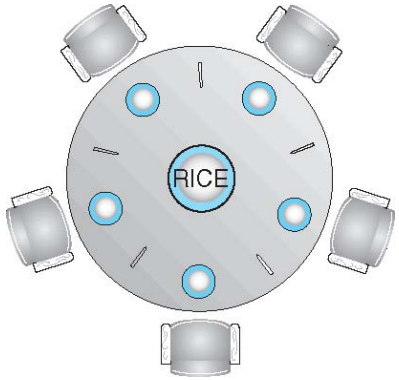
```
do {
        wait(mutex);
        read_count++;
        if (read_count == 1)
        wait(rw_mutex);
    signal(mutex);
        ...
        /* reading is performed */
        ...
    wait(mutex);
        read_count--;
        if (read_count == 0)
    signal(rw_mutex);
    signal(mutex);
} while (true);
```

Readers-Writers Problem III

The readers–writers problem and its solutions have been generalized to provide reader–writer locks on some systems. Acquiring a reader–writer lock requires specifying the mode of the lock: either read or write access. Reader–writer locks are most useful in the following situations:

- In applications where it is easy to identify which processes only read shared data and which processes only write shared data.
- In applications that have more readers than writers. This is because reader–writer locks generally require more overhead to establish than semaphores or mutual-exclusion locks. The increased concurrency of allowing multiple readers compensates for the overhead involved in setting up the reader–writer lock.

Dining-Philosophers Problem



- An example of a large class of concurrency-control problems (a representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner): Consider five philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl (Need both to eat, then release both when done)
- Shared data
  Bowl of rice (data set)
  Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem Algorithm: Philosopher i

One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore. She releases her chopsticks by executing the signal() operation on the appropriate semaphores. Thus, the shared data are semaphore chopstick[5]

```
do {
    wait (chopstick[i] );
     wait (chopStick[ (i + 1) % 5] );

                //  eat

     signal (chopstick[i] );
     signal (chopstick[ (i + 1) % 5] );

                 //  think

} while (TRUE);
```

Dining-Philosophers Problem Algorithm II

Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock. Suppose that all five philosophers become hungry at the same time and each grabs her left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section.
  - Use an asymmetric solution – an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

Problems with Semaphores

- Incorrect use of semaphore operations
  - signal (mutex) ... wait (mutex) -> several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement. This error may be discovered only if several processes are simultaneously active in their critical sections. Note that this situation may not always be reproducible.
  - wait (mutex) ... wait (mutex) -> the process will permanently block on the second call to wait(), as the semaphore is now unavailable.
  - Omitting of wait (mutex) or signal (mutex) (or both) -> either mutual exclusion is violated or the process will permanently block.
- Deadlock and starvation are possible when programmers use semaphores or mutex locks incorrectly to solve the critical-section problem -> One strategy for dealing with such errors is to incorporate simple synchronization tools as high-level language constructs.

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Abstract data type, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes

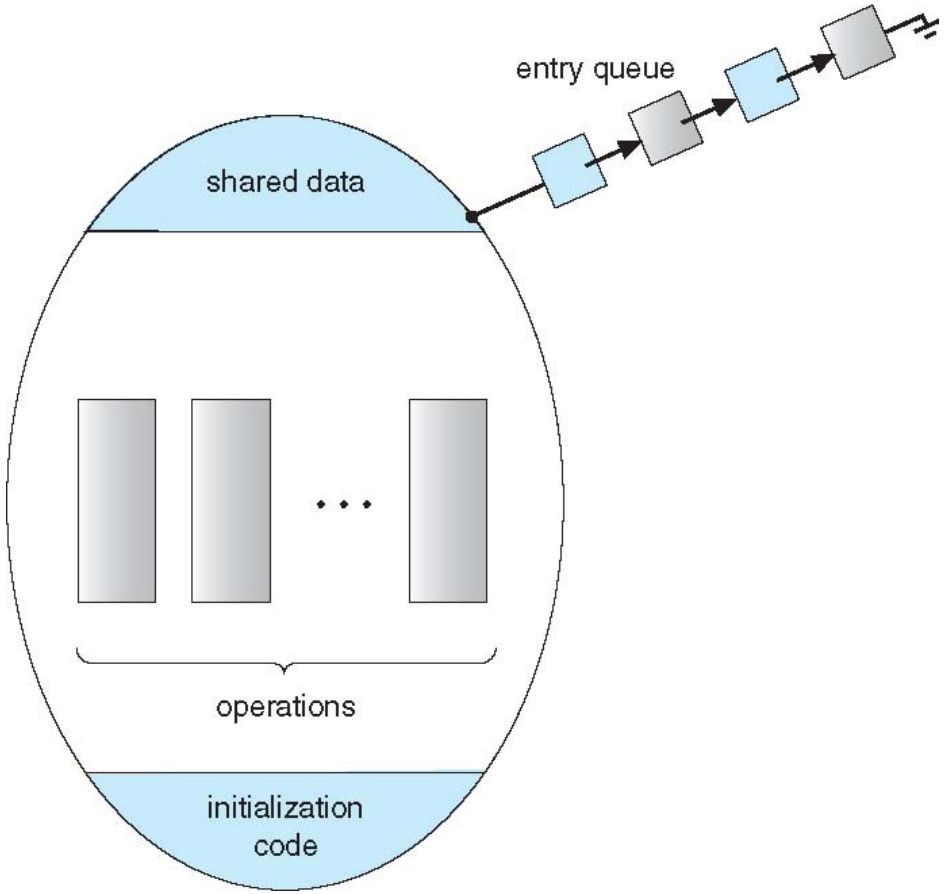
```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }

  procedure Pn (…) {……}

    Initialization code (…) { … }
  }
}
```

Schematic view of a Monitor

The monitor construct ensures that only one process at a time is active within the monitor. Consequently, the programmer does not need to code this synchronization constraint explicitly



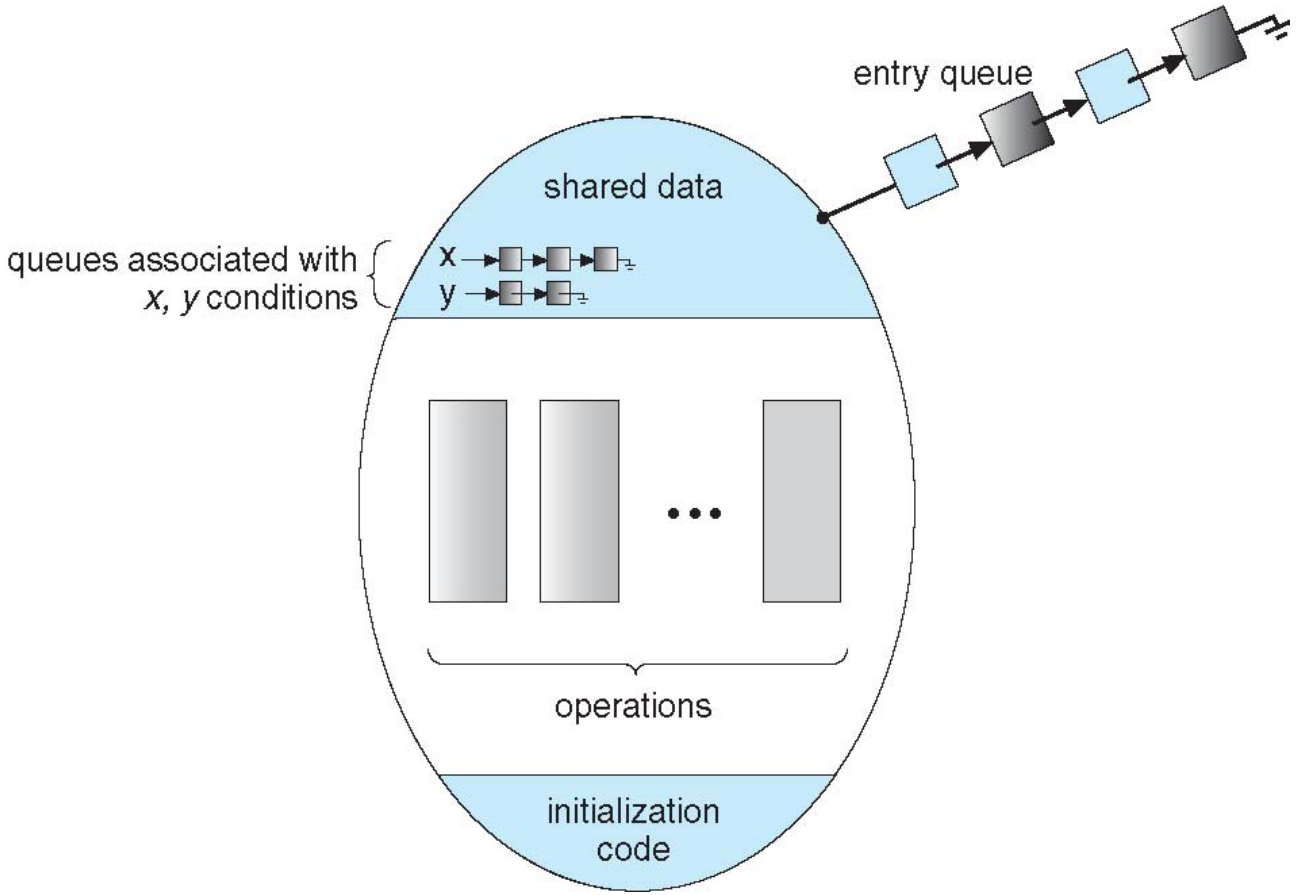
Condition Variables

The monitor construct, as defined so far, is not sufficiently powerful for modeling some synchronization schemes. For this purpose, we need to define additional synchronization mechanisms. These mechanisms are provided by the condition construct. A programmer who needs to write a tailor-made synchronization scheme can define one or more variables of type condition

- condition x, y;
- Two operations are allowed on a condition variable:
  - x.wait() – a process that invokes the operation is suspended until x.signal()
  - x.signal() – resumes one of processes (if any) that invoked x.wait()
    If no x.wait() on the variable, then it has no effect on the variable

Monitor with Condition Variables

The x.signal() operation resumes exactly one suspended process. If no process is suspended, then the signal() operation has no effect; that is, the state of x is the same as if the operation had never been executed

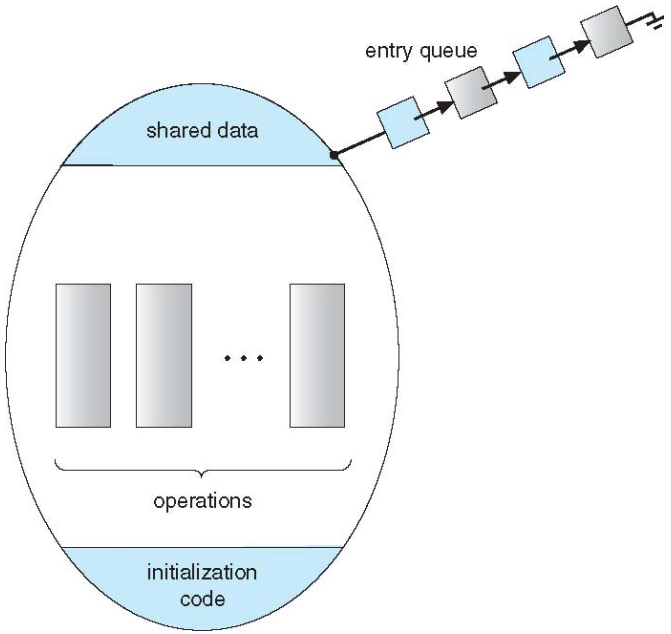


Condition Variables Choices

- If process P invokes x.signal(), and process Q is suspended in x.wait(), what should happen next?
  Both Q and P cannot execute in paralel. If Q is resumed, then P must wait
- Options include
  - Signal and wait – P waits until Q either leaves the monitor or it waits for another condition
  - Signal and continue – Q waits until P either leaves the monitor or it waits for another condition
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise
    P executing signal immediately leaves the monitor, Q is resumed

Monitor Solution to Dining Philosophers I

This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available. To code this solution, we need to distinguish among three states in which we may find a philosopher. For this purpose, we introduce the following data structure:
enum {THINKING, HUNGRY, EATING} state[5];



Monitor Solution to Dining Philosophers II

Philosopher i can set the variable state[i] = EATING only if her two neighbors are not eating:
(state[(i+4) % 5] != EATING) and (state[(i+1) %5] != EATING)
We also need to declare
condition self[5];
This allows philosopher i to delay herself when she is hungry but is unable to obtain the chopsticks she needs.

```
monitor DiningPhilosophers
{
   enum { THINKING; HUNGRY, EATING) state [5] ;
   condition self [5];

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

   void putdown (int i) {
          state[i] = THINKING;
                   // test left and right neighbors
           test((i + 4) % 5);
           test((i + 1) % 5);
}
```

Monitor Solution to Dining Philosophers III

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

    initialization_code() {
       for (int i = 0; i < 5; i++)
       state[i] = THINKING;
     }
}
```

Each philosopher i invokes the operations pickup() and putdown() in the following sequence:

```
DiningPhilosophers.pickup(i);
```

EAT

```
DiningPhilosophers.putdown(i);
```

No deadlock, but starvation is possible

Monitor Implementation Using Semaphores

- Variables
  semaphore mutex; // (initially = 1) for each monitor, a binary semaphore mutex is provided to ensure mutual exclusion
  semaphore next; // (initially = 0)
  int next_count = 0;
- Each procedure F will be replaced by

```
wait(mutex);
...
body of F;
...
if (next_count > 0)
   signal(next)
else
   signal(mutex);
```

- Mutual exclusion within a monitor is ensured

Monitor Implementation – Condition Variables

- For each condition variable x, we have:
  semaphore x_sem; // (initially = 0)
  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 II

Since a signaling process must wait until the resumed process either leaves or waits, an additional binary semaphore, next, is introduced, initialized to 0. The signaling processes can use next to suspend themselves. An integer variable next_count is also provided to count the number of processes suspended on next.

- The operation x.signal can be implemented as:

```
if (x_count > 0) {
        next_count++;
        signal(x_sem);
        wait(next);
        next_count--;
}
```

Semaphore vs Monitor

Semaphore	Monitor
It is an integer variable	It is an abstract data type.
The value of this integer variable tells about the number of shared resources that are available in the system.	It contains shared variables.
Mutual exclusion needs to be implemented explicitly	Mutual exclusion in monitors is automatic
It doesn't have condition variables.	It has condition variables.