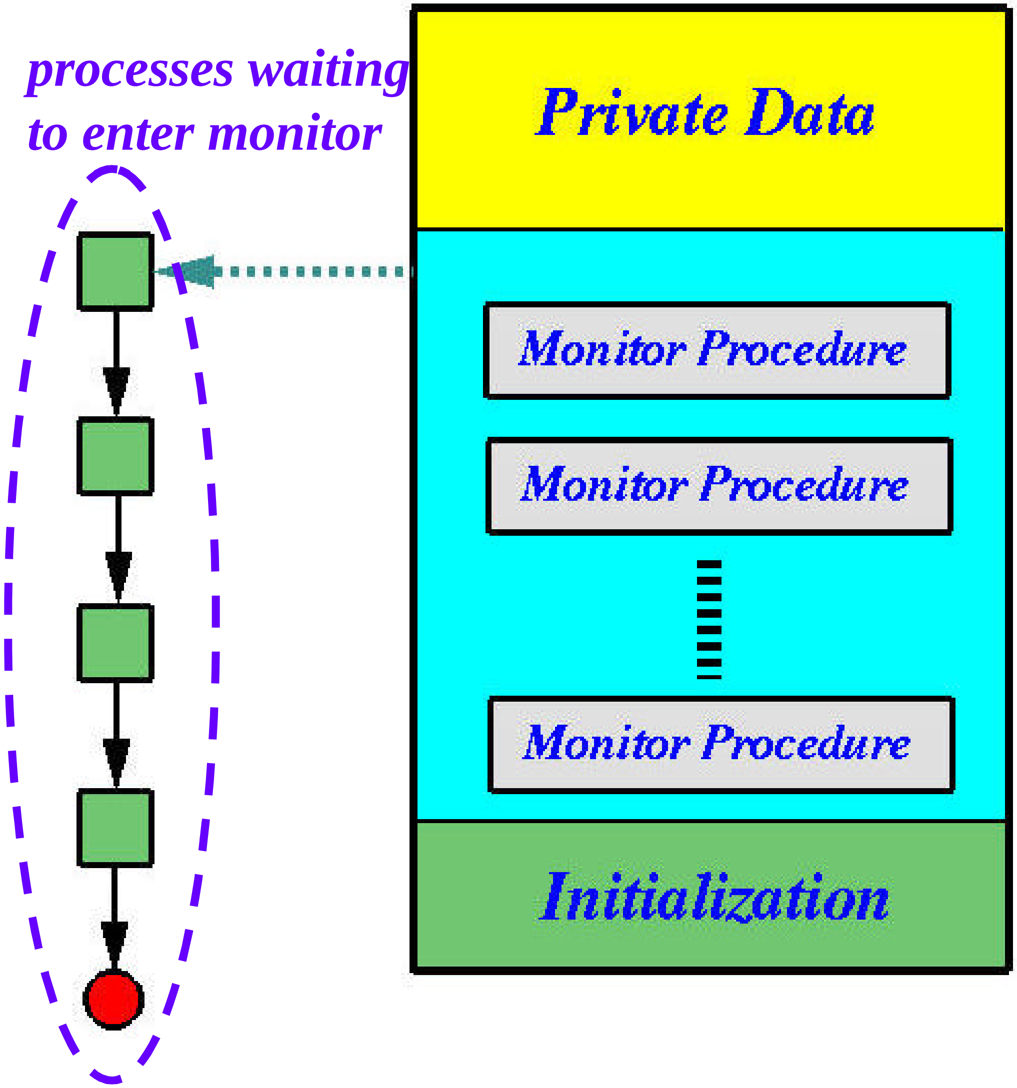
What Is a Monitor? - Basics

- 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 can be invoked by 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** ***within*** **a monitor. Thus,** ***mutual exclusion*** **is guaranteed within 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 running, the caller will be 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 can be made 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
}
```

Condition Variables

- **With monitors, mutual exclusion is an easy task.**
- **While the process is executing *within* a monitor, a programmer may want to block this process and force it to wait until an event occurs.**
- **Thus, each programmer-defined event is artificially associated with a *condition variable*.**
- **A condition variable, or a condition, has a**

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 *within a monitor*.
- A process that executes a condition wait **blocks immediately** and is put into the waiting list of that condition variable.
- 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 occurs.**
- **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!**

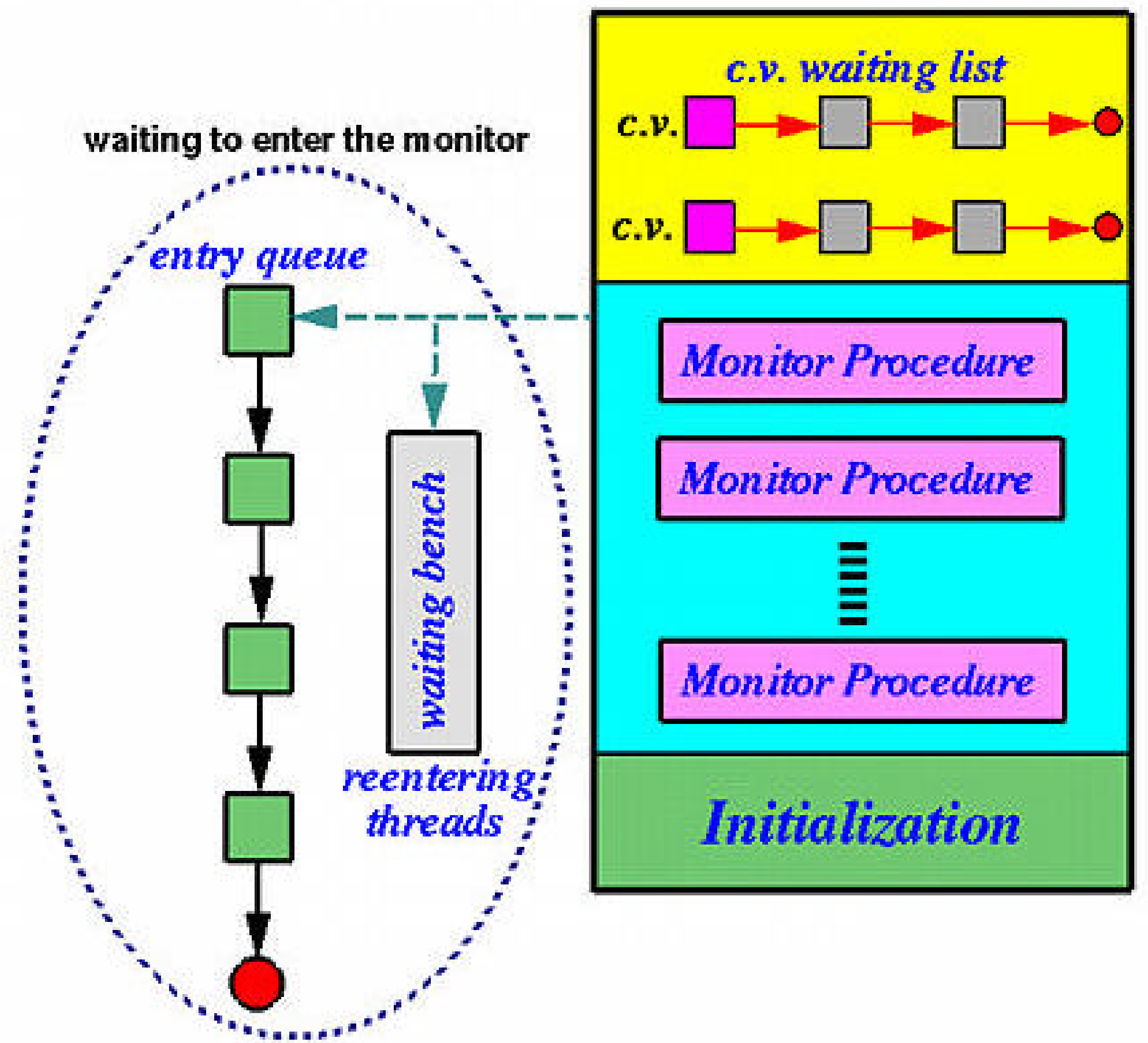
Two Types of Monitors

- After a signal, the released process and the signaling process may be executing in the monitor.
- There are two common and popular approaches to address this problem:
  - 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.

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 in a monitor for these processes to wait.
- As a result, each process that involves in a monitor call can 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 suspended 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 *incoming* or *re-entry* .**

What is the major difference?

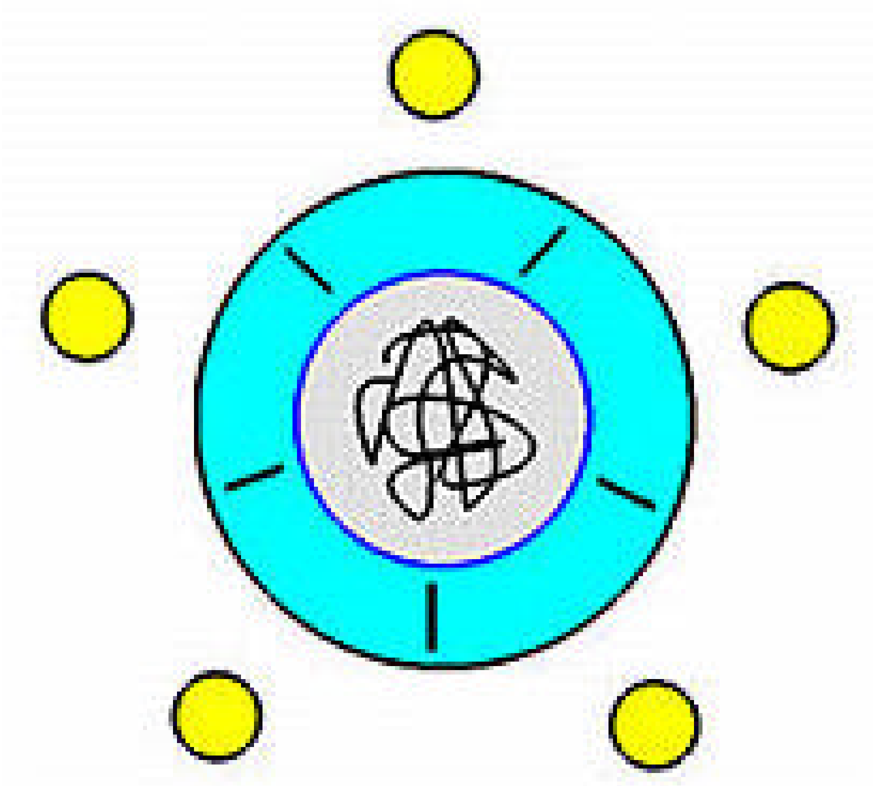
```
Condition  UntilHappen;

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

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

With Hoare type, once a signal arrives, the signaler has yielded the monitor to the released process and the condition is not changed. Thus, a `if` is sufficient.

With Mesa type, the released process may be suspended 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`!

Monitor: Dining Philosophers Revisited

- **Instead of picking up chopsticks one by one, we insist that a philosopher can eat only if he can *pick up both simultaneously*.**
- **Can we use a semaphore to protect chopsticks 0 and 1, another for 1 and 2, and so on? *No, no, no*.**
- ***Race condition!!!!!***



Monitor Definition

```
monitor Control
{
   bool used[5];
   condition self[5];
   private:
      int CanEat(int);

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

   { // initialization
     for (i=0;i<5;i++)
        used[i] = FALSE;
   }
}
```

```
int  CanEat(int i)
{
  if (!Used[i] &&
      !Used[(i+1)%5])
    return TRUE;
  else
    return FALSE;
}
```

Function `CanEat()` returns `TRUE` if both chops for Philosopher *i* are available.

Monitor: `GET()` and `PUT()`

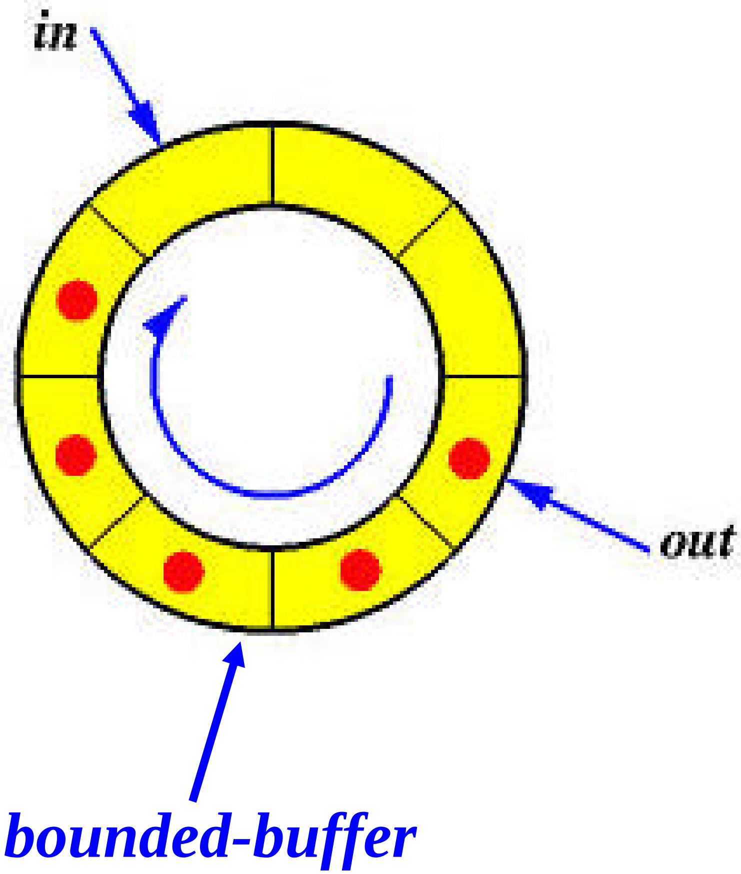
Why a `while` *rather than a* `if` `even with a Hoare monitor?`

```
void GET(int i)
{
  while (!CanEat(i))
      self[i].wait();
  Used[i] = TRUE;
  Used[(i+1)%5] = TRUE;
}
```

```
void PUT(int i)
{
  Used[i] = FALSE;
  Used[(i+1)%5]
     = FALSE;
  for (i=0;i<5;i++)
    self[i].signal();
}
```

- **In fact, `PUT()` only requires to signal `self[(i+1)%5]` and `self[(i+4)%5]`, the two neighbors of philosopher *i*.**
- **Does it really matter? Why?**

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();
}
```

- Change `if` to `while` for Mesa type monitors.

Dining Philosophers: Again!

- In addition to thinking and eating, a philosopher has one more state, hungry, in which he is trying to get chops.
- 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: test(int);

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

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

The `test()` Procedure

the left and right neighbors of philosopher k are not eating

```
void  test(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,
Then block myself

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

I finished eating

Let my neighbors
use my chops

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

Which type of monitor am I using?

How about Deadlock?

```
void test(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
  - The only place where eating permission is granted is in procedure `test()`, and …..
  - Philosopher *k* can eat only if his neighbors are not eating. Thus, no two neighboring philosophers can eat at the same time.

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.

Semaphore vs. Condition

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***