Part III
Synchronization
Monitors

That's been one of my mantras - focus and simplicity.
Simple can be harder than complex:
You have to work hard to get your thinking clean to make it simple.
But it's worth it in the end because once you get there, you can move mountains.

Steve Jobs

Spring 2019

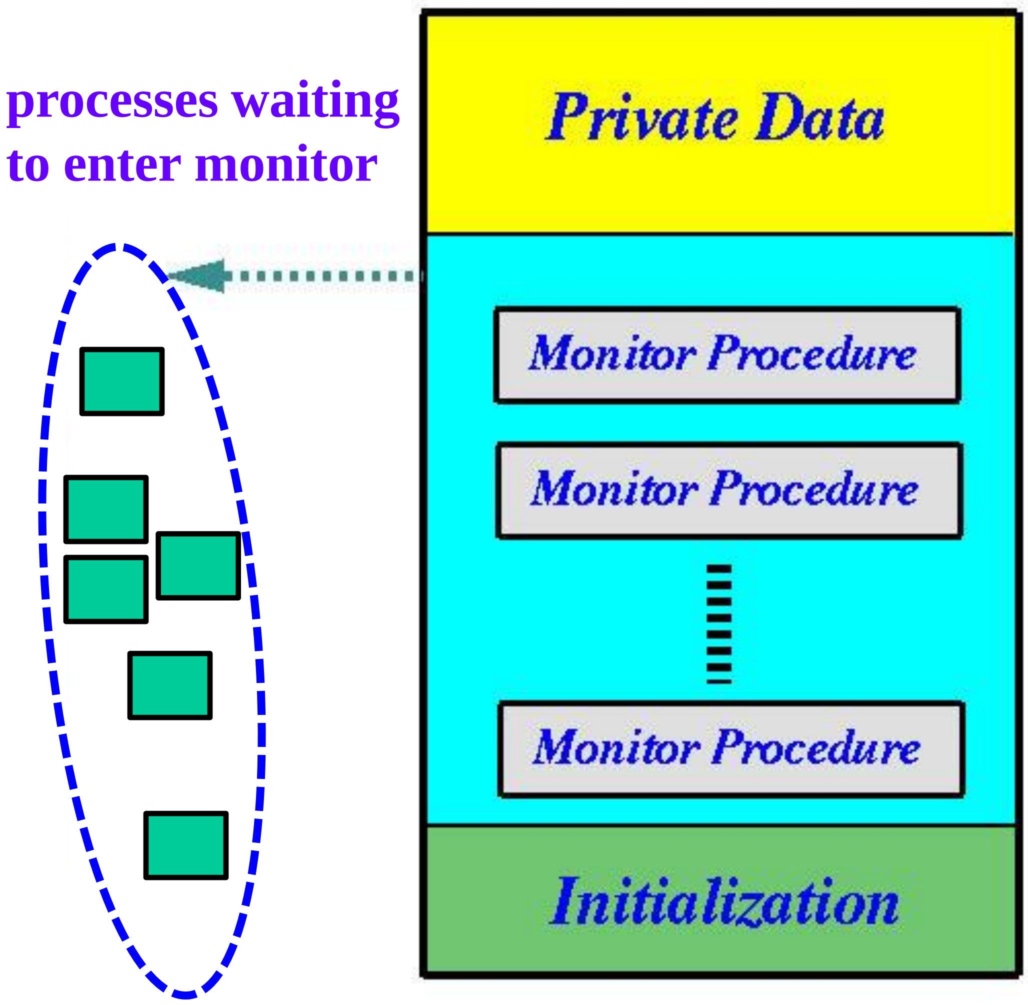
What Is a Monitor? - Basics

- 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 process*** **can be executing *in* a monitor. Thus, mutual exclusion is automatically guaranteed in a monitor.**
- **When a process calls a monitor procedure and enters the monitor successfully, it is the *only* process *executing* in the monitor.**
- **When a process calls a monitor procedure and the monitor has a process executing, the caller is blocked outside of the monitor.**

Monitor: Mutual Exclusion 2/2



- If there is a process executing in a monitor, any process that calls a monitor procedure is blocked ***outside*** of the monitor.
- When the monitor has no executing process, one process will be let in.

Monitor: Syntax

```
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

```
monitor IncDec
{
   int  count;
   void Increase(void)
   { count++; }

   void Decrease(void)
   { count--; }

   int GetData(void)
   {  return count; }

   {  count = 0; }
}
```

initialization

process Increment

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

Is the printed value the one just updated?

Condition Variables

- **Mutual exclusion is an easy task with monitors.**
- **While a process 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 process 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 process inside).**
- **This means that this process is waiting for the indicated event to occur.**

Condition signal

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

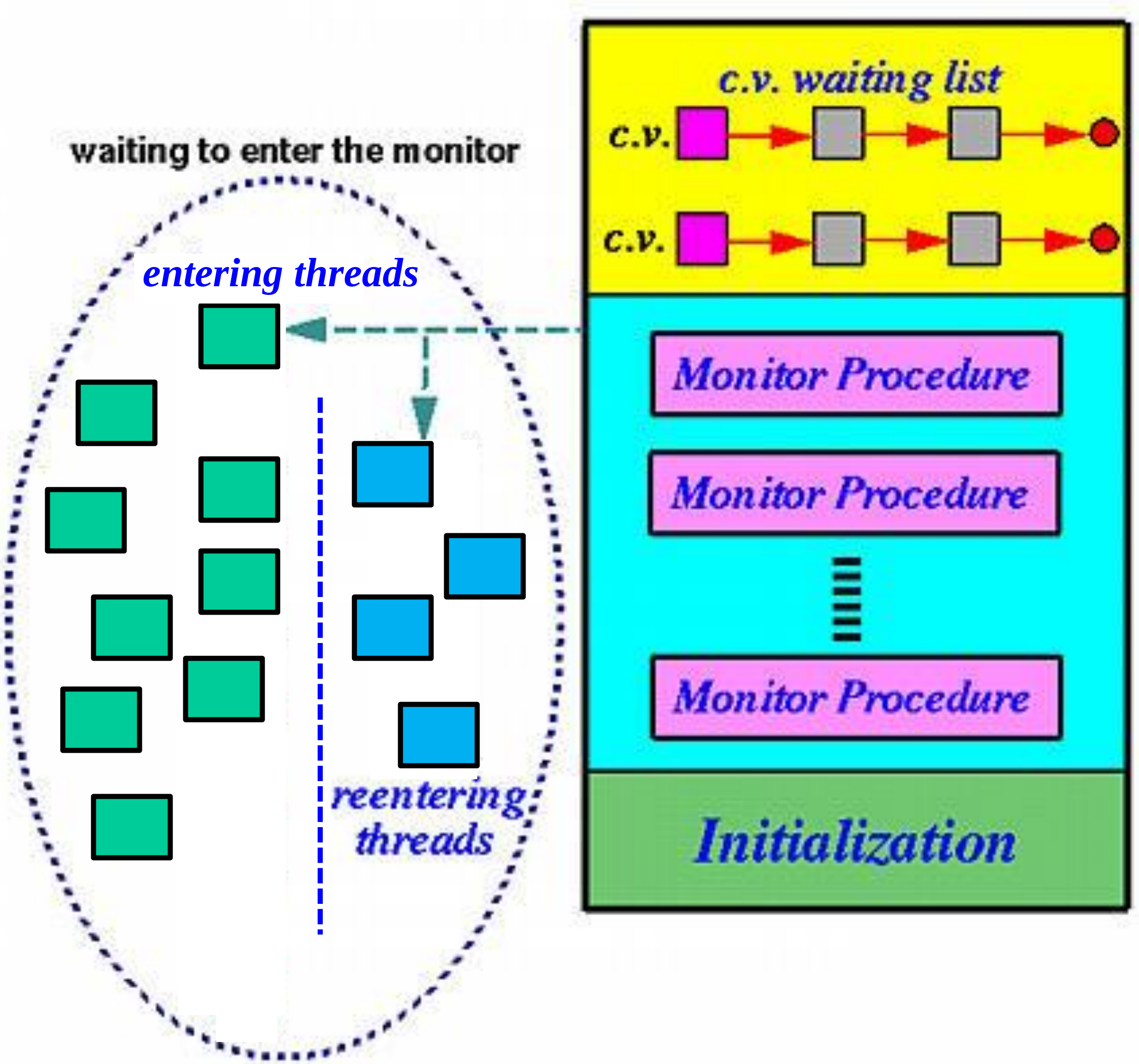
Two Types of Monitors

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

What Do You Mean by "*Waiting Somewhere*"?

- **The signaling process (Hoare type) or the released process (Mesa type) must wait somewhere.**
- **You could consider there is a waiting bench for these processes to wait.**
- **Hence, each process 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.**

Monitor with Condition Variables



- Processes blocked due to signal/wait are in the ***re-entry*** list (*i.e.*, waiting bench).
- When the monitor is free, a process is released from either ***entry*** or ***re-entry*** .

What Is the Major Difference?

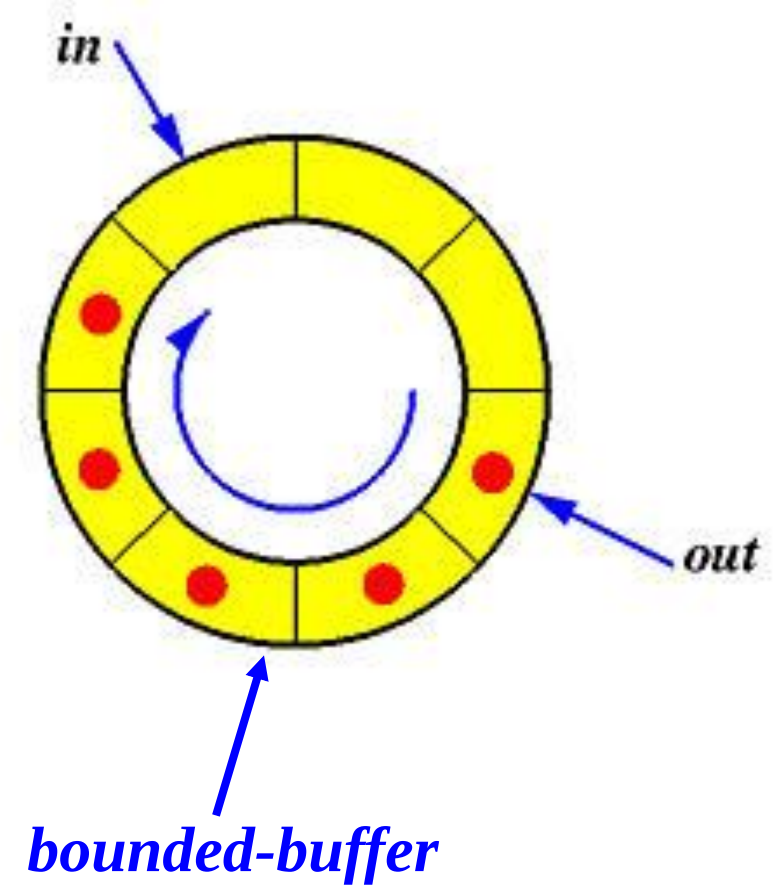
```
Condition  UntilHappen;

// 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 process and the condition is not changed. Thus, an `if` is sufficient.

With Mesa type, the released process may be waiting for a while before it runs. During this period, other processes 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.

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

P_1	P_2	P_3	P_4	C_1	Count
					0
Add 1 item					1
	Add 1 item				2
		Wait `UntilEmpty`			2
				Take 1 item	1
				Signal `UntilEmpty`	1
			Add 1 item		2
					2
		No space!			2

if it is a Hoare monitor, P_3 runs immediately

By the time P3 finally runs, there is no space!

Under Mesa, C_1 continues. Upon exit, the monitor becomes empty and selects P_4 to enter

Dining Philosophers: Again!

- **In addition to thinking and eating, a philosopher has one more state, hungry, in which he is trying to get chopsticks.**
- **We use an array `state[]` to keep track the state of a philosopher. Thus, philosopher `i` can eat (*i.e.*, `state[i] = EATING`) only if his neighbors are not eating (*i.e.*, `state[(i+4)%5]` and `state[(i+1)%5]` are not `EATING`).**

Monitor Definition

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

  procedure GET(int);
  procedure PUT(int);

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

The CanEat() Procedure

the left and right neighbors of philosopher *k* are not eating

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

philosopher *k* is hungry

- **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!**

The GET() and PUT() Procedures

I am hungry

see if I can eat

If I could not eat,
block myself

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

I finished eating

Let my neighbors eat

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

Which type of monitor am I using?

How about Deadlock?

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

- **This solution does not have 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).**

How about Bounded Waiting?

```
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***.

The Reader-Writer Problem: Again!

- **Let us add one minor modification to the original reader-writer problem to make it a bit more realistic:**
  - **If a writer is waiting, the new readers should waits their turn, even though it is safe to proceed if there are readers reading.**

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);
}
```

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
```

Monitor Code for Readers

```
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

this means reading or writing being non-zero

```
void write_REQUEST()
{
  writers++;                  // one more write request
  if (reading || writing)     // if there is readers reading
    Take_Turn.Wait();         //    or w 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
}
```

Hoare Type vs. Mesa Type

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

Semaphores vs. Monitors

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

Semaphores vs. Conditions

Semaphores	Condition Variables
Can be used anywhere, but not in a monitor	Can only be used in monitors
`wait()` does not always block its caller	`wait()` always blocks its caller
`signal()` either releases a process, or increases the semaphore counter	`signal()` either releases a process, or the signal is lost as if it never occurs
If `signal()` releases a process, the caller and the released *both continue*	If `signal()` releases a process, either the caller or the released continues, but *not both*

Semaphore and Monitor Equivalence

- In terms of expressive power, semaphores and monitors are equivalent.
- A semaphore can be implemented with a monitor. This is easy and is your homework.
- Conversely, a monitor and its condition variables may also be simulated with multiple semaphore, although this is tedious. See weekly reading list.
- Therefore, semaphores and monitors are equivalent because one may be implemented by the other.

Monitors with *ThreadMentor*

Monitor: Definition

```
class MyMon::public Monitor
{
  public:
    MyMon(); // constructor
    MonitorProcedure-1();
    MonitorProcedure-2();
    // other procedures
  private:
    // variables used in
    // this monitor
};
```

- **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

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

`MonitorBegin()` *locks the monitor and* `MonitorEnd()` *unlocks it. Thus, mutual exclusion is guaranteed.*

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

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

Monitor: A Simple Example

```
Class Count
      ::public Monitor
{
  public:
    int  Inc();
    int  Dec();
    void Count();
  private:
    int  Counter;
}

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

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

Monitor: Condition Variables

```
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.

Philosopher Monitor Definition

condition variable pointers, one for each philosopher

```
class Mon::public Monitor
{
   public:
      Mon();
      GET(int); PUT(int);
   private:
      Condition *Self[5];
      int  State[5];
      int  CanEat(int);
};
```

get and put chopsticks

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

are both chopsticks available?

state of each philosopher

Philosopher Monitor Implementation

```
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();
}
```

if I cannot eat, wait

check to see if I can eat

Specifying a Monitor Type

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

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

- A monitor type must be specified in your monitor constructor.
- Use HOARE or MESA for Hoare type and Mesa type monitors.

The End