

6 Indirect Communication

- P and V are low level primitives for protecting critical sections and establishing synchronization between tasks.
- Shared variables provide the actual information that is communicated.
- Both of these can be complicated to use and may be incorrectly placed.
- Split-binary semaphores and baton passing are complex.
- Need higher level facilities that perform some of these details automatically.
- Get help from programming-language/compiler.

6.1 Critical Regions

- Declare which variables are to be shared, as in:

```
VAR v : SHARED INTEGER
```

- Access to shared variables is restricted to within a REGION statement, and within the region, mutual exclusion is guaranteed.

Except as otherwise noted, the content of this presentation is licensed under the Creative Commons Attribution 2.5 License.

```

                                semaphore v_lock(1);
REGION v DO                      P(v_lock)
    // critical section          ...      // x = v; (read)  v = y (write)
END REGION                       V(v_lock)

```

- Nesting can result in deadlock.

```
VAR x, y : SHARED INTEGER
```

```

task1                task2
REGION x DO            REGION y DO
    ...
    REGION y DO        REGION x DO
    ...
    END REGION        END REGION
...
END REGION            END REGION

```

- Simultaneous reads are impossible!
- Modify to allow reading of shared variables outside the critical region and modifications in the region.
- Problem: reading partially updated information while a task is updating

the shared variable in the region.

6.2 Conditional Critical Regions

- Introduce a condition that must be true as well as having mutual exclusion.

```

REGION v DO
    AWAIT conditional-expression
    ...
END REGION

```

- E.g. The consumer from the producer-consumer problem.

```

VAR Q : SHARED QUEUE<INT,10>

```

```

REGION Q DO
    AWAIT NOT EMPTY( Q ) buffer not empty
    take an item from the front of the queue
END REGION

```

If the condition is false, the region lock is released and entry is started again (busy waiting).

6.3 Monitor

- A **monitor** is an abstract data type that combines shared data with serialization of its modification.

```
_Monitor name {  
    shared data  
    members that see and modify the data  
};
```

- A **mutex member** (short for mutual-exclusion member) is one that does NOT begin execution if there is another active mutex member.
 - \Rightarrow a call to a mutex member may become blocked waiting entry, and queues of waiting tasks may form.
 - Public member routines of a monitor are implicitly mutex and other kinds of members can be made explicitly mutex (**_Mutex**).
- Basically each monitor has a lock which is P-ed on entry to a monitor member and V-ed on exit.

```
class Mon {  
    int v;  
    uSemaphore MonitorLock(1)  
    public:  
    int x(...) {  
        MonitorLock.P()  
        ...           // int temp = v;  
        MonitorLock.V()  
        return v;     // return temp;  
    }  
}
```

- Unhandled exceptions raised within a monitor always release the implicit monitor locks so the monitor can continue to function.
- Atomic counter using a monitor:

```

_Monitor AtomicCounter {
    int counter;
    public:
        AtomicCounter( int init ) : counter( init ) {}
        int inc() { counter += 1; return counter; }
        int dec() { counter -= 1; return counter; }
};
AtomicCounter a, b, c;
... a.inc(); ...
... b.dec(); ...
... c.inc(); ...

```

- Recursive entry is allowed (owner mutex lock), i.e., one mutex member can call another or itself.
- Destructor is mutex, so ending a block with a monitor or deleting a dynamically allocated monitor, blocks if thread in monitor.

6.4 Scheduling (Synchronization)

- A monitor may want to schedule tasks in an order different from the order in which they arrive.

- There are two techniques: external and internal scheduling.
 - external is scheduling tasks outside the monitor and is accomplished with the accept statement.
 - internal is scheduling tasks inside the monitor and is accomplished using condition variables with signal & wait.

6.4.1 External Scheduling

- The accept statement controls which mutex members can accept calls.
- By preventing certain members from accepting calls at different times, it is possible to control scheduling of tasks.
- E.g. Bounded Buffer

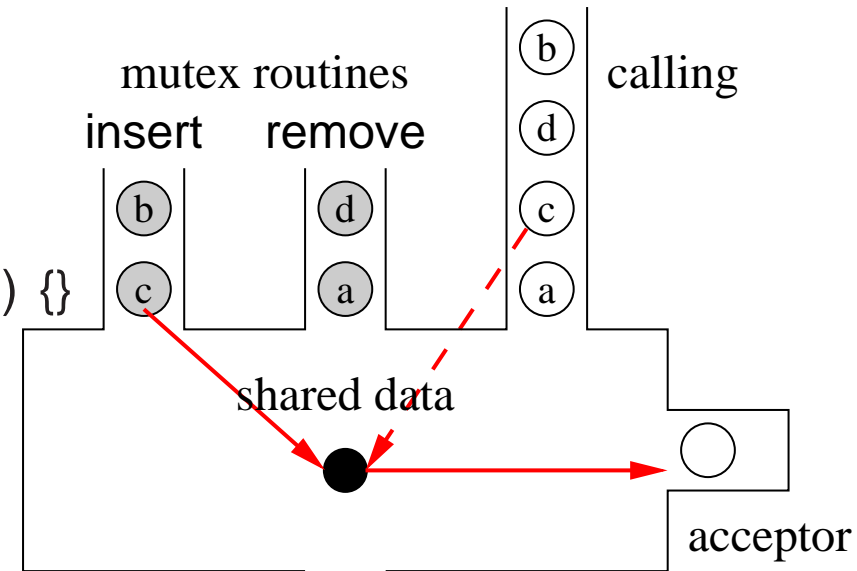
```

_Monitor BoundedBuffer {
    int front, back, count;
    int Elements[20];
public:
    BoundedBuffer() : front(0), back(0), count(0) {}
    _Nomutex int query() { return count; }
    [_Mutex] void insert( int elem );
    [_Mutex] int remove();
};

void BoundedBuffer::insert( int elem ) {
    if ( count == 20 ) _Accept( remove ); // hold insert calls
    Elements[back] = elem;
    back = ( back + 1 ) % 20;
    count += 1;
}

int BoundedBuffer::remove() {
    if ( count == 0 ) _Accept( insert ); // hold remove calls
    int elem = Elements[front];
    front = ( front + 1 ) % 20;
    count -= 1;
    return elem;
}

```



- Queues of tasks form outside the monitor, waiting to be accepted into either insert or remove.
- An acceptor blocks until a call to the specified mutex member(s) occurs.
- Accepted call is executed like a conventional member call.
- When the accepted task exits the mutex member (or blocks), the acceptor continues.
- If the accepted task does an accept, it blocks, forming a stack of blocked acceptors.

6.4.2 Internal Scheduling

- Scheduling among tasks inside the monitor.
- A **condition** is a queue of waiting tasks:

```
uCondition x, y, z[5];
```

- A task waits (blocks) by placing itself on a condition:

```
x.wait();    // wait( mutex, condition )
```

Atomically places the executing task at the back of the condition queue, and allows another task into the monitor by releasing the monitor lock.

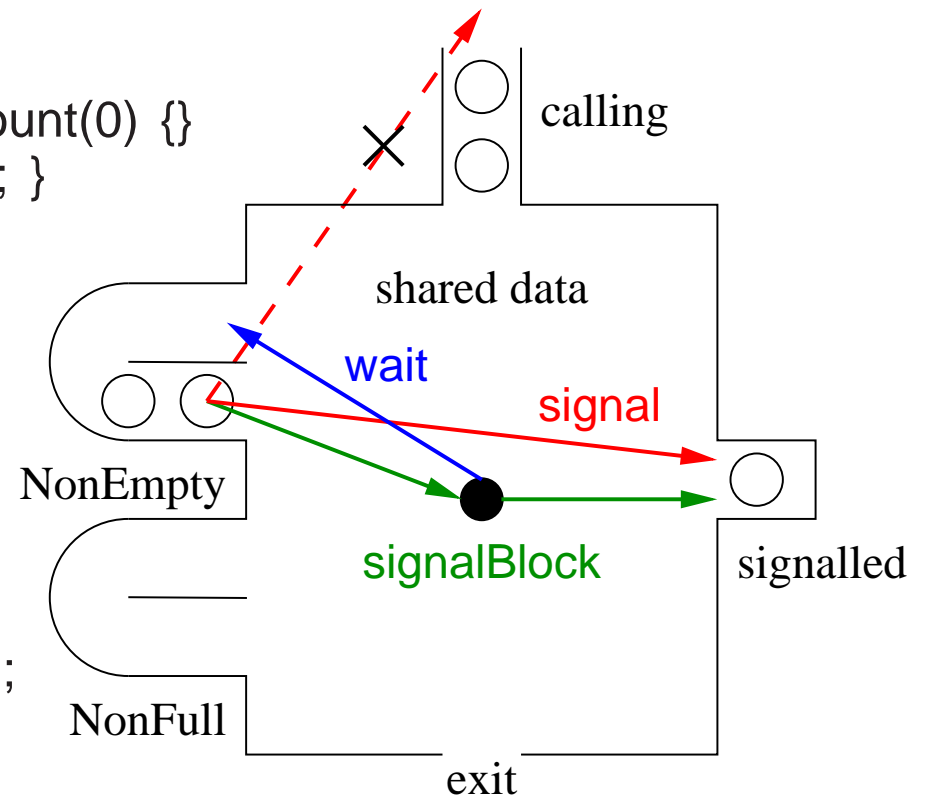
- A task on a condition queue is made ready by signalling the condition:
 `x.signal();`
 This removes a blocked task at the front of the condition queue and makes it ready.
- Like Ving a semaphore, the signaller does not block, so the signalled task must continue waiting until the signaller exits or waits.
- A signal on an empty condition is lost!

- E.g. Bounded Buffer (like binary semaphore solution):

```

_Monitor BoundedBuffer {
    uCondition NonEmpty, NonFull;
    int front, back, count;
    int Elements[20];
public:
    BoundedBuffer() : front(0), back(0), count(0) {}
    _Nomutex int query() { return count; }
    void insert( int elem ) {
        if ( count == 20 ) NonFull.wait();
        Elements[back] = elem;
        back = ( back + 1 ) % 20;
        count += 1;
        NonEmpty.signal();
    }
    int remove() {
        if ( count == 0 ) NonEmpty.wait();
        int elem = Elements[front];
        front = ( front + 1 ) % 20;
        count -= 1;
        NonFull.signal();
        return elem;
    }
};

```



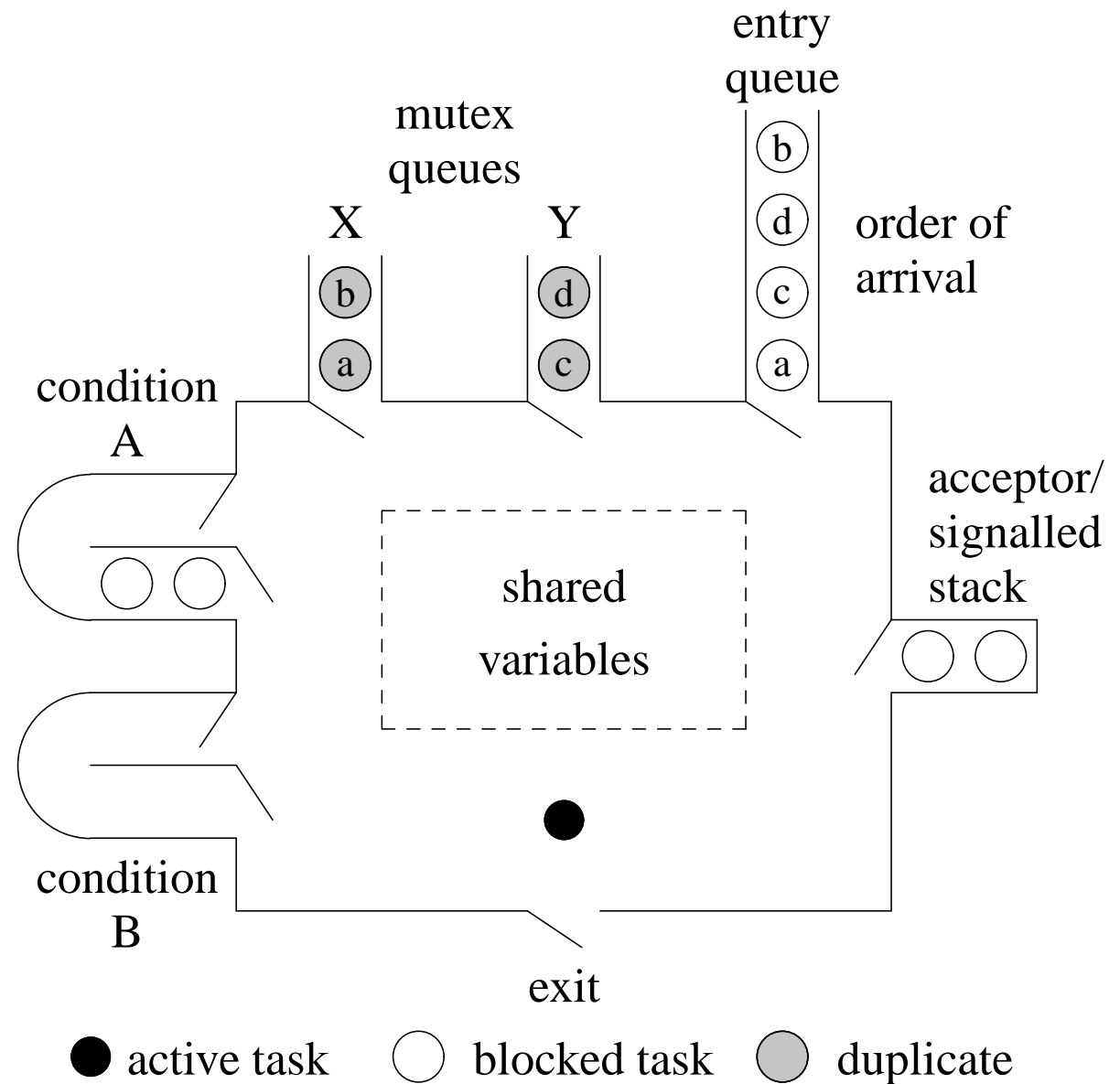
- `wait()` blocks the current thread, and restarts a signalled task or implicitly releases the monitor lock.
- `signal()` unblocks the thread on the front of the condition queue *after* the signaller thread blocks or exits.
- `signalBlock()` unblocks the thread on the front of the condition queue and blocks the signaller thread.
- General Model

```

_Monitor Mon {
    uCondition A, B;

    ...
public:
    int X(...) {...}
    void Y(...) {...}
};

```



- The **entry queue** is a queue of all calling tasks in the order the calls were made to the monitor.

- **explicit scheduling** occurs when:
 - An accept statement blocks the active task on the acceptor stack and makes a task ready from the specified mutex member queue.
 - A signal moves a task from the specified condition to the signalled stack.
 - **implicit scheduling** occurs when a task waits in or exits from a mutex member, and a new task is selected first from the A/S stack, then the entry queue.
- | | |
|-----------------------|------------------------------|
| ● explicit scheduling | internal scheduling (signal) |
| | external scheduling (accept) |
| implicit scheduling | monitor selects (wait/exit) |
- Use external scheduling unless:
 - scheduling depends on member parameter value(s), e.g., compatibility code for dating:
 - a task might be further blocked while in the monitor (e.g., wait for additional resources)

```

_Monitor DatingService {
    uCondition girls[20], boys[20], exchange;
    int girlPhoneNo, boyPhoneNo;
public:
    int girl( int phoneNo, int ccode ) {
        if ( boys[ccode].empty() ) {
            girls[ccode].wait();
            girlPhoneNo = phoneNo;
            exchange.signal();
        } else {
            girlPhoneNo = phoneNo;
            boys[ccode].signal(); // signalBlock() & remove exchange
            exchange.wait();
        }
        return boyPhoneNo;
    }
    int boy( int phoneNo, int ccode ) {
        // same as above, with boy/girl interchanged
    }
};

```

- Use implicit mutex queues to prevent double (queueing) blocking.

6.5 Readers/Writer

- E.g. Readers and Writer Problem (Solution 3)

```

_Monitor ReadersWriter {
    int rcnt, wcnt;
    uCondition readers, writers;
public:
    ReadersWriter() : rcnt(0), wcnt(0) {}
    void startRead() {
        if ( wcnt != 0 || ! writers.empty() ) readers.wait();
        rcnt += 1;
        readers.signal();
    }
    void endRead() {
        rcnt -= 1;
        if ( rcnt == 0 ) writers.signal();
    }
    void startWrite() {
        if ( wcnt !=0 || rcnt != 0 ) writers.wait();
        wcnt = 1;
    }
    void endWrite() {
        wcnt = 0;
        if ( ! readers.empty() ) readers.signal();
        else writers.signal();
    }
};

```

- Can the monitor read/write members perform the reading/writing?
- Has the same protocol problem as P and V.

```
ReadersWriter rw;
```

```
    readers
```

```
    rw.startRead()
```

```
    // read
```

```
    rw.endRead()
```

```
    writers
```

```
    rw.startWrite()
```

```
    // write
```

```
    rw.endWrite()
```

- Alternative interface:

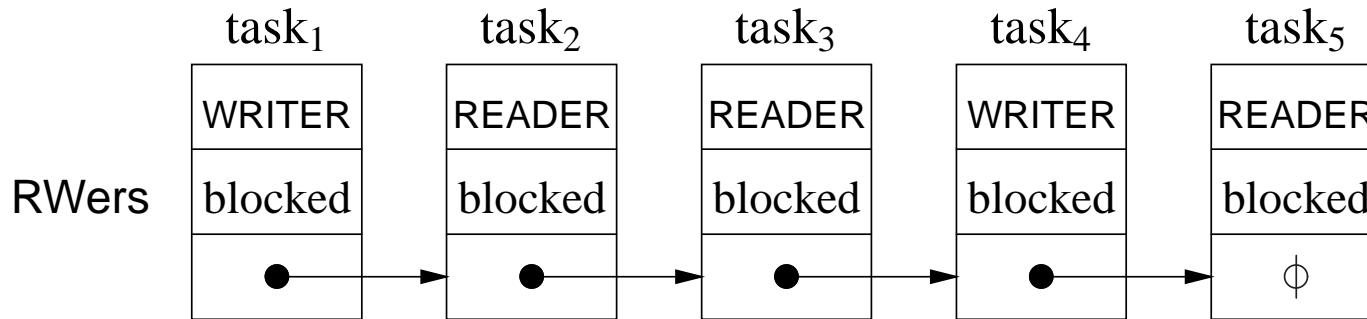
```
_Monitor ReadersWriter {
    _Mutex void startRead();
    _Mutex void endRead();
    _Mutex void startWrite();
    _Mutex void endWrite();
public:
    _Nomutex void read(...) {
        startRead();
        // read
        endRead();
    }
    _Nomutex void write(...) {
        startWrite();
        // write
        endWrite();
    }
}
```

- E.g. Readers and Writer Problem (Solution 4)

```

_Monitor ReadersWriter {
    int rcnt, wcnt;
    uCondition RWers;
    enum RW { READER, WRITER };
public:
    ReadersWriter() : rcnt(0), wcnt(0) {}
    void startRead() {
        if ( wcnt !=0 || !RWers.empty() ) RWers.wait( READER );
        rcnt += 1;
        if ( ! RWers.empty() && RWers.front() == READER ) RWers.signal();
    }
    void endRead() {
        rcnt -= 1;
        if ( rcnt == 0 ) RWers.signal();
    }
    void startWrite() {
        if ( wcnt != 0 || rcnt != 0 ) RWers.wait( WRITER );
        wcnt = 1;
    }
    void endWrite() {
        wcnt = 0;
        RWers.signal();
    }
};

```



- E.g. Readers and Writer Problem (Solution 5)

```

_Monitor ReadersWriter {
    int rcnt, wcnt;
public:
    ReadersWriter() : rcnt(0), wcnt(0) {}
    void endRead() {
        rcnt -= 1;
    }
    void endWrite() {
        wcnt = 0;
    }
    void startRead() {
        if ( wcnt > 0 ) _Accept( endWrite );
        rcnt += 1;
    }
    void startWrite() {
        if ( wcnt > 0 ) { _Accept( endWrite ); } // braces needed
        else while ( rcnt > 0 ) _Accept( endRead );
        wcnt = 1;
    }
}

```

- Why has the order of the member routines changed?

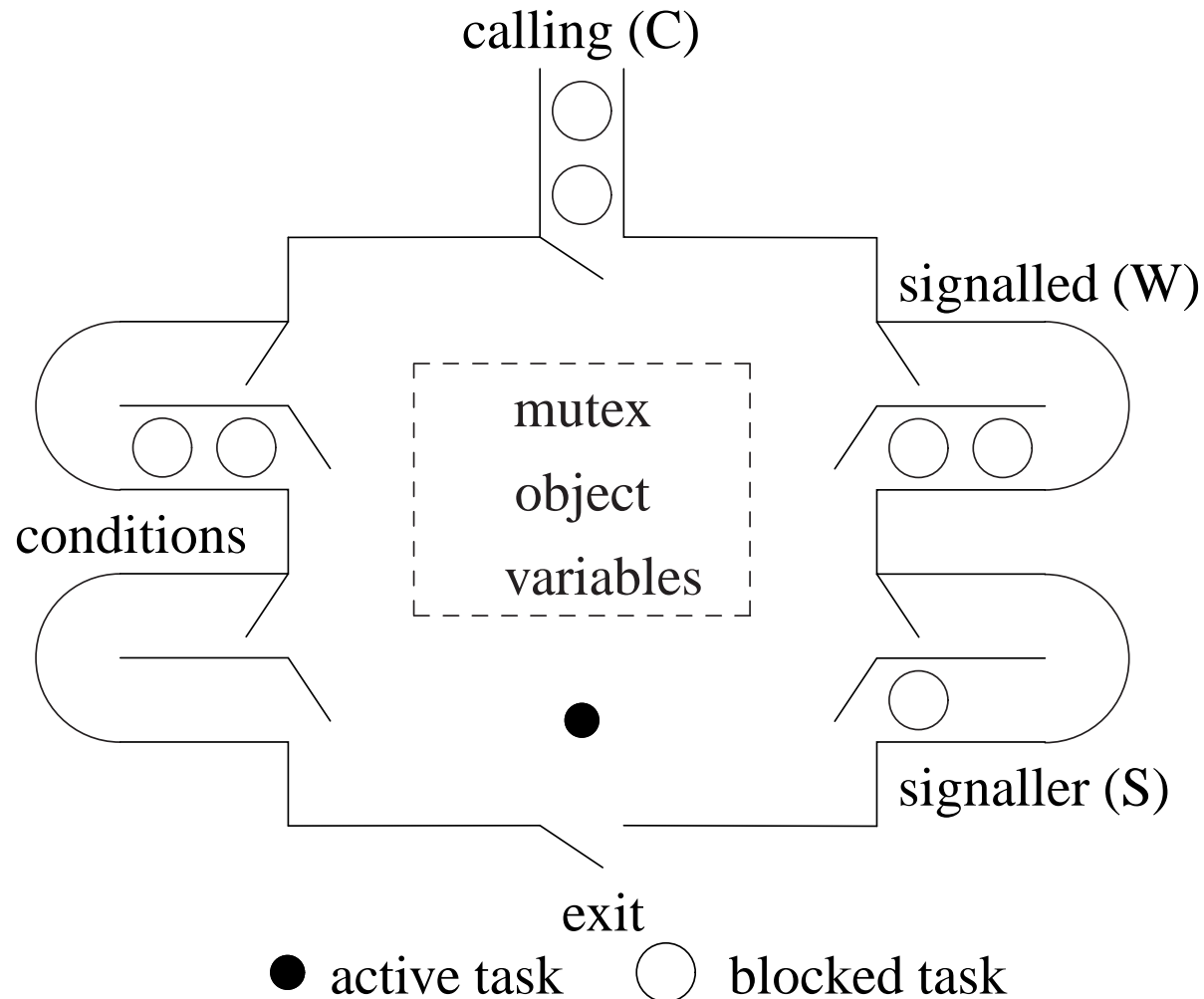
6.6 Condition, Signal, Wait vs. Counting Semaphore, V, P

- There are several important differences between these mechanisms:
 - wait always blocks, P only blocks if semaphore = 0
 - if signal occurs before a wait, it is lost, while a V before a P affects the P
 - multiple Vs may start multiple tasks simultaneously, while multiple signals only start one task at a time because each task must exit serially through the monitor
- It is possible to construct P and V using a monitor:

```
_Monitor semaphore {  
    int sem;  
    uCondition semcond;  
public:  
    semaphore( int cnt = 1 ) : sem( cnt ) {}  
    void P() {  
        if ( sem == 0 ) semcond.wait();  
        sem -= 1;  
    }  
    void V() {  
        sem += 1;  
        semcond.signal();  
    }  
};
```

6.7 Monitor Types

- Monitors are classified by the implicit scheduling (who gets control) of the monitor when a task waits or signals or exits.
- Implicit scheduling can select from the calling (C), signalled (W) and signaller (S) queues.



- Assigning different priorities to these queues creates different monitors (e.g., $C < W < S$).
- Many of the possible orderings can be rejected as they do not produce a useful monitor (e.g., $W < S < C$).

- Implicit Signal

- Monitors either have an explicit signal (statement) or an implicit signal (automatic signal).
- The implicit signal monitor has no condition variables or explicit signal statement.
- Instead, there is a `waitUntil` statement, e.g.:

`waitUntil logical-expression`

- The implicit signal causes a task to wait until the conditional expression is true.

```

_ Monitor BoundedBuffer {
    int front, back, count;
    int Elements[20];
public:
    BoundedBuffer() : front(0), back(0), count(0) {}
    _Nomutex int query() { return count; }
    void insert( int elem ) {
        waitUntil count != 20; // not in uC++
        Elements[back] = elem;
        back = ( back + 1 ) % 20;
        count += 1;
    }
    int remove() {
        waitUntil count != 0; // not in uC++
        int elem = Elements[front];
        front = ( front + 1 ) % 20;
        count -= 1;
        return elem;
    }
};

```

- There is a restricted monitor type that requires that the signaller exit immediately from the monitor (i.e., signal \Rightarrow return), called immediate-return signal.
- Ten kinds of useful monitor are suggested:

signal type	priority	no priority
Blocking	Priority Blocking (Hoare) $C < S < W$ (μ C++ signalBlock)	No Priority Blocking $C = S < W$
Nonblocking	Priority Nonblocking $C < W < S$ (μ C++ signal)	No Priority Nonblocking $C = W < S$ (Java/C#)
Quasi -blocking	Priority Quasi $C < W = S$	No Priority Quasi $C = W = S$
Immediate Return	Priority Return $C < W$	No Priority Return $C = W$
Implicit Signal	Priority Implicit Signal $C < W$	No Priority Implicit Signal $C = W$

- No-priority monitors require the signalled task to recheck the waiting condition in case of a **barging** task.
 \Rightarrow use a **while** loop around a wait instead of an **if**
- Implicit (automatic) signal monitors are good for prototyping but have poor performance.

- Immediate-return monitors are not powerful enough to handle all cases but optimize the most common case of signal before return.
- Quasi-blocking monitors makes cooperation too difficult.
- priority-nonblocking monitor has no barging and optimizes signal before return (supply cooperation).
- priority-blocking monitor has no barging and handles internal cooperation within the monitor (wait for cooperation).
- Java monitor
 - synchronized (wrong name) \Rightarrow mutex
 - only one condition variable per monitor
(new Java library has multiple conditions but are incompatible with language condition)
 - condition operations: wait(), signal(), notifyall()
 - no-priority nonblocking monitor \Rightarrow **while** (! C) wait();
- coroutine monitor
 - coroutine with implicit mutual exclusion on calls to specified member routines:

```

_Mutex _Coroutine C { // _Cormonitor
    void main() {
        ... suspend() ...
        ... suspend() ...
    }
public:
    void m1( ... ) { ... resume(); ... } // mutual exclusion
    void m2( ... ) { ... resume(); ... } // mutual exclusion
    ... // destructor is ALWAYS mutex
};

```

- can use `resume()`, `suspend()`, condition variables (`wait()`, `signal()`, `signalBlock()`) or **`_Accept`** on mutex members.
- coroutine can now be used by multiple threads, e.g., coroutine print-formatter accessed by multiple threads.