Monitors

Abstract data types (classes) that ensure mutual exclusion between operations (methods).

- Can't access 'permanent' (member) variables except through operations.
- Operations cannot access variables outside monitor (i.e., can only access permanent variables, local variables and parameters).
- Permanent variables are initialized before any operation can be invoked (constructor).
- Mutual exclusion is implicit. *At most 1 process occupies the monitor.*
- Implementation: Processes calling a monitor method are delayed on an "entry queue" until the monitor is unoccupied.
- Conditional Synchronization through *condition* variables.

Example: Time of Day

```
monitor TOD {
  int hr =0 , min = 0 , sec = 0 ;
  ## Inv: 0 <= hr < 24 /\ 0 <= min < 60 /\ 0 <= sec < 60

  procedure set( int h, int m, int s ) {
     ## Pre: 0 <= h < 24 /\ 0 <= m < 60
     ##  /\ 0 <= s < 60
     hr := h ; min := m ; sec := s ; }

  procedure get( int &h, int &m, int &s ) {
      h := hr ; m := min ; s := sec ; }

  procedure tick() {
      sec += 1 ;
      min += sec / 60 ; sec := sec % 60 ;
      hr += min / 60 ; min := min % 60 ;
      hr := hr % 24 ; } }
```

Monitor Invariant

MI — an assertion

- MI should be true whenever the monitor is "unoccupied"
- Capture consistency (sanity, invariant) properties of data.
- In terms of permanent variables only.
- Ensure that initialization makes it true.
- Ensure that $\{\mathrm{MI}\}$ op $\{\mathrm{MI}\}$ (all public methods keep it true)
- Ensure that it is true at any **wait** points.

Condition Variables

`cond c;` — only used within monitor.

`wait(c)` Wait for the condition to be signalled

- Unoccupy monitor
- Wait on c's condition queue.

`signal(c)` Indicate condition is true

Signaling Disciplines

What happens when `signal` is called?

Signal and Wait (SW): Signaller moves to entry queue, signalled process immediately enters monitor.

Signal and Urgent wait (SU): Signaller moves to front of entry queue, signalled process immediately enters monitor.

Signal and Exit (SX): Signaller leaves monitor immediately.

Signal and Continue (SC): Signaller retains occupancy, signalled process is moved to entry queue.

Signal and Wait

Since occupancy of the monitor is passed seamlessly from the signaller to the signallee, any facts about the monitor's data will remain true between the start fo the `signal` and end of the `wait`.

Semantics We assume the following:

- P_c is an assertion associated with condition variable c.
- MI is the monitor invariant.
- L is an assertion that only involves variables local to the process.

signal axiom: $\{\mathrm{P}_c \land \mathrm{MI} \land \mathrm{L}\}\texttt{signal}(c)\{\mathrm{MI} \land \mathrm{L}\}$

wait axiom: $\{\mathrm{MI} \land \mathrm{L}\}\texttt{wait}(c)\{\mathrm{P}_c \land \mathrm{MI} \land \mathrm{L}\}$

- Since occupancy of the monitor passes from the signaller to the waiter without interruption, if P_c is true prior to every $\texttt{signal}(c)$ it will also be true after each $\texttt{wait}(c)$.
- Local variables of the processes are unaffected.
- Since waiting yeilds occupancy, we must ensure MI is true before waiting.
- Since the signaller reenters when the monitor becomes unoccupied, it can assume MI after the signal is complete.
- Since, if the wait queue is empty signalling leaves the monitor unoccupied, we should ensure MI is true before signalling.

	Obligation of montor	Benefit to monitor
$\texttt{signal}(c)$	Ensure $\mathrm{MI} \land \mathrm{P}_c$ before	MI is true after
$\texttt{wait}(c)$	Ensure MI before	$\mathrm{P}_c \land \mathrm{MI}$ is true after

Example: Bounded Buffer

```
monitor Bounded_buffer {
  char buf[n];    # buffer
  int front = 0;  # first full slot
  int rear = 0;   # first empty slot
  int count = 0;  # number of full slots
  cond not_full;  # signaled when count < n
  cond not_empty; # signaled when count > 0
  ## MI: 0 <= front < n /\ 0 <= rear < n /\ 0 <= count <= n /\
  ##     (front+count) % n == rear

  procedure deposit(char data) {
    if (count == n) wait(not_full);
    ## MI /\ count < n
    buf[read] = data;
    rear = (rear+1)% n;
    count++;
    signal(not_empty);
  }
  procedure fetch(char data) {
    if (count == 0) wait(not_empty);
    ## MI /\ count > 0
    result = buf[front];
    front = (front+1)% n;
    count--;
    signal(not_full);
  }
}
```

Signal and Continue

- Signalled process must compete with other processes on the entry queue
- Can't assume that P_c is true after `wait`
- We know that it was "recently" true.
- To ensure that P_c is true we check after awaking:
  `do`$\{$`wait`$(c);\}$`while`$(\mathrm{P}_c);\#\#\{\mathrm{P}_c\}$
  or
  `while`$(\neg\mathrm{P}_c)\{$`wait`$(c);\}\#\#\{\mathrm{P}_c\}$

Example: Bounded Buffer (SC version)

```
monitor Bounded_buffer {
  char buf[n];    # buffer
  int front = 0;  # first full slot
  int rear = 0;   # first empty slot
  int count = 0;  # number of full slots
  cond not_full;  # signaled when count < n
  cond not_empty; # signaled when count > 0
  ## MI: 0 <= front < n /\ 0 <= rear < n /\ 0 <= count <= n /\
  ##     (front+count) % n == rear

  procedure deposit(char data) {
    while (count == n) wait(not_full);
    ## MI /\ count < n
    buf[read] = data;
    rear = (rear+1)% n;
    count++;
    signal(not_empty);
  }
```

```
  procedure fetch(char data) {
    while (count == 0) wait(not_empty);
    ## MI /\ count > 0
    result = buf[front];
    front = (front+1)% n;
    count--;
    signal(not_full);
  }
}
```

Monitors vs. Semaphores

Passing a P operation indicates something happened in the past (i.e., the semaphore was incremented), but does not say much about the current state fo the program.

We're often tempted to write:

```
P0:                              P1:
  ...                              ...
  ## { B }                         P(s)
  V(s)                             ## { B }
  ...                              ...
```

The assertion in P1 is **not** valid — anything could have happened between V and P.

Passing a wait operation means something is true right now — data encapsulation ensures that no other process can clobber the assertion.

```
P0:                              P1:
  ...                              ...
  ## { B }                         wait(c)
  signal(c)                        ## { B }
  ...                              ...
```

Monitors are not more powerful than semaphores, but they're easier to use.

Example: Implementing Semaphores using Monitors

```
monitor Semaphore {
  int s = 0;     # value of the semaphore
  cond not_zero; # signalled when s > 0
  ## MI: s >= 0

  procedure init(int new_s) { ## pre: new_s >= 0
    s := new_s; }

  procedure V() {
    s++;
    signal(not_zero);
  }

  procedure P() {
    if (s == 0) wait(not_zero); ## s > 0
    s--;
  } }
```

Additional operations

signal_all(c)	Move all processes waiting on c to the entry queue. (Only makes sense in SC.)
length(c)	Length of c's queue
empty(c)	True iff c's queue is empty
wait(c, rank)	Priority wait, lowest rank awakened first
minrank(c)	Value of lowest rank waiting

Java Monitors

- Keyword `synchronized` declares object (method, section of code) to be critical section.
- Class with private data and all public methods `synchronized` is a monitor.
- No explicit condition variables — just call `wait()`.
- Signal with `notify()` or `notifyAll()`.
- Only one queue per object.
- Signal and continue discipline.

Example: Bounded Buffer

```
class BoundedBuffer
{
  private char buf[];
  private int front, rear, count, n;
  // ...
  public synchronized void deposit(char data)
    throws InterruptedException
  {
    while (count == n) wait();
    buf[rear] = data;
    rear = (rear+1)%n;
    count++;
    notifyAll();
  }
  public synchronized char fetch()
    throws InterruptedException
```

```
  {
    while (count == 0) wait();
    char result = buf[front];
    front = (front+1)%n;
    count--;
    notifyAll();
    return result;
  }
}
```

Rendezvous: Sleeping Barber Problem

An easygoing town contains a small barber-shop having two doors and a few chairs. Customers enter through one door and leave through the other. Because the shop is small, at most one customer or the barber can move around in it at a time. The barber spends his life serving customers. When none are in the shop, the barber sleeps in the barber's chair. When a customer arrives and finds the barber sleeping the customer awakens the barber, sits in the barber's chair, and sleeps while the barber cuts his hair. If the barber is busy when a customer arrives, the customer goes to sleep in one of the other chairs. After giving a haircut, the barber opens the exit door for the customer and closes it when the customer leaves. If there are waiting customers, the barber then awakens one and waits for the customer to sit in the barber's chair. Otherwise, the barber goes back to sleep until a new customer arrives.

Program Structures: Disk Scheduling

- Contol access (read/write) to a disk head.
- Choose order of access to optimize.

Shortest-seek-time: Always select pending request with closest address. (Unfair)

Elevator Algorithm (SCAN): Move disk heads in only one direction until all requests have been serviced in that direction, then reverse. (Large variance in expected waiting time.)

Circular SCAN: Only service requests in one direction.

Separate Scheduler Monitor

- Users call `request` to access disk, `release` to release it.
- Users do their own access to disk.
- Requires well behaved users.

```
public synchronized void
request(int id, Section s)
  throws InterruptedException
{
  if (granted != 0) {
    pending.insert(
            new Request(id, s));
    while (granted != id) wait();
  } else {
    granted = id;
  }
  cylinder = cyl;
}
```

```
public synchronized void
release(int id)
  throws InterruptedException
{
  if (pending.isEmpty()) {
    granted = 0;
  } else {
    Request toDo = new Request();
    pending.get(cylinder, toDo);
    granted = toDo.id;
  }
  notifyAll();
}
```

Intermediary

Driver process accesses disk, users call disk interface monitor.

```
  public synchronized void diskAccess(int id, Section s)
    throws InterruptedException
  {
    if (granted != 0) {
      pending.insert(new Request(id, s));
      while (granted != id) wait();
    } else {
      granted = id;
    }
    curRequest.set(id, s);
    requestReady = true;
    notifyAll();

    // Wait for access to complete
```

```
    while (!resultsReady) wait();
    resultsReady = false;
    notifyAll();
  }

  public synchronized void getNextRequest(Request res)
    throws InterruptedException
  {
    if (pending.isEmpty()) { // no more to grant
      granted = 0;
    } else {
      Request toDo = new Request();
      pending.get(cylinder, toDo);
      granted = toDo.id;
      notifyAll();
    }
    while (!requestReady) wait();
    res.set(curRequest);
    requestReady = false;
  }

  public synchronized void finishedTransfer(int id)
    throws InterruptedException
  {
    results = id;
    resultsReady = true;
    notifyAll();

    while (resultsReady) wait();
  }
```

Nested Monitor

- Disk transfer monitor called by Disk scheduler monitor.
- Requires *open call* — when monitor A calls monitor B, exclusion monitor A is released.
- (Java uses *closed call* — nested call doesn't release exclusion.)

```
public synchronized void diskAccess(int id, Section s, int length)
  throws InterruptedException
{
  request(id, s);
  disk.access(id, s, length);
  release(id);
}
```