Part III
Synchronization
Monitors

You cannot build (or understand) a modern operating system unless you know the principles of concurrent programming.

Per Brinch Hansen
What Will Be Covered?

- Some historical remarks
- Monitor Basics
- What is a condition variable?
- Condition variable \textit{wait} and \textit{signal}
- Two Types of monitors: Hoare and Mesa
- Examples
- Hoare type vs. Mesa type and Semaphores vs. Monitors
- \textbf{ThreadMentor} Monitor Programming
- \textbf{ThreadMentor} Monitor Visualization
Some Historical Remarks: 1/2

- The concept of a monitor was invented by Per Brinch Hansen in early 1970s.
- Per Brinch Hansen used the concept of class in Simula 67 and defined a shared class as the beginning of today’s monitor.
- Hansen is also considered as a pioneer of concurrent programming. His *Concurrent Pascal* language used monitors for synchronization.

Per Brinch Hansen, *Operating System Principles*, Prentice-Hall, 1973 (Section 7.2)
Hansen’s landmark book *Operating System Principles* (Section 7.2) in 1973

Hansen’s another landmark book *The Architecture of Concurrent Programs* (1977) in which the concept of monitor and his Concurrent Pascal were clearly defined and discussed.

Hoare’s work on refining the concept of monitor was published in 1974.
What Is a Monitor? - Basics

- Monitor is a highly structured programming language construct. It consists of
  - private variables and private procedures that can only be used within a monitor.
  - constructors that initialize the monitor.
  - A number of (public) monitor procedures that are available to the users.
- Note that monitors have no public data.
- A monitor is a mini-OS with monitor procedures as system calls.
Monitor: Mutual Exclusion 1/2

- No more than one thread can be executing in a monitor. Thus, mutual exclusion is automatically guaranteed in a monitor.

- When a thread calls a monitor procedure and enters the monitor successfully, it is the only thread executing in the monitor.

- When a thread calls a monitor procedure and the monitor has a thread executing, the caller is blocked outside of the monitor.
Monitor: Mutual Exclusion 2/2

- If there is a thread executing in a monitor, any thread that calls a monitor procedure is blocked outside of the monitor.
- When the monitor has no executing thread, one waiting thread will be let in.
Monitor: Syntax

```java
monitor Monitor-Name
{
    local variable declarations;

    Procedure1(…)
    { // statements };
    Procedure2(…)
    { // statements }; // other procedures
    { // initialization

    }
}
```

- All variables are **private**. Why? Exercise!
- **Monitor procedures are public**; however, some procedures may be private so that they can only be used within a monitor.
- **Initialization procedures** (i.e., **constructors**) execute only once when the monitor is created.
Monitor: A Very Simple Example

```c
monitor IncDec
{
    int count;
    void Increase(void)
    { count++; }
    void Decrease(void)
    { count--; }
    int GetData(void)
    { return count; }
    { count = 0; }
}
```

thread Increment
while (1) {
    // do something
    IncDec.Increase();
    cout << IncDec.GetData();
    // do something
}

Is the printed value the one just updated?

initialization
Condition Variables

- Mutual exclusion is an easy task with monitors.
- While a thread is executing in a monitor, it may have to wait until an event occurs.
- Each programmer-defined event is conceptually represented by a condition variable.
- A condition variable, or a condition, has a private waiting list, and two public methods: signal and wait.
- Note that a condition variable has no value and cannot be modified.
**Condition wait**

- Let `cv` be a condition variable. The use of methods `signal` and `wait` on `cv` are
  `cv.signal()` and `cv.wait()`.

- Condition wait and condition signal can only be used **in a monitor**.

- A thread that executes a condition wait **blocks immediately** and is put into the waiting list of that condition variable. The monitor becomes “empty” (i.e., no executing thread inside).

- This means that this thread is waiting for the indicated event to occur.
Condition signal

- Condition signal is used to indicate an event has occurred.
- If there are threads waiting on the signaled condition variable, one of them will be released.
- If there is no waiting thread waiting on the signaled condition variable, this signal is lost as if it never happens.
- Consider the released thread (from the signaled condition) and the thread that signals. There are two threads executing in the monitor, and mutual exclusion is violated! Something has to be done to fix this problem.
Two Types of Monitors

- After a signal, the released thread and the signaling thread may be executing in the monitor.
- There are two approaches to address this issue:
  - **Hoare Type** (proposed by C. A. R. Hoare)\(^1\): The released thread takes the monitor and the signaling thread waits somewhere.
  - **Mesa Type** (proposed by Lampson and Redell)\(^2\): The released thread waits somewhere and the signaling thread continues to use the monitor. This is also used in Java.

---

What Do You Mean by “Waiting Somewhere”? 

- The signaling thread (Hoare type) or the released thread (Mesa type) must wait somewhere.
- You could consider there is a waiting bench for these threads to wait.
- Hence, each thread that involves in a monitor call may be in one of the four states:
  - **Active**: The running one.
  - **Entering**: Those blocked by the monitor.
  - **Waiting**: Those waiting on a condition variable.
  - **Inactive**: Those waiting on the waiting bench.
Threads blocked due to signal/wait are in the re-entry list (i.e., waiting bench).

When the monitor is free, a thread is released from either entry or re-entry.

Not all systems distinguish entry and re-entry threads.
What Is the Major Difference?

Condition UntilHappen;

// Hoare Type
if (!event)
    UntilHappen.wait();

// Mesa Type
while (!event)
    UntilHappen.wait();

With Hoare type, once a signal arrives, the signaler yields the monitor to the released thread and the condition is not changed. Thus, an if is sufficient.

With Mesa type, the released thread may be waiting for a while before it runs. During this period, other threads may be in the monitor and change the condition. It is better to check the condition again with a while!

Unless stated otherwise, we only use the Hoare type monitors in this course.
Examples

- Producer/Consumer
- Dining Philosophers
- Alarm Clock
- Readers-Writers (Reader Priority)
- Readers-Writers (Take Turns)
We use the Hoare type monitors unless stated otherwise
Monitor: Producer/Consumer

Monitor ProdCons
{
  int count, in, out;
  int Buf[SIZE];
  condition
    UntilFull,
    UntilEmpty;

  procedure PUT(int);
  procedure GET(int *);
  {
    count = 0
  }
}
Monitor: PUT() and GET()

void PUT(int X)
{
    if (count == SIZE)
        UntilEmpty.wait();
    Buf[in] = X;
    in = (in+1) % SIZE;
    count++;
    if (count == 1)
        UntilFull.signal();
}

void GET(int *X)
{
    if (count == 0)
        UntilFull.wait();
    *X = Buf[out];
    out = (out+1) % SIZE;
    count--;
    if (count == SIZE-1)
        UntilEmpty.signal();
}
Run This Solution with Mesa?

**Buffer Size = 2**

<table>
<thead>
<tr>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_4$</th>
<th>$C_1$</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Add 1 item</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Add 1 item</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Wait <strong>UntilEmpty</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Take 1 item</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>buffer is full</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Signal <strong>UntilEmpty</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>if it is a Hoare monitor, $P_3$ runs immediately</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add 1 item</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td><strong>signalizing thread continues and exits</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>By the time $P_3$ finally runs, there is no space!</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No space!</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td><strong>monitor is empty. allow one to enter.</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Under Mesa, $C_1$ continues.</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Upon exit, the monitor becomes empty and selects $P_4$ to enter</strong></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>
Dining Philosophers, Again!

- Let us look at another solution to the dining philosophers problem.
- Recall that slides 138-139 of 08-Semaphores.pdf discussed a solution suggested by Dijkstra. This solution requires that a philosopher can eat only if he can get **BOTH** chopsticks **at the same time**.
- This solution can be implemented using a monitor easily.
Monitor Definition

monitor philosopher
{
    enum { THINKING, HUNGRY, EATING} state[5];
    condition self[5];
    private: CanEat(int);

    procedure GET(int);  // get BOTH chopsticks
    procedure PUT(int);  // release chopsticks

    { for (i=0; i<5; i++)
        state[i] = THINKING;
    }
}
The **CanEat()** Procedure

- If the left and right neighbors of philosopher \( k \) are not eating and philosopher \( k \) is hungry, then philosopher \( k \) can eat. Thus, release him!

```c
void CanEat(int k) {
    if ((state[(k+4)%5] != EATING) &&
        (state[k] == HUNGRY) &&
        (state[(k+1)%5] != EATING)) {
        state[k] = EATING;
        self[k].signal();
    }
}
```
The GET() and PUT() Procedures

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

void PUT(int i)
{
    state[i] = THINKING;
    CanEat((i+4) % 5);
    CanEat((i+1) % 5);
}
How about Deadlock?

This solution does not cause deadlock, because

1. The only place where eating permission is granted is in procedure `CanEat()`, and
2. Philosopher $k$ can eat only if he could get both chopsticks (i.e., no hold-and-wait and no circular waiting).

```c
void CanEat(int k)
{
    if ((state[(k+4)%5] != EATING) &&
        (state[k] == HUNGRY) &&
        (state[(k+1)%5] != EATING)) {
        state[k] = EATING;
        self[k].signal();
    }
}
```
How about Bounded Waiting?

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

- **Question**: The Progress condition is meet and could be proved easily. How about the **Bounded Waiting** condition? More precisely, is it possible that some philosophers can continue the process of thinking and eating and block some others indefinitely? **Exercise.**
A Simple Alarm Clock: 1/4

- A set of **Sleeper** threads wish to **Slumber** for various times and set an alarm clock to wake them when it is time to get up.

- Unfortunately, their alarm clock is a little primitive:
  - Every hour it squirts cold water at the nearest sleeper, who immediately prods the next sleeper.
  - Each sleeper checks the time: if it is not the time for him/her to go to work, then he/she goes back to sleep.

- An external thread calls a monitor procedure every hour to initiate this waking up operation.

A Simple Alarm Clock: 2/4

Monitor Definition

```
monitor AlarmClock
{
    condition Wake;       // sleepers sleep here
    int Now = 0;           // hour counting

    procedure Tick();
    procedure Slumber(int n);  // n is the # of hours the
                                //  caller needs to sleep
}
```
void Tick(void) // called by an external timer
               // every hour
{
    Now = Now + 1;  // This is internal hour count
    Wake.signal();  // wake up a sleeper
}
The **while** loop controls the number of hours a **Slumber** can sleep.  
The **Tick()** procedure updates the time and wakes up the first slumber.  
This **Slumber** wakes up the next one. The **Signal** is lost if no one there.  
This is referred to as cascading release/signal.  
Cascading release can release all waiting threads even though the # is unknown.
The Readers-Writers Problem
Reader Priority

- We still need a reading reader count reading and a waiting reader count readers.
- Two condition variables are needed: OK_to_Read and OK_to_Write:
  - OK_to_Read : readers wait here if they cannot read because a writer is writing.
  - OK_to_Write: writers wait here if they cannot write because a writer is writing, or readers are reading.
Monitor Definition

monitor reader-writer
{
    int reading = 0;  // reading readers
    int readers = 0;  // waiting readers
    Bool busy = FALSE; // writer is writing

    condition OK_to_Read, OK_to_Write;

    procedure read_REQUEST(void);
    procedure read_RELEASE(void);
    procedure write_REQUEST(void);
    procedure write_RELEASE(void);
}


Readers and Writers

Reader

while (1) {
  // do something
  read_REQUEST();
  // reading
  read_RELEASE();
  // do something
}

Writer

while (1) {
  // do something
  write_REQUEST();
  // writing
  write_RELEASE();
  // do something
}
void read_REQUEST()
{
    if (busy) { // if a writer is writing
        readers++; // this reader must wait
        OK_to_Read.wait(); // wait on OK_to_read
        readers--; // released!
    }
    reading++; // if not busy or released
    OK_to_Read.signal(); // let the next reader to go
}

void read_RELEASE()
{
    reading--; // a reader has done reading
    if (reading == 0) // is this reader the last one?
        OK_to_Write.signal(); // YES, allow a writer to go
 ...
if there is no reader reading, yield to a writer

Reader Priority:
If there is a writer writing, this reader waits!
void write_REQUEST()
{
    if (busy || reading != 0) // if a writer is writing
        // or readers are reading
        OK_to_Write.wait();     // this writer must wait
    busy = TRUE;              // otherwise, start to write
}

void write_RELEASE()
{
    busy = FALSE;             // a writer done writing
    if (readers != 0)         // if some waiting readers
        OK_to_Read.signal();  // allow a reader to go
    else                      // otherwise
        OK_to_Write.signal(); // allow a writer to go
}
Monitor Code: Summary

```c
void read_REQUEST()
{
    if (busy) {
        readers++;
        OK_to_Read.wait();
        readers--;
    }
    reading++
    OK_to_Read.signal();
}

void write_REQUEST()
{
    if (busy || reading != 0)
        OK_to_Write.wait();
    busy = TRUE;
}
```

```c
void read_RELEASE()
{
    reading--;
    if (reading == 0)
        OK_to_Write.signal();
    else
        OK_to_Read.signal();
}
```

```c
void write_RELEASE()
{
    busy = FALSE;
    if (readers != 0)
        OK_to_Read.signal();
    else
        OK_to_Write.signal();
}
```

- Readers only block here.
- The exiting writer releases a writer only if there is no waiting readers.
- The exiting writer releases a waiting reader.
- The last reader releases a waiting writer.
- Only if there is no readers waiting or reading, a writer is released.
The Reader-Writer Problem: Again!

- Let us add a minor modification to the readers-writers (priority version) problem to make it a bit more realistic:
  - If a writer is waiting, the new readers should yield to a writer.
  - Upon the exit of a reader,
    - If there are waiting writers, let one go
  - Upon the exit of a writer,
    - If there are waiting readers, let one go
    - If there are waiting writers, let one go
  - So, the readers and writers take turns.
Monitor Definition

monitor reader-writer
{
    int readers = 0;  // waiting readers
    int reading = 0;  // reading readers
    int writing = 0;  // writing writers
    int writers = 0;  // waiting writers

    condition OK_to_Read, OK_to_Write;

    procedure read_REQUEST (void);
    procedure read_RELEASE (void);
    procedure write_REQUEST (void);
    procedure write_RELEASE (void);
}

void read_REQUEST()
{
    if (writing > 0 || writers > 0) {// if a writer writing
        // or there are waiting writers
        readers++;
        OK_to_Read.wait(); // this reader waits
        reading++;         // one more reading reader
        OK_to_Read.signal(); // allow other readers to read
    }
}

void read_RELEASE()
{
    if (--reading == 0) // if this is the last reader
        OK_to_Write.signal(); // let a writer go
Monitor Code for Writers

```c
void write_REQUEST()
{
    if (writing > 0 || readers > 0) { // if a writer writing
        writers++; // or there are waiting readers
        OK_to_Write.wait(); // this writer waits
    }
    writers--; // this writer is released
}
writing = 1; // this writer starts writing
}

void write_RELEASE()
{
    writing = 0; // this writer finishes writing
    if (readers > 0) // if there are readers waiting
        OK_to_Read.signal(); // let a reader to go
    else // otherwise
        OK_to_Write.signal(); // let a writer to go
}
Monitor Code: Summary

void read_REQUEST()
{
    if (writing > 0 || writers > 0) {
        readers++; OK_to_Read.wait();
        readers--; OK_to_Read.signal();
    }
    reading++; OK_to_Read.signal();
}

void write_REQUEST()
{
    if (writing > 0 || readers > 0) {
        writers++; OK_to_Write.wait();
        writers--; OK_to_Write.signal();
    }
    writing = 1;
}

void read_RELEASE()
{
    if (--reading == 0)
        OK_to_Write.signal();
}

void write_RELEASE()
{
    writing = 0;
    if (readers > 0)
        OK_to_Read.signal();
    else
        OK_to_Write.signal();
}

readers → writer → readers → writer → ... (Taking Turns)

cascading release

the exiting writer releases a waiting reader

the last reader releases a writer

if there is no waiting writers, these signals are lost!

no waiting readers let a writer go
The Reader-Writer Problem Exercise...

- Consider a solution that only uses a single condition, `Take_Turn`, rather than two: `OK_to_Read` and `OK_to_Write`.

- A reader waits on `Take_Turn` until there are no waiting writers, and then waits on `Take_Turn` again until there is no writer writing. In this case, this reader is safe to read.

- A writer waits on `Take_Turn` if there are readers reading or a writer writing.

- A reader or writer signals `Take_Turn` when they finish reading or writing.
Monitor Definition

monitor reader-writer
{
    int reading = 0;  // reading readers
    int writing = 0;  // writing writers
    int writers = 0;  // waiting writers

    condition Take_Turn;

    procedure read_REQUEST(void);
    procedure read_RELEASE(void);
    procedure write_REQUEST(void);
    procedure write_RELEASE(void);
}


Monitor Code for Readers

```c
void read_REQUEST()
{
    if (writers > 0) // if there are writers waiting,
        Take_Turn.wait(); // wait until released
    if (writing > 0) // if there is a writer writing,
        Take_Turn.wait(); // wait, of course
    reading++;       // because no writer is writing,
                     // a reader can read
}

void read_RELEASE()
{
    reading--;       // a reader has done reading
    Take_Turn.signal(); // let one reader/writer to go
}
```
Monitor Code for Writers

```c
void write_REQUEST()
{
    writers++;           // one more write request
    if (reading || writing) // if there are readers reading
        Take_Turn.wait();  //    or a writer writing, wait
    writing++;           // if no readers reading and no
                           //    writer writing, then go!
}

void write_RELEASE()
{
    writing--;           // reduce writing count
    writers--;           // reduce writer count
    Take_Turn.signal();  // let some one to proceed
}
```

data this means reading or writing being non-zero

The Reader-Writer Problem

Question...

- Is this solution correct?
- If you think that this solution is correct, please prove that this solution does implement the requirements correctly (i.e., taking turns correctly).
- If you think that this solution is incorrect, then
  - Explain what the problems are and use execution sequences to show that this solution fails to implement taking turns correctly.
  - And, modify this solution to make it working!
Three Extensions

empty(), Priority wait(), broadcast (i.e., signal_all)
Some monitor implementations may have additional features. For example, in addition to `wait()` and `signal()`, a condition variable may have one more function `empty()`, which returns `TRUE` if the condition variable has no waiting threads.

C. A. R. Hoare proposed the following version:

- A new reader should not be permitted to start if there is a waiting writer;
- At the end of a write operation, waiting readers are given preference over waiting writers.

We have seen this previously.
Monitor Definition

```c
monitor reader-writer
{
    int reading = 0;    // reading readers
    Bool writing = FALSE; // writing?

    condition OKtoRead, // readers wait here
    OKtoWrite;       // writers wait here

    procedure read_REQUEST(void);
    procedure read_RELEASE(void);
    procedure write_REQUEST(void);
    procedure write_RELEASE(void);
}
```
Monitor Code for Readers

void read_REQUEST()
{
    if (writing || !empty(OKtoWrite)) // if writing or
        OKtoRead.wait(); // writer queue not empty, wait!
    reading++; // then this reader can read
    OKtoRead.signal(); // allow a waiting reader to go
} // cascading release here!

void read_RELEASE()
{
    reading--; // a reader has done reading
    if (reading == 0)
        OKtoWrite.signal();
}
Monitor Code for Writers

```c
void write_REQUEST()
{
    if (writing || reading > 0) // if someone writing
        OKtoWrite.wait(); // or readers waiting, wait here
    writing = TRUE; // after released, write!
}

void write_RELEASE()
{
    writing = FALSE; // finished writing
    if (!empty(OKtoRead))// if there are waiting readers
        OKtoRead.signal(); // let one go (cascading!)
    else
        OKtoWrite.signal(); // or let a waiting writer go
} // if there is no waiting
```

not all systems have `empty()`
because of `empty()`, a waiting readers count is eliminated
Disk Organization: 1/2

- The information stored on the disk is organized into a series of cylinders, each of which consists of several tracks, and each track consists of several sectors or blocks.
- When a read/write request comes, the disk head assembly is moved to the correct cylinder, the head corresponding to the required track is selected, and the information may be read or written when the required sector passes the head as the disk rotates.
- The time to move the head assembly to a track is referred to as the seek time, which is mechanical and time consuming.
- Because there are many cylinders to be accessed, how can we minimize the total seek time?
Disk Organization: 2/2

- Instead accessing the cylinders based on the incoming order, it would be more efficient and probably cause less mechanical strain on the device if the requests could be handled by making continuous sweeps across the disk surfaces, first in one direction and the other.

- Some systems have an extended `wait()` method.

- The `wait(Priority)` has an integer argument `Priority`.

- When a condition variable is signaled, the thread with the highest priority is released. A smaller value of `Priority` usually means a higher priority. **WARNING**: check the system for the details.
monitor **Disk_Scheduler**
{
    int Head = 0; // initially at cylinder 0
    int Dir = IN; // thus, IN sweep first
    int MAX = ...; // highest cylinder number
    Bool Busy = FALSE; // initially the disk is free

    condition IN_Sweep, // IN sweep queue
    OUT_Sweep; // OUT sweep queue

    procedure Request(int Cy); // request to use cylinder Cy
    procedure Release(void); // release cylinder Cy
}
The Use of Disk_Scheduler

Request(Cylinder); // request to move the
//   disk assembly to
//   cylinder Signal all

activate the desired track
read or write the needed sector/block

Release(); // release the cylinder
// after the access
Monitor Code for Request

```c
void Request(int Cy) // request to access Cylinder Cy
{
    if (Busy) {
        if ((Head < Cy) ||
            (Head == Cy && Dir == IN))
            // if disk is busy, wait!
            // if beyond the current pos
            // or at the current pos
            // & direction is IN
            IN_Sweep.Wait(Cy);
        else
            // Otherwise, wait for OUT
            OUT_Sweep.Wait(MAX - Cy);
    } // if disk is busy, wait!
    // if beyond the current pos
    // or at the current pos
    // & direction is IN
    // wait on IN sweep CV
    // Otherwise, wait for OUT
    // sweep using a reversed
    // priority order
    // Once get out, can access
    // the disk head is at Head
    Busy = TRUE;
    Head = Cy;
}
```

When the disk head moves inward, cylinders are accessed in an ascending order. Thus, a smaller cylinder number means a higher priority.

When the disk head moves outward, cylinders are accessed in a descending order. Thus, a smaller cylinder number means a lower priority, and MAX - Cy is used.
Monitor Code for Release

void Release(void) {
  Busy = FALSE;
  if (Dir == IN) {
    if (!IN_Sweep.empty())
      IN_Sweep.signal();
    else {
      Dir = OUT;
      OUT_Sweep.signal();
    }
  }
  else {
    if (!OUT_Sweep.empty())
      OUT_Sweep.signal();
    else {
      Dir = IN;
      IN_Sweep.signal();
    }
  }
}

// request to release

// done using cylinder Cy
// IN sweep?
// if there are waiting in IN
// release one of them
// no one waiting in IN
// switch direction to OUT
// release one of them

// symmetric for IN and OUT
// current OUT sweep
// if there are waiting in OUT
// release one of them
// no one waiting in OUT
// switch direction to IN
// release one of them
The Broadcast/Signal_all Method

- In some systems, each condition variable may have a `broadcast` or `signal_all` method.
- Let `cv` be a condition variable. `cv.broadcast()` or `cv.signal_all()` releases all waiting threads on condition variable `cv` with a single signal call.
- When `cv.broadcast()` (or `cv.signal_all()`) is called, all waiting threads on condition variable `cv` are released and changed to inactive/re-entry.
- This will save the use of a cascading release.
A Simple Alarm Clock

Procedures Tick & Slumber

```c
void Tick(void)   // called by an external timer
   //   every hour
{
    Now = Now + 1;  // This is internal hour count
    Wake.broadcast();   // wake up all sleepers
}

void Slumber(int n)   // n = sleeping hours
{
    int AlarmCall;  // time to wake up
    AlarmCall = Now + n;  // update my alarm clock
    while (Now < AlarmCall) {  // as long as I can sleep
        Wake.Wait();   // sleep
        Wake.signal();  // no more needed
    }
}
```

- Cascading release is not needed.
Readers-Writers (Reader Priority)

void read_REQUEST()
{
    if (busy) {
        readers++;
        OK_to_Read.wait();
        readers--;
    }
    reading++
    OK_to_Read.signal();
}

cascading release
no more needed

void read_RELEASE()
{
    reading--;
    if (reading == 0)
        OK_to_Write.signal();
}

cascading release
is no more needed

void write_REQUEST()
{
    if (busy || reading != 0)
        OK_to_Write.wait();
    busy = TRUE;
}

the exiting writer releases a writer
only if there is no waiting readers

void write_RELEASE()
{
    busy = FALSE;
    if (readers != 0)
        OK_to_Write.signal();
    else
        OK_to_Read.broadcast();
    OK_to_Write.signal();
}

release ALL
waiting readers

the last reader releases
a waiting writer
Readers-Writers: Taking Turns

void read_REQUEST()
{
    if (writing > 0 || writers > 0) {
        readers++;
        OK_to_Read.wait();
        readers--;
    }
    reading++;
    OK_to_Read.signal();
}

void write_REQUEST()
{
    if (writing > 0 || readers > 0) {
        writers++;          OK_to_Write.wait();
        writers--;
    }          writing = 1;
}

void read_RELEASE()
{
    if (--reading == 0)      OK_to_Write.signal();
        OK_to_Read.signal();
}

void write_RELEASE()
{
    writing = 0;
    if (readers > 0)
        OK_to_Read.broadcast();
    if (writers > 0)
        OK_to_Write.signal();
    else
        OK_to_Write.signal();
}
An Important Note

- `cv.broadcast()` is not the same as
  ```
  while (!cv.empty())
    cv.signal();
  ```
- In each iteration of the `while` loop, a waiting thread is released, and this released thread gets the monitor (Hoare type) while the signaling thread becomes inactive/re-entry.
- When the released thread exits, the monitor mechanism picks a thread to enter. If the picked one is the signaling thread, the situation is fine.
- Otherwise, the newcomer could cause problems.
Monitor Barrier has an initial value \( n > 0 \) and a \( \text{Barrier}_\text{wait}() \) method.

A thread that calls \( \text{Barrier}_\text{wait}() \) blocks if the number of blocked threads is less than \( n - 1 \).

When the \( n \)-th thread calls \( \text{Barrier}_\text{wait}() \), it releases all waiting \( n-1 \) threads.

In other words, barrier Barrier blocks the first \( n-1 \) threads that calls \( \text{Barrier}_\text{wait}() \) and the \( n \)-th one releases all waiting threads. Thus, all \( n \) threads would act in a single group.
Example: Barrier 2/8

Monitor Barrier
{
  int n;         // the maximum threads to be queued
  int count = 0; // counting the current # queued

  condition barrier; // queue threads here

  procedure Barrier_wait() // method for threads to call
  {
    count++;        // one more thread entering barrier
    if (count < n) {
      barrier.wait(); // no! queue this thread
      barrier.signal(); // when released, releases the next
    } else {
      count = 0;      // this is the n-th thread …
      // reset the counter
      barrier.signal(); // releases a waiting one and cascades
    }
  }
}

NOTE: This is a Hoare type monitor
Example: Barrier 3/8

- What if the order of `count=0` and `barrier.signal()` is switched as follows?
  
  ```java
  barrier.signal();
  count = 0;
  ```

- Does this solution work? In other words, after switching the two statements in the `else` part, can the n-th thread correctly release the waiting n-1 threads?
Example: Barrier 4/8

- **NO, this does not work properly.**
- Suppose n = 3. We may have the following scenario:
  - $T_1$ calls `Barrier_wait()` and waits ($\text{count} = 1$)
  - $T_2$ calls `Barrier_wait()` and waits ($\text{count} = 2$)
  - $T_3$ calls `Barrier_wait()` and executes the `else`:
    - $T_3$ calls `barrier.signal()` (currently $\text{count} = 3$) and assume that $T_1$ is released. $T_3$ yields the monitor.
    - $T_1$ is active and calls `barrier.signal()` (current: $\text{count} = 3$). This releases $T_2$.
    - $T_2$ calls `barrier.signal()` and exits.
  - Who will enter? If it is the signaling threads $T_3$, it is OK. Otherwise, a new thread enters, sees $\text{count < 3}$ being `FALSE` and go to `else` and start another `signal()`. This is a wrong program logic.
Example: Barrier 5/8

monitor Barrier
{
  int n;  // the maximum threads to be queued
  int count = 0;  // counting the current # queued

  condition barrier;  // queue threads here

  procedure Barrier_wait()  // method for threads to call
  {
    count++;  // one more thread entering barrier
    if (count < n) {
      // is this the last one?
      barrier.wait();  // no! queue this thread
      barrier.signal();
    }
    else {
      // this is the n-th thread …
      count = 0;  // reset the counter
      barrier.broadcast();  // releases a waiting one and cascades
    }
  }
}

NOTE: This is a Hoare type monitor
Example: Barrier 6/8

monitor Barrier
{
    int n; // the maximum threads to be queued
    int count = 0, i; // counting the current # queued

    condition barrier; // queue threads here

    procedure Barrier_wait() // method for threads to call
    {
        count++; // one more thread entering barrier
        if (count < n) // is this the last one?
            barrier.wait(); // no! queue this thread
        else {
            count = 0; // reset the counter
            for (i = 1; i <= n-1; i++) // releases the threads using for
                barrier.signal(); // one at a time
        }
    }
}
Example: Barrier 7/8

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>barrier</th>
<th>count</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>$T_2$</td>
<td>$T_3$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>count++</td>
<td>$\emptyset$</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wait()</td>
<td>$\emptyset$</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>count++</td>
<td>$T_1$</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wait()</td>
<td>$T_1$</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_1$ is released</td>
<td>count++</td>
<td>$T_1$, $T_2$</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>count = 0</td>
<td>$T_1$, $T_2$</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_1$ comes back fast</td>
<td>signal()</td>
<td>$T_2$, 0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>continue &amp; exit</td>
<td>$T_2$, 0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>come back fast</td>
<td>$T_2$</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>count++</td>
<td>$T_2$, 1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wait()</td>
<td>$T_1$, $T_2$</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_1$ is released again!</td>
<td>signal()</td>
<td>$T_1$, $T_2$</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>continue</td>
<td>$T_2$, 1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: Barrier 8/8

- If you use a Hoare type monitor, try to use cascading release if it is possible.
- If `broadcast()` is available, this is a good alternative of cascading release.
- Do not use a `for` or a `while` loop to release the blocked threads on a condition.
- **QUESTION**: Is the `for/while` version working correctly under a MESA monitor?
- **QUESTION**: Are cascading and `broadcast` releases work correctly under a MESA monitor?
Semaphore and Monitor Equivalence
Semaphore and Monitor Equivalence

- In terms of expressive power, semaphores and monitors are equivalent.
- A semaphore can easily be implemented with a monitor.
- Conversely, a monitor and its condition variables can also be implemented with multiple semaphores, although this is a bit tedious.
- Therefore, semaphores and monitors are equivalent because one may be implemented by the other.
Semaphore Implementation Using a Monitor
Semaphores in Hoare Type: 1/2

- On the right is a possible implementation of a semaphore using a Hoare type monitor.
- \textit{count} is the semaphore counter.
- \textit{c} is a condition variable for blocking a thread that must wait.
- Does it work?

```c
monitor Hoare_Sem
{
    int count = 0;
    condition c;

    P(void)  // wait
    {
        count--;
        if (count < 0)
            c.wait();
    }

    V(void)  // signal
    {
        count++;
        c.signal();
    }
}
```

there is no if statement here
Why is there no if?

Recall that \( \text{abs}(\text{count}) \) is the number of waiting threads on that semaphore.

It \texttt{count++} is not positive, there are waiting threads and \texttt{c.signal()} will release one.

If \texttt{count++} is positive, no waiting threads on \texttt{c} and \texttt{c.signal()} is lost.

```c
monitor Hoare_Sem
{
    int        count = 0;
    condition  c;

    P(void)    // wait
    {
        count--;
        if (count < 0)
            c.wait();
    }

    V(void)    // signal
    {
        count++;
        c.signal();
    }
}
```
Semaphores in Mesa Type: 1/2

- Now let us try the same implementation using the Mesa type monitor.
- `count` is the semaphore counter.
- `c` is a condition variable for blocking a thread that has to wait.
- Does it work?

```c
monitor Mesa_Sem
{
    int        count = 0;
    condition  c;

    P(void)    // wait
    {
        count--;
        while (count < 0)
            c.wait();
    }

    V(void)    // signal
    {
        count++;
        c.signal();
    }
}
```
Semaphores in Mesa Type: 2/2

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
<th>count</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>count--</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>while (...)</td>
<td></td>
<td></td>
<td>-1</td>
<td>$T_1$ calls $\mathcal{P}()$</td>
</tr>
<tr>
<td>c.wait</td>
<td></td>
<td></td>
<td>-1</td>
<td>$T_1$ blocks</td>
</tr>
<tr>
<td>count++</td>
<td></td>
<td></td>
<td>0</td>
<td>$T_2$ calls $\mathcal{V}()$</td>
</tr>
<tr>
<td>c.signal</td>
<td></td>
<td></td>
<td>0</td>
<td>$T_1$ released</td>
</tr>
<tr>
<td>waiting to re-enter</td>
<td>$T_2$ exit</td>
<td>count--</td>
<td>-1</td>
<td>$T_3$ calls $\mathcal{P}()$</td>
</tr>
<tr>
<td>while (...)</td>
<td></td>
<td>while (...)</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>c.wait</td>
<td></td>
<td></td>
<td>-1</td>
<td>$T_3$ blocks</td>
</tr>
<tr>
<td>while (...)</td>
<td></td>
<td>c.wait</td>
<td>-1</td>
<td>$T_1$ re-enters</td>
</tr>
<tr>
<td>c.wait</td>
<td></td>
<td></td>
<td>-1</td>
<td>$T_1$ blocks</td>
</tr>
<tr>
<td>$T_1$ and $T_3$ are both blocked</td>
<td></td>
<td></td>
<td>-1</td>
<td>$T_1$ and $T_3$ both blocked</td>
</tr>
</tbody>
</table>

Because of the Mesa type, $T_2$ continues after signaling. Upon exit, $T_3$ (rather than $T_1$) gets the control of monitor.

A correct signal fails to release a waiting thread properly.
Hoare Monitor Implementation
Using Semaphores
Simulating a Hoare Monitor
Using Semaphores: 1/4

- Because the boundary of a monitor must be mutually exclusive, a **mutex lock** is needed.
- Let this mutex lock be **Mutex**.
- Because the signaling thread must yield the monitor and wait, a semaphore **Suspended**, initialized to 0, is needed for this purpose.
- Threads waiting on **Suspended** are inactive ones.
- **Suspended_count** counts the number of threads suspended on semaphore **Suspended**.
Simulating a Hoare Monitor
Using Semaphores: 2/4

- Each monitor procedure has the following form:

```java
Mutex.wait(); // wait to enter

......
// procedure body

......
if (Suspended_count > 0) // there are inactive ones
    Suspended.signal(); // let one go
else
    Mutex.signal(); // or release the monitor
```

- If there are inactive/re-entry threads, let one go!
- The baton, Mutex, is given to the released thread.
- Therefore, this is in favor of the inactive/re-entry threads.
Simulating a Hoare Monitor
Using Semaphores: 3/4

- For condition `cv`, a semaphore `cv_Sem` and a counter `cv_Count` are needed, initialized to 0.

```java
void cv.wait()
{
    cv_Count++;
    if (Suspended_count > 0)
    {
        Suspended.signal();
        // let one go
    } else
    {
        Mutex.signal();
        // release the monitor
    }
    cv_Sem.wait();
    cv_Count--;
}
```

- Before a thread waits on a `cv`, an inactive/re-entry thread is released if there is one.
- Otherwise, it releases the `Mutex`.
- Then, wait on the `cv`. 
Simulating a Hoare Monitor
Using Semaphores: 4/4

- Here is `cv.signal()`:

```cpp
if (cv_Count > 0) {  // some waiting threads
    Suspended_count++;  // this one becomes inactive
    cv_Sem.signal();  // releases a waiting thread
    Suspended.wait();  // block the signaling one,
                        //    yielding the monitor
    Suspended_count--;  // signaling is back
}
// if no inactive, no action!
```

- Before the signaling thread yields the monitor to the released one, the signaling thread signals `cv_Sem` to release a waiting thread on that `cv`.
- If there is no inactive thread, this signal is considered never happened.
Simulating a Mesa Monitor
Using Semaphores

- We have seen the simulation of a Hoare type monitor using semaphores.
- It is certainly possible to simulate the Mesa type monitor using semaphores.
- Please think this over and do it as an exercise.
- The implementation discussed earlier gives inactive/re-entry threads higher priority to enter a monitor. What if there is no such requirement? In other words, an inactive/re-entry thread is pushed outside of the monitor and waits alongside other entering threads?
Comparisons
Hoare Type vs. Mesa Type

- When a signal occurs, a Hoare type monitor uses two context switches, one switching the signaling thread out and the other switching the released in. However, a Mesa type monitor uses one.
- Thread scheduling must be very reliable with Hoare type monitors to ensure once the signaling thread is switched out the next one to execute in the monitor must be the released thread.
- With Mesa type monitors, a condition may be evaluated multiple times. However, incorrect signals will do less harm because every thread checks its own condition.
## Semaphores vs. Monitors

<table>
<thead>
<tr>
<th>Semaphores</th>
<th>Monitors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can be used anywhere, but should not be in a monitor</td>
<td>Can only be accessed with monitor procedure calls</td>
</tr>
<tr>
<td>No connection between the semaphore and the data this semaphore protects</td>
<td>Data and access procedures are in the same place (i.e., a monitor)</td>
</tr>
<tr>
<td>Semaphores are low level assembly language-like instructions</td>
<td>Monitors are well-structured higher-level construct</td>
</tr>
<tr>
<td>Not easy to use and prone to bugs</td>
<td>Easy of use and good protection of vital data</td>
</tr>
</tbody>
</table>
## Semaphores vs. Conditions

<table>
<thead>
<tr>
<th>Semaphores</th>
<th>Condition Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can be used anywhere, but not in a monitor</td>
<td>Can only be used in monitors</td>
</tr>
<tr>
<td><strong>wait()</strong> does not always block its caller</td>
<td><strong>wait()</strong> always blocks its caller</td>
</tr>
<tr>
<td><strong>signal()</strong> either releases a thread, or increases the semaphore counter</td>
<td><strong>signal()</strong> either releases a thread, or the signal is lost as if it never occurs</td>
</tr>
<tr>
<td>If <strong>signal()</strong> releases a thread, the caller and the released *<strong>both continue</strong></td>
<td>If <strong>signal()</strong> releases a thread, either the caller or the released continues, but <em><strong>not both</strong></em></td>
</tr>
</tbody>
</table>
Remarks
Reminder 1

- It is suggested that a `signal()` should be the last executed statement before exiting a monitor procedure. We have seen this in many examples.
- Why?
  - Whether this monitor is a Hoare type or Mesa type, the difference does not matter.
  - If this is a Hoare type, the signaling thread will not do anything even though it is inactive.
  - If this is a Mesa type, the signaling thread will exit immediately.
  - Therefore, this is a safer approach.
Reminder 2

Why is using semaphores in a monitor a bad idea?

- If a thread $T$ is executing in monitor $A$, monitor $A$ is “occupied” by thread $T$.
- What if thread $T$ calls a semaphore wait in monitor $A$ and is blocked?
- Because thread $T$ is executing in monitor $A$, if it waits on a semaphore in monitor $A$, monitor $A$ cannot be used as monitor $A$ is occupied by the thread $T$.
- Therefore, using semaphores in a monitor is not a good idea.
Reminder 3

- Why is calling a monitor procedure in another monitor from this monitor a bad idea?
  
  ➢ If a thread $T$ is executing in monitor $A$, monitor $A$ is “occupied” by thread $T$.
  
  ➢ If thread $T$ successfully enters another monitor $B$, monitor $A$ is not empty and cannot be used by any thread. This is inefficient!
  
  ➢ Worse, what if thread $T$ waits on a condition variable in monitor $B$? Then, monitor $A$ could not be used for a long time until thread $T$ returns from monitor $B$ and exits.
Monitors with ThreadMentor
ThreadMentor does not support empty(), Priority wait() and broadcast() or signal_all()
A monitor must be a derived class of class Monitor.

The initialization part should be in constructors.

Make monitor procedures public.

Local variables should be private/protected.
Monitor: Monitor Procedures

```cpp
int MyMon::MonProc(...) {
    MonitorBegin();
    // other statements
    // of this procedure
    MonitorEnd();
}
```

- Monitor procedures are C/C++ functions.
- Before you do anything, call `MonitorBegin()`.
- Before exit, call `MonitorEnd()`.
- The following is wrong:

```cpp
int MyMon::MonProc() {
    MonitorBegin();
    // other stuffs
    return 0;
    MonitorEnd();
}
```

MonitorBegin() locks the monitor and MonitorEnd() unlocks it. Thus, mutual exclusion is guaranteed.
Monitor: A Simple Example

Class Count

::public Monitor

{
  public:
  int Inc();
  int Dec();
  Count();

  private:
  int Counter;

  Count::Count(void)
  {
    Counter = 0;
  }

int Count::Inc()
{
  MonitorBegin();
  Counter++;
  MonitorEnd();
  return Counter;
}

int Count::Dec()
{
  MonitorBegin();
  Counter--;
  MonitorEnd();
  return Counter;
}
Monitor: Condition Variables

```java
Condition Event;
Event.Wait();
Event.Signal();
```

- **Condition** is a class and has two methods, `Wait()` and `Signal()`.  
- Waiting on a condition variable means waiting for that event to occur.  
- Signaling a condition variable means that the event has occurred.
The Philosopher Monitor Definition is as follows:

```cpp
class Mon::public Monitor
{
    public:
        Mon();
        GET(int);
        PUT(int);

    private:
        Condition Self[5];
        int State[5];
        int CanEat(int);
};
Mon::Mon()
{
    int i;
    for (i=0; i < 5; i++) {
        State[i] = THINKING;
    }
}
```

- **Condition variable pointers, one for each philosopher**
- **Get and put chopsticks**
- **Are both chopsticks available?**
- **State of each philosopher**
Philosopher Monitor Implementation

```cpp
int Mon::CanEat(int k) {
    if ((state[(k+4)%5] != EATING) && (state[k] == HUNGRY) && (state[(k+1)%5] != EATING)) {
        state[k] = EATING;
        self[k].Signal();
    }
}

void Mon::GET(int k) {
    MonitorBegin();
    state[k] = HUNGRY;
    CanEat(k);
    if (state[k] != EATING)
        self[k].Wait();
    MonitorEnd();
}
```

check to see if I can eat

if I cannot eat, wait
Specifying a Monitor Type

- A monitor type must be specified in your monitor constructor.

- Use **HOARE** or **MESA** for Hoare type and Mesa type monitors.

```cpp
MyMonitor::MyMonitor(char *Name) : Monitor(Name, HOARE )
{
    // initialization here
}
```

Replace **HOARE** with **MESA** if you wish to use a Mesa type monitor.
The End