ID1217 Concurrent Programming
Lecture 11

ROYAL INSTITUTE
OF TECHNOLOGY

Monitors

Vladimir Vlassov
KTH/ICT/EECS

ROYAL INSTITUTE OF TECHNOLOGY

Outline

- Origin, syntax and semantics of ***monitors*** and condition variables
- ***Programming with monitors*** illustrated with examples:
  - Bounded buffers: Basic condition synchronization
  - Readers and Writers: Broadcast signal
  - Shortest-job-next (SJN) allocation: Priority wait
  - Interval timer: Covering conditions
  - The sleeping barber: Rendezvous of processes
  - Disk scheduling: Program structures; Nested monitors
- ***Tutorial: Java Monitors. Concurrent Utilities in Java SDK*** – to be considered *in a separate lecture*

ROYAL INSTITUTE OF TECHNOLOGY

Short Review: Locks, Condition Variables, Semaphores

- ***Locks*** – general, but busy-waiting (spin locks, condition synchronization)
  - OS should provide time slicing.
- ***Condition variables*** – a synchronization (signaling) mechanism to block/resume processes holding locks.
  - FIFO queues of blocked processes.
  - Used for condition synchronization.
- ***Semaphores***:
  - Sufficiently general and can be used to solve any mutual exclusion or condition synchronization problem.
  - FIFO semaphore queues of delayed processes.
  - Main disadvantages: too low-level, error prone
- The next synch mechanism to be considered: ***Monitors*** (with condition variables)

ROYAL INSTITUTE OF TECHNOLOGY

Monitors

- A ***monitor*** is a programming language construct (a synchronized object) which encapsulates variables, access procedures and initialization code within an abstract data type (a class).
  - ***Monitor variables*** represent a state of an instance of the monitor;
  - ***Monitor procedures (a.k.a. operations or methods)*** are the only means to access (to inspect and to alter) the monitor variables;
    - ***Monitor operations are executed with mutual exclusion.***
  - Monitors were named and popularized by Tony Hoare in late 70s
    - Showed how to implement monitors with split binary semaphores
- Programming environments that provide monitors and cond variables:
  - Monitors in Java – objects with synchronized methods (and implicit condition variables);
    `Condition` – interface in `java.util.concurrent.locks`.
  - Explicit condition variables in the Pthreads C-library.

ROYAL INSTITUTE OF TECHNOLOGY

Monitor Definition and Use

- For simplicity, assume that a monitor is an object (a static class) declared as:

```
monitor monitorName {
    declarations of variables;      // monitor state
    initialization code;            // to initialize the monitor
    procedures;                     // a.k.a. methods, operations
}
```

  - The initialization code initializes monitor variables – executed once when the monitor is created and before any monitor procedure is called;
  - **Monitor invariant** is a predicate about a monitor state that is true when no call is active.
- A process calls a monitor procedure by executing:

```
call monitorName.procedureName(arguments)
```

ROYAL INSTITUTE OF TECHNOLOGY

Need for Condition Variables in Monitors

- Suppose, a process needs to alter some monitor variables when the monitor is in some desirable state:

  <await (desirable_monitor_state) alter_the_monitor_state; >

  - Busy-waiting in the monitor leads to deadlock because the process does not release the monitor lock;
  - Busy-waiting by repeatedly entering and exiting the monitor is not efficient – waste of CPU time.

- **Condition variables** provide a safe signaling mechanism for blocking/resuming processes in a monitor.
  - **Condition variable** is an opaque object that represents a queue of processes waiting to be resumed.
  - See also Lecture 4: Critical Sections. Lock. Condition Variables

ROYAL INSTITUTE OF TECHNOLOGY

Condition Variables in Monitors

- The monitor's condition variables may be declared, and hence used only within the monitor.
- Declaration: `cond cv;`
- Operations:
  - *The monitor entry lock is implicit* and does not need to be specified in the wait/signal operations on condition variables

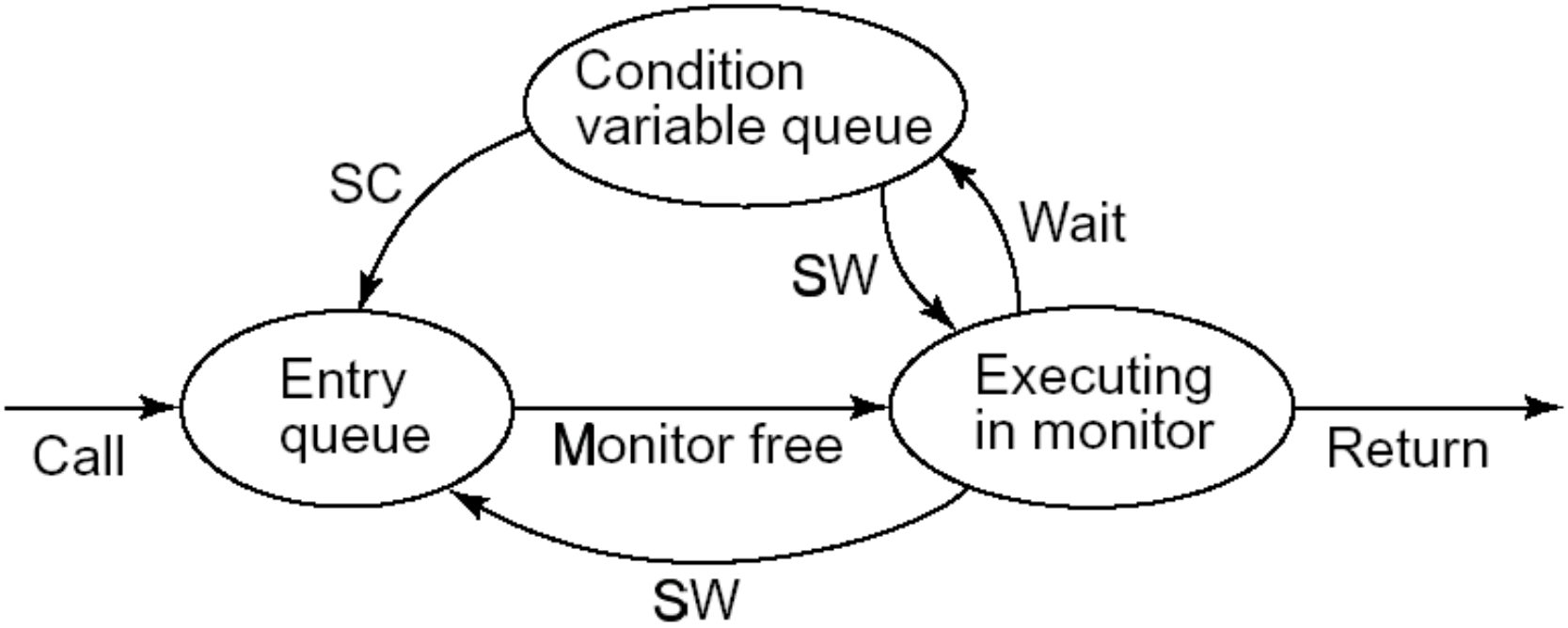
`wait(cv)`	Release the monitor lock and wait at end queue
`wait(cv, rank)`	Release the monitor lock and wait in order of increasing value of rank
`signal(cv)`	Awaken process at front of queue then continue
`signal_all(cv)`	Awaken all processes on queue then continue
`empty(cv)`	True if queue is empty; false otherwise
`minkrank(cv)`	Value of rank of proc at front of wait queue

ROYAL INSTITUTE OF TECHNOLOGY

Signaling Disciplines for Condition Variables

- **Signal and Continue (SC)** – the signaling process continues, the resumed process reacquires the lock
  - The default discipline
- **Signal and Wait (SW)** – the signaling process passes the lock to the resumed process and reacquires the lock
- **Signal and Urgent Wait (SUW)** – as SW but the signaling process is placed to the head of the lock queue

State Diagram for Synchronization in Monitors

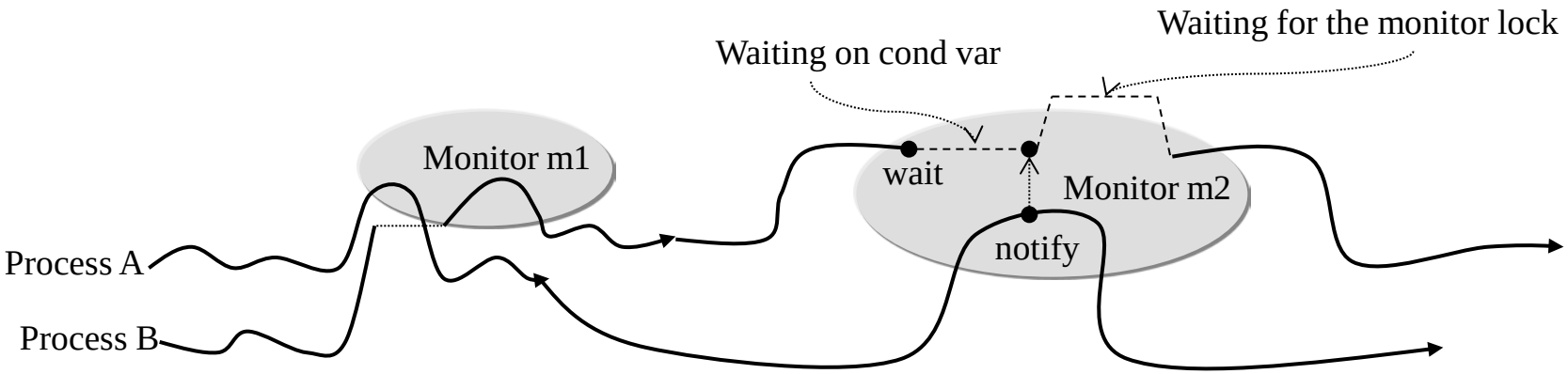


SC – Signal and Continue

SW – Signal and Wait

Concurrent Programming with Monitors and Condition Variables

- A concurrent program with monitors is formed of ***processes*** and ***shared monitors*** (with condition variables).
- The processes ***communicate via*** monitors by calling monitor methods
  - Mutual exclusion: At most one process can execute in a monitor at a time
  - Condition synchronization: A process can wait in the monitor on a condition variable – the waiting process releases the monitor lock



Monitor Implementation of a Semaphore

```
monitor Semaphore {
  int s = 0;   ## s >= 0
  cond pos;    #  signaled when s > 0
  procedure Psem() {
    while (s == 0) wait(pos);
    s = s-1;
  }
  procedure Vsem() {
    s = s+1;
    signal(pos);
  }
}
```

```
monitor FIFOsemaphore {
  int s = 0;   ## s >= 0
  cond pos;    #  signaled when s > 0
  procedure Psem() {
    if (s == 0)
      wait(pos);
    else
      s = s-1;
  }
  procedure Vsem() {
    if (empty(pos))
      s = s+1;
    else
      signal(pos);
  }
}
```

- This code above works for both the SC and SW signaling disciplines.

ROYAL INSTITUTE OF TECHNOLOGY

Use of Monitors

- A monitor as a class (a synchronized shared object) can be used for:
  - atomic operations on shared variables encapsulated in the monitor;
  - controlling access to shared resources “outside” the monitor.
- Examples considered:
  - (each example also illustrate some synchronization technique)
    - Bounded buffers: Basic condition synchronization.
    - Readers and Writers: Broadcast signal.
    - Shortest-job-next (SJN) allocation: Priority wait.
    - Interval timer: Covering conditions.
    - The sleeping barber: Rendezvous (in a client-server application)
    - Disk scheduling: Put-altogether example.

ROYAL INSTITUTE OF TECHNOLOGY

Bounded Buffers

- The example illustrates basic condition synchronization in monitors.
- ***Producer*** and ***Consumer*** communicate via a ***shared n-slot buffer*** to be implemented as a ***monitor***.
- Monitor variables (state):
  - `buf[n]`, `front`, `rare`, `count` (the number of full slots ≤ **n**)
  - `cond not_full` – to delay Producer until the buffer becomes empty;
  - `cond not_empty` – to delay Consumer until the buffer becomes full.
- Monitor procedures:
  - `deposit(typeT)` – called by Producer;
  - `fetch(*typeT)` – called by Consumer.

Monitor Implementation of a Bounded Buffer

```
monitor Bounded_Buffer {

  typeT buf[n];      # an array of some type T
  int front = 0,     # index of first full slot
      rear = 0;      # index of first empty slot
      count = 0;     # number of full slots
  ## rear == (front + count) % n
  cond not_full,     # signaled when count < n
       not_empty;    # signaled when count > 0

  procedure deposit(typeT data) {
    while (count == n) wait(not_full);
    buf[rear] = data; rear = (rear+1) % n; count++;
    signal(not_empty);
  }

  procedure fetch(typeT &result) {
    while (count == 0) wait(not_empty);
    result = buf[front]; front = (front+1) % n; count--;
    signal(not_full);
  }

}
```

- Usage:
  - In Producer:

```
Bounded_Buffer.deposit(a[i]);
```

  - In Consumer:

```
Bounded_Buffer.fetch(&a[i]);
```

ROYAL INSTITUTE OF TECHNOLOGY

A Quick Look to Monitors in Java

- A ***Java monitor*** is an object of a class with **`synchronized`** methods that can be invoked by one thread at a time.
  - A class may have synchronized and non-synchronized methods.
- Each monitor object has ***a monitor lock (a.k.a. an entry lock)*** and one condition variable (a.k.a. "***wait-set***")
  - No declaration of the condition variable and the lock is required.
- Operations on the condition variable:
  - **`wait()`**, **`notify()`** and **`notifyAll()`** in scope of a **`synchronized`** method;
  - No priority wait;
  - Signal-and-Continue policy of **`notify()`** and **`notifyAll()`**

The Bounded Buffer Class in Java

```
public class Bounded_Buffer {
   private Object[] buf;
   private int n, count = 0, front = 0, rear = 0;
   public Bounded_Buffer(int n) {
      this.n = n;
      buf = new Object[n];
   }
   public synchronized void deposit(Object obj) {
      while (count == n)
         try { wait(); }
         catch (InterruptedException e) { }
      buf[rear] = obj; rear = (rear + 1) % n; count++;
      notifyAll();
   }
   public synchronized Object fetch() {
      while (count == 0)
         try { wait(); }
         catch (InterruptedException e) { }
      Object obj = but[front]; buf[front] = null;
      front = (front + 1) % n; count--;
      notifyAll();
      return obj;
   }
}
```

- Methods `deposit` and `fetch` are synchronized (to be executed with mutual exclusion)
- Condition variable is implicit – methods **`wait`**, **`notify`**, **`notifyAll`**
- Java threads and monitors will be consider later in the course.

ROYAL INSTITUTE OF TECHNOLOGY

Readers/Writers using a Monitor

- The example illustrates use of **`signall_all`** – broadcast signal
- ***Readers*** read a database; ***Writers*** write the database.
- ***Selective mutual exclusion***:
  - Only concurrent reads are allowed;
  - Write operation is executed in mutual exclusion with reads and other writes.
  - See also Readers/Writers with semaphores in Lecture 10.
- Over-constrained solution: Implement the shared database as a monitor with read and write operations.
  - Guarantees exclusive access to the database, but precludes concurrent reads.
- Idea: ***Develop a monitor that arbitrates access to the database***.
  - Use the monitor as an access controller that grants access permission to processes.

The Access Control Monitor

- Arbitrates access to the database.
- State (monitor variables):
  - **`nr`** – number of active readers
  - **`nw`** – number of active writers
  - **`oktoread`** – cond var: queue of readers waiting for access permission
  - **`oktowrite`** – cond var: queue of writers waiting for access permission
  - Monitor invariant: **`RW: (nr == 0 ∨ nw == 0) ∧ nw ≤ 1`**
- Operations (monitor procedures):
  - **`request_read:`** grant access to the reader if there is no active writer otherwise wait;
  - **`request_write:`** grant access to the writer if there are no active writers or readers, otherwise wait;
  - **`release_read:`** the last active reader signals a waiting writer if any;
  - **`release_write:`** signal all waiting processes;

Readers/Writers Solution using a Monitor

```
monitor RW_Controller {

  int nr = 0, nw = 0;  ## (nr == 0 ∨ nw == 0) ∧ nw <= 1
  cond oktoread;     #  signaled when nw == 0
  cond oktowrite;    #  signaled when nr == 0 and nw == 0

  procedure request_read() {
    while (nw > 0) wait(oktoread);
    nr = nr + 1;
  }

  procedure release_read() {
  nr = nr - 1;
  if (nr == 0) signal(oktowrite);  # awaken one writer
  }

  procedure request_write() {
    while (nr > 0 || nw > 0) wait(oktowrite);
    nw = nw + 1;
  }

  procedure release_write() {
    nw = nw - 1;
    signal(oktowrite);       # awaken one writer and
    signal_all(oktoread);    # all readers
  }

}
```

- Usage:
  - In Reader:

```
RW_Controller.request_read();
read the database;
RW_Controller.release_read();
```

  - In Writer:

```
RW_Controller.request_write();
write the database;
RW_Controller.release_write();
```

ROYAL INSTITUTE OF TECHNOLOGY

Shortest Job Next (SJN) Allocation: Priority Wait

- SJN (or SJFirst) is an allocation policy which favors processes with the shortest estimated time the process will occupy a shared resource.
  - See also SJN allocation with semaphores in Lecture 10.
- The example illustrates a usage of ***priority wait***:
  - A condition variable by default is a FIFO queue
    - `wait(c)` – wait in FIFO order
  - To wait in order of increasing value of rank, call
    - `wait(c, rank)`
- A monitor with priority wait can be used as a controller that allocates the resource in some priority order.

The SJN Monitor

- A SJN scheduler can be develop as a monitor that allocates a shared resource to processes in the SJF order.
- State (monitor variables):
  - **`boolean free`** – indicates whether the resource is free;
  - **`cond turn`** – a queue of processes waiting for the resource.
  - Monitor invariant:
    `turn` ordered by **`time`** $\wedge$ (**`free`** $\Rightarrow$ **`turn`** is empty)
- Procedures:
  - **`request(int time)`**: allocate the resource if free, otherwise wait in increasing order of time;
  - **`release:`** signal the waiter in the front of the queue if any, otherwise release the resource;

SJN Allocation using a Monitor

```
monitor Shortest_Job_Next {

  bool free = true;  ## Invariant SJN:  see text
  cond turn;         #  signaled when resource available

  procedure request(int time) {
    if (free)
      free = false;
    else
      wait(turn, time);
  }

  procedure release() {
    if (empty(turn))
      free = true
    else
      signal(turn);
  }
}
```

- Usage of the monitor:

```
Shortest_Job_Next.request(mytime);
use the resource;
Shortest_Job_Next.release();
```

ROYAL INSTITUTE OF TECHNOLOGY

Interval Timer. Covering Conditions

- Design an interval timer that makes its possible for processes to pause (to sleep) for a specified amount of time.
  - Solution idea: develop the interval timer as a monitor, delay a process in the monitor for the specified amount of ticks:

```
delay: < await (wake_time ≤ time_of_day); >
```

- The example illustrates the ***covering condition technique***

ROYAL INSTITUTE OF TECHNOLOGY

The Covering Condition Technique

- A ***covering condition*** "covers" (disjunction of) the actual conditions for which the processes are waiting:

  covering cond = cond1 ∨ cond2 ∨ …

  - The covering condition is weaker than each actual condition it covers.
- All waiters are signaled on the covering condition to recheck their actual conditions.
  - The signal is false if the actual condition is not met.
- Tradeoff: the cost of ***false signals*** (***false alarms***) with one covering condition for the cost of "private" condition variables (one for each condition to wait on).

The Interval Timer Monitor

- Monitor variables:
  - **`int tod`** – current time-of-day (in ticks)
  - **`cond check`** – queue of delayed processes
  - Invariant: ***CLOCK:*** **`tod`** $\geq 0 \wedge$ (**`tod`** increases monotonically by 1)
- Monitor procedures:
  - **`delay(interval)`** – delay a calling process for a number of ticks specified by **`interval`**
  - **`tick()`** – increment **`tod`** – the procedure is called with a fixed rate by a clock "process" (e.g. a HW-timer interrupt handler).
- Usage:

```
Timer.delay(numerOfTicks);
```

Interval Timer with a Covering Condition

- The covering condition is **“one tick has been passed”**
- An actual condition: **my_wake_time <= time_of_day**
- On each tick, all delayed processes are signaled to check if it is the time for a proc to exit the monitor, and if it is not (false alarm), the proc is delayed until the next tick.

```
monitor Timer {

  int tod = 0;    ## invariant CLOCK -- see text
  cond check;     #  signaled when tod has increased

  procedure delay(int interval) {
    int wake_time;
    wake_time = tod + interval;
    while (wake_time > tod) wait(check);
  }

  procedure tick() {
    tod = tod + 1;
    signal_all(check);
  }

}
```

- Usage: `Timer.delay(numerOfTicks);`

Interval Timer with Priority Wait

- Wait on the **`check`** queue in order of increasing value of wake-up time.
- Call **`minrank`** in **`tick`** to check if it is the time to signal a head process.
- Solution with priority wait avoids false alarms

```
monitor Timer {
  int tod = 0;    ## invariant CLOCK -- see text
  cond check;     #  signaled when minrank(check)<=tod
  procedure delay(int interval) {
    int wake_time;
    wake_time = tod + interval;
    if (wake_time > tod) wait(check, wake_time);
  }
  procedure tick() {
    tod = tod+1;
    while (!empty(check) && minrank(check) <= tod)
      signal(check);
  }
}
```

- Usage: **`Timer.delay(numerOfTicks);`**

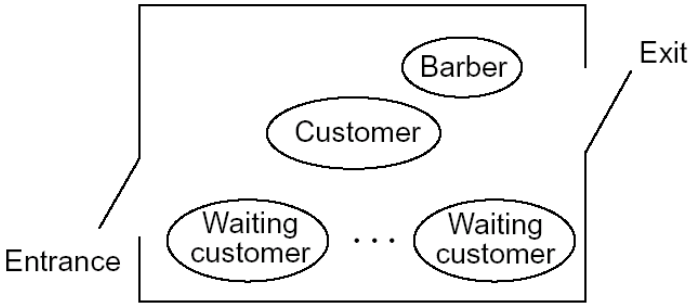
ROYAL INSTITUTE OF TECHNOLOGY

The Sleeping Barber: Rendezvous

- Example illustrates use of monitors and condition variables in ***a client-server application***: Servicing multiple clients by a server.
- Point-to-point synchronization (**rendezvous**) between processes via a shared monitor using condition variables for signaling:
  - ***A barber (a server process)*** and ***a customer (a client process)*** need to rendezvous – the barber has to wait for a customer to arrive, and a customer has to wait for the barber to be available;

The Sleeping Barber Problem

- ***The Barber is a server*** that repeatedly provides a service (haircut) to customers .
  - Waits for a customer;
  - Gives a haircut;
  - Signals the customer to leave;
  - Waits until the customer leaves.



- ***A Costumer is a client*** that requests the service (haircut):
  - Enters the barber shop;
  - Waits until the barber is available;
  - Waits until the haircut is finished;
  - Leaves the shop.
- **The problem: To develop a monitor ("Barber shop") that allows the Barber (the server) to service Customers (clients)**

The Barber Shop Monitor

- Procedures:
  - **`get_haircut`** – called by a Customer to request a haircut.
  - **`get_next_customer`** – called by the Barber to get a customer.
  - **`finished_cut`** – called by the Barber when it finishes the service.
- Usage of the monitor:
  - The Barber process (server):

```
while (true) {
  Barber_Shop.get_next_customer();
  Service a customer;
  Barber_Shop.finish_cut();
}
```

  - A Customer process (client):

```
Barber_Shop.get_haircut();
```

ROYAL INSTITUTE OF TECHNOLOGY

Variables of the Monitor

- Integer flags (used for conditions):
  - **`barber`** – the barber is free (the server can accept a request);
  - **`chair`** – the chair is occupied (there is a request);
  - **`open`** – the door is open (the request is processed and the reply is ready), when **`open`** is 0 – the customer has left the shop.
- Condition variables (delay points):
  - Used for point-to-point synchronization (rendezvous).
  - The barber is waiting at:
    - **`chair_occupied`** – to wait for a customer: <await (**`chair`** > 0);>
    - **`customer_left`** – to wait for the customer to leave: <await (**`open`** == 0);>
  - A customer is waiting at:
    - **`barber_available`** – to wait for the barber: < await (**`barber`** > 0);>
    - **`door_open`** – to wait for haircut to finish: <await (**`open`**> 0);>

The Barber Shop Monitor Outline

A Customer process (client):

```
Barber_Shop.get_haircut();
```

The Barber process (server):

```
while (true) {
 Barber_Shop.get_next_customer();
 Service a customer;
 Barber_Shop.finish_cut();
}
```

- Has multiple delay points.
- Uses point-to-point (rendezvous) synchronization.
- Q: Can one use semaphores instead of flags and condition variables?

```
monitor Barber_Shop {
  int barber = 0, chair = 0, open = 0;
  cond barber_available;     # signaled when barber > 0
  cond chair_occupied;       # signaled when chair > 0
  cond door_open;            # signaled when open > 0
  cond customer_left;        # signaled when open == 0
  procedure get_haircut() {
    while (barber == 0) wait(barber_available);
    barber = barber - 1;
    chair = chair + 1; signal(chair_occupied);
    while (open == 0) wait(door_open);
    open = open - 1; signal(customer_left);
  }
  procedure get_next_customer() {
    barber = barber + 1; signal(barber_available);
    while (chair == 0) wait(chair_occupied);
    chair = chair - 1;
  }
  procedure finished_cut() {
    open = open + 1; signal(door_open);
    while (open > 0) wait(customer_left);
  }
}
```

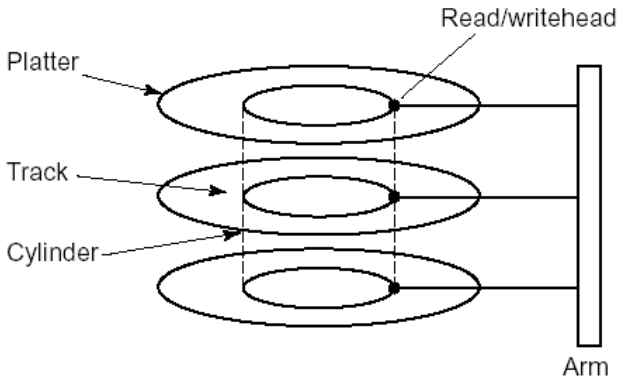
Solution of the Sleeping Barber Problem Using Semaphores

- In this solution the Barber Shop is implemented as a set of un-synchronized procedures rather than a monitor

```
sem barber = 0, chair = 0, open = 0, left = 0;
  procedure get_haircut() {
     P(barber);
     V(chair);
     P(open);
     V(left);
  }
  procedure get_next_customer() {
     V(barber);
     P(chair);
  }
  procedure finished_cut() {
     V(open);
     P(left);
  }
```

Example: Disk Scheduling Problem

- Illustrates a larger problem solved with monitors and condition variables
- ***A moving head disk*** used to store files:
  - Disk – several platters on a center spindle rotated with a fixed rate
  - Recording track – a concentric circle on a platter
  - Cylinder – set of tracks in the same relative position on platters
  - One read/write head for each platter
    - R/w heads on a single arm moved in and out to place read/write heads at any cylinder



ROYAL INSTITUTE OF TECHNOLOGY

Accessing Moving Head Disk

- I/O instructions to transfer data between the main memory and the disk :
  - Opcode (read or write);
  - A physical disk address;
  - The address of a data buffer;
  - The number to bytes to be transferred to/from the disk.
- Data access:
  - Position a r/w head over a track
  - Wait for the platter to rotate until the desired data passed by the head
  - Read/write the track

ROYAL INSTITUTE OF TECHNOLOGY

Disk Scheduling Problem

- Assume multiple proc issue i/o disk instructions. Instructions (requests) are queued to be scheduled for execution.
- ***The problem***: Develop an efficient scheduler for scheduling access to a moving head disk.
- Should reduce the average disk access time that includes
  - ***Seek time*** to move the r/w head to the given cylinder – depends on the distance between the current position and the required cylinder;
  - ***Rotational delay*** – much shorter than the seek time;
  - ***Data transmission time*** – depends on the amount of data.
- Which time to reduce? – The most effective way is to reduce seek time.

ROYAL INSTITUTE OF TECHNOLOGY

Scheduling Strategies

- Shortest seek time (SST) strategy:
  - The request with the shortest seek time is scheduled next.
  - Minimizes seek time, but unfair – may cause starvation.
  - Used in UNIX.
- SCAN (LOOK, the ***Elevator algorithm***)
  - The request with the shortest seek time in the direction of the last move is scheduled next.
    - Service all requests in one direction, then change the direction.
  - Large variance in expected waiting time.
- CSCAN (CLOOK) – circular SCAN
  - Like SCAN but requests are serviced only in one direction, e.g. from outermost to innermost.
  - Reduces invariant in expected waiting time.

ROYAL INSTITUTE OF TECHNOLOGY

Solutions Considered

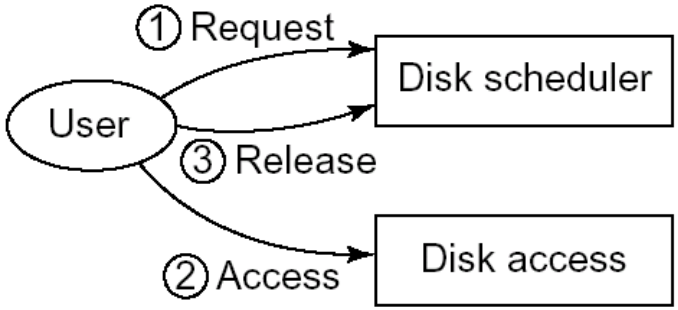
Assume the CSCAN strategy.

1. A separate scheduler as a resource controller
   - A monitor that controls access to the disk (similar to the Readers/Writers monitor)
2. A scheduler as an intermediary
   - A monitor between the requestor and the disk driver (similar to the "Sleeping Barber" monitor)
   - Forwards access requests to the driver in the desired order;
   - Forwards results back to the requesting process.
3. Using a nested monitor
   - Two monitors: one for scheduling, and one for access to the disk
     - The former calls the latter – nested calls

1. Using a Separate Monitor

- The monitor (Disk Scheduler) serves as an access controller
  - Usage :

```
Disk_Scheduler.request(cyl);
Access the disk
Disk_Scheduler.release();
```



  - The proc may be delayed in the monitor waiting for the access permission.
  - Simple structure, but all processes must follow the same protocol of three steps: request; access; release.

Separate Disk Scheduler Monitor

- Monitor operations: **`request(cyl)`**, **`release()`**
- Monitor variables (state):
  - **`int position`** – current head position (<0 – not accessed)
  - **`cond scan[0:1]`** – two queues of pending requests (for the current scan and for the next scan)
  - **`c`** – points to the queue of requests to be serviced in the current scan (ahead of the r/w head)
  - **`n`** –points to the queue of requests to be serviced in the next scan (behind of the r/w head)
- Monitor invariant:

  DISK: **`scan[c]`** and **`scan[n]`** are ordered ∧

  all elements in **`scan[c]`** > **`position`** ∧

  all elements in **`scan[n]`** ≤ **`position`** ∧

  (**`position`** == -1) ⇒ (both **`scan[c]`** and **`scan[n]`** are empty)

Separate Disk Scheduler Monitor

Usage :

```
Disk_Scheduler.request(cyl);
```

Access the disk

```
Disk_Scheduler.release();
```

```
monitor Disk_Scheduler {  ## Invariant DISK
  int position = -1, c = 0, n = 1;
  cond scan[2];   # scan[c] signaled when disk released
  procedure request(int cyl) {
    if (position == -1)  # disk is free, so return
      position = cyl;
    elseif (position != -1 && cyl > position)
      wait(scan[c],cyl);
    else
      wait(scan[n],cyl);
  }
  procedure release() {
    int temp;
    if (!empty(scan[c]))
      position = minrank(scan[c]);
    elseif (empty(scan[c]) && !empty(scan[n])) {
      temp = c; c = n; n = temp;        # swap c and n
      position = minrank(scan[c]);
      }
    else
      position = -1;
    signal(scan[c]);
  }
}
```

2. A Scheduler as an Intermediary

- A disk interface monitor between a client and a disk driver

  - Similar to a barber shop, but more complex
  - Simpler interface than in 1 – only one call by a client:

```
Disk_Interface.use_disk(cyl, op, req and res pointers)
```

  - The disk driver – a server proc that gets a request from the disk interface monitor:

```
Disk_Interface.get_next_request(&request);
Process request;
Disk_interface.finished_transfer(status);
```

Outline of Disk Interface Monitor

Usage:

– Client:

```
Disk_Interface.use_disk(cyl, op, req
    and res pointers)
```

– The disk driver:

```
Disk_Interface.get_next_request(
    &request);
```

Process request;

```
Disk_interface.finished_transfer(
    status)
```

```
monitor Disk_Interface
   permanent variables for status, scheduling, and data transfer
   procedure use_disk(int cyl, transfer and result parameters) {
      wait for turn to use driver
      store transfer parameters in permanent variables
      wait for transfer to be completed
      retrieve results from permanent variables
   }
   procedure get_next_request(someType &results) {
      select next request
      wait for transfer parameters to be stored
      set results to transfer parameters
   }
   procedure finished_transfer(someType results) {
      store results in permanent variables
      wait for results to be retrieved by client
   }
}
```

ROYAL INSTITUTE OF TECHNOLOGY

Disk Interface Monitor

- Queuing and signaling can be implemented easily with semaphores (like the “Sleeping Barber” example)

```
monitor Disk_Interface {
  int position = -2, c = 0, n = 1, args = 0, results = 0;
  cond scan[2];
  cond args_stored, results_stored, results_retrieved;
  argType arg_area; resultType result_area;
  procedure use_disk(int cyl; argType transfer_params;
                     resultType &result_params) {
    if (position == -1)
      position = cyl;
    elseif (position != -1 and cyl > position)
      wait(scan[c],cyl);
    else
      wait(scan[n],cyl);
    arg_area = transfer_params;
    args = args+1; signal(args_stored);
    while (results == 0) wait(results_stored);
    result_params = result_area;
    results = results-1; signal(results_retrieved);
  }
  procedure get_next_request(argType &transfer_params) {
    int temp;
    if (!empty(scan[c]))
      position = minrank(scan[c]);
    elseif (empty(scan[c]) && !empty(scan[n])) {
      temp = c; c = n; n = temp;      # swap c and n
      position = minrank(scan[c]);
      }
    else
      position = -1;
    signal(scan[c]);
    while (args == 0) wait(args_stored);
    transfer_params = arg_area; args = args-1;
  }
  procedure finished_transfer(resultType result_vals) {
    result_area := result_vals; results = results+1;
    signal(results_stored);
    while (results > 0) wait(results_retrieved);
  }
}
```

3. Using a Nested Monitor

- This solution uses 2 monitors:
  - One for scheduling (see Disk_Scheduler in slide 42),
  - One for access to the disk.


```
monitor Disk_Access {
   permanent variables as in Disk_Scheduler;
   procedure doIO(int cyl; transfer and result arguments) {
      actions of Disk_Scheduler.request;
      call Disk_Transfer.read or Disk_Transfer.write;
      actions of Disk_Scheduler.release;
   }
}
```

  - **`Disk_Access`** calls **`Disk_Transfer`** – this is a ***nested call***.

ROYAL INSTITUTE OF TECHNOLOGY

Nested Monitor Calls

- Nesting of monitor calls:
  - A process executing a procedure in one monitor calls a procedure in another monitor and hence temporarily leaves the first monitor
- ***Closed call*** – monitor exclusion is retained during the nested call – process holds two locks
- ***Open call*** – monitor exclusion is released during the nested call

- In the disc scheduling example:
  - The `Disk_Transfer` call is nested in `Disk_Access` and it must be open, otherwise: only one process uses both monitors at a time.

ROYAL INSTITUTE OF TECHNOLOGY

Exercise 1

- Consider the following monitor, which is proposed as a solution to the shortest-job-next (SJN) allocation problem. Client processes call **`request`** and then **`release`**. The resource can be used by at most one client at a time. When there are two or more competing requests, the one with the minimum value for argument time is to be serviced next.

```
monitor SJN {
    bool free = true;
    cond turn;
    procedure request(int time) {
      if (not free) wait(turn, time);
      free = false;
    }
    procedure release() {
      free = true;
      signal(turn);
    }
  }
```

- Does the above monitor work correctly for the SC discipline? Explain why or why not.
- Does the above monitor work correctly for the SW discipline? Explain why or why not.

ROYAL INSTITUTE OF TECHNOLOGY

Exercise 2

- Develop a monitor that has one operation:
  `void exchange(int* variable)`
- The procedure is called by two processes to exchange the values of their private variables (actual parameters). After two processes have called exchange, the monitor swaps the values of the variables which the arguments point to, and returns the control to the processes.
- The monitor should be reusable in the sense that it should exchange values of the variables of the first pair of callers, then the next pair of callers, and so on. You may use either the signal-and-continue (SC) or the signal-and-wait (SW) disciplines, but state which one you are using.