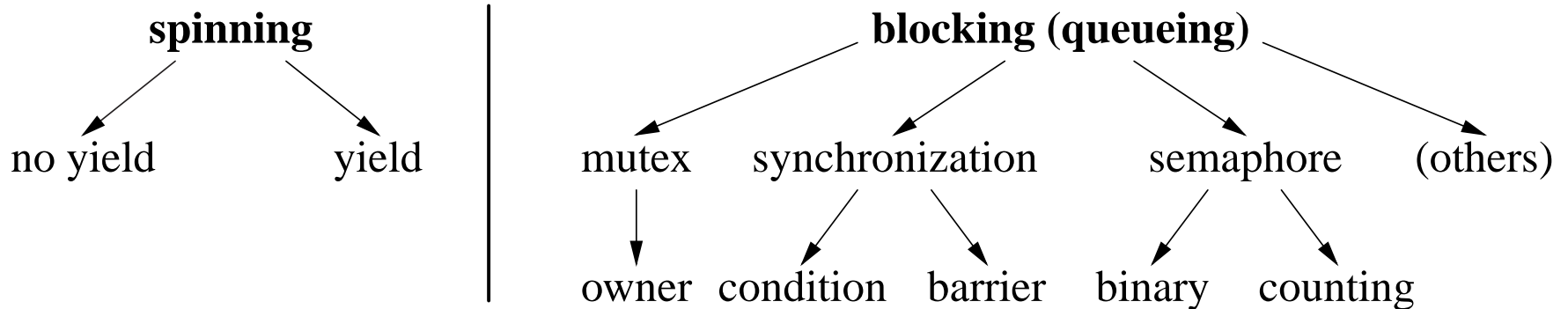


4 Lock Abstraction

- Package software/hardware locking into abstract type for general use.
- Locks are constructed for synchronization or mutual exclusion or both.

4.1 Lock Taxonomy

- Lock implementation is divided into two general categories: spinning and blocking:



- Spinning locks busy wait until an event occurs \Rightarrow task oscillates between ready and running states due to time slicing.

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

- Blocking locks do not busy wait, but block until an event occurs \Rightarrow some *other* mechanism must unblock the waiting task when the event happens.
- Within each category, different kinds of spinning and blocking locks exist.

4.2 Spin Lock

- A **spin lock** is implemented using busy waiting, which spins in a loop checking for an event to occur.
- In the examples so far, if a task is busy waiting, it loops until:
 1. A critical section becomes unlocked or an event happens.
 2. The waiting task is preempted (time-slice ends) and put back on the ready queue.

Hence, the CPU is wasting time constantly checking the event.

- To increase efficiency in the uniprocessor case, a task could explicitly terminate its time-slice and move back to the ready state after the first event check fails.
- The multiprocessor case can have a spinning task terminate its time-slice and move back to the ready state after N checks have failed.

- Some systems allow the duration of spinning to be adjusted, called an **adaptive spin-lock**.
- Depending on how the spin lock is implemented, it may break rule 5, i.e., no bound on service, resulting in starvation of one or more tasks.
- Nevertheless, a spin lock is appropriate and necessary in situations where there is no other work to do.

4.2.1 Spin Lock Details

- $\mu\text{C++}$ provides a non-yielding spin lock, `uSpinLock`, and a yielding spin lock, `uLock`.

```
class uSpinLock {  
    public:  
        uSpinLock(); // open  
        void acquire();  
        bool tryacquire();  
        void release();  
};
```

```
class uLock {  
    public:  
        uLock(unsigned int value = 1);  
        void acquire();  
        bool tryacquire();  
        void release();  
};
```

- Both locks are built directly from an atomic hardware instruction.

- Locks are either closed (0) or opened (1), and waiting tasks compete to acquire the lock after it is released.
- In theory, starvation could occur; in practice, it is seldom a problem.
- `uSpinLock` is non-preemptive, meaning no other task may execute once the lock is acquired, to permit optimizations and additional error checking in the $\mu\text{C++}$ kernel.
- A non-preemptive lock can only be used for mutual exclusion because the task acquiring the lock must release it as no other task may execute.
- Hence, `uSpinLock`'s constructor does not take a starting state and its instances are initialized to open.
- `uLock` does not have the non-preemptive restriction and can be used for both synchronization and mutual exclusion.
- `tryacquire` makes one attempt to acquire the lock, i.e., it does not wait.
- Any number of releases can be performed on a lock as a release only (re)sets the lock to open (1).
- It is *not* meaningful to read or to assign to a lock variable, or copy a lock variable, e.g., pass it as a value parameter.

- synchronization

```

_Task T1 {
    uLock &lk;
    void main() {
        ...
        lk.acquire();
        S2
        ...
    }
public:
    T1( uLock &lk ) : lk(lk) {}
};

void uMain::main() {
    uLock lock(0); // closed
    T1 t1( lock );
    T2 t2( lock );
}

```

```

_Task T2 {
    uLock &lk;
    void main() {
        ...
        S1
        lk.release();
        ...
    }
public:
    T2( uLock &lk ) : lk(lk) {}
};

```

- mutual Exclusion

```

_Task T {
    uLock &lk;
    void main() {
        ...
        lk.acquire();
        // critical section
        lk.release();
        ...
        lk.acquire();
        // critical section
        lk.release();
        ...
    }
public:
    T( uLock &lk ) : lk(lk) {}
};

```

```

void uMain::main() {
    uLock lock(1); // start open
    T t0( lock ), t1( lock );
}

```

- Does this solution afford maximum concurrency?
- Are critical sections independent (disjoint) or dependent?

4.3 Blocking Locks

- Blocking locks reduce busy waiting by making the task releasing the lock do additional work, called **cooperation**.
- Hence, the responsibility for detecting an open lock is not borne solely by an acquiring task, but is shared with the releasing task.

4.3.1 Mutex Lock

- A **mutex lock** is used only for mutual exclusion.
- Tasks waiting on these locks block rather than spin when an event has not occurred.
- Restricting a lock to just mutual exclusion:
 - separates lock usage between synchronization and mutual exclusion
 - permits optimizations and checks as the lock only provides one specialized function
- Mutex locks are divided into two kinds:
 - **single acquisition** : task that acquired the lock (lock owner) cannot acquire it again

- **multiple acquisition** : lock owner can acquire it multiple times, called an **owner lock**
- Single acquisition cannot handle looping or recursion involving a lock:

```
void f() {  
    ...  
    lock.acquire();  
    ... f();      // recursive call within critical section  
    lock.release;  
    ...  
}
```

- Multiple-acquisition locks may require only one release to unlock, or as many releases as acquires.

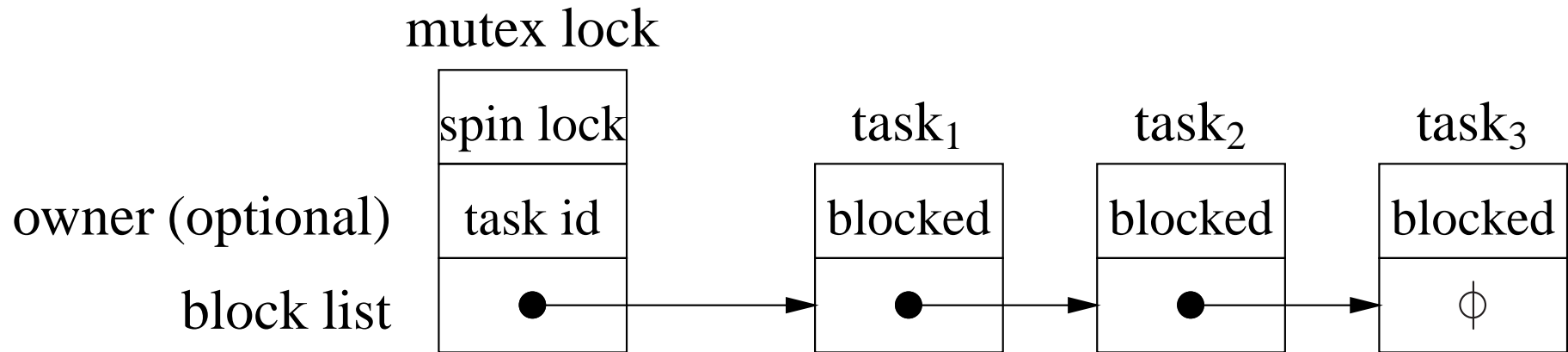
4.3.1.1 Mutex Lock Implementation

- Implementation of a mutex lock requires a:
 - blocking task to link itself onto a list and yield its time slice
 - releasing task to unblock a waiting task (cooperation)
- However, operations like adding and removing a node to/from a list are not atomic \Rightarrow mutual exclusion.

```

class MutexLock {
    spinlock lock;           // nonblocking lock
    queue<Task> blocked;     // blocked tasks
    bool inUse;             // resource being used ?
    Task *owner              // optional
public:
    MutexLock() : inUse( false ), owner( NULL ) {}
    void acquire() {
        lock.acquire();
        if ( inUse
            && owner != thistask() ) { // (optional)
            // add self to lock's blocked list
            lock.release(); // release before blocking
            // yield
            lock.acquire(); // re-acquire lock
        }
        inUse = true;
        owner = thistask(); // set new owner (optional)
        lock.release();
    }
    void release() {
        lock.acquire();
        owner = NULL; // no owner (optional)
        if ( ! blocked.empty() ) {
            // remove task from blocked list and make ready
        } else {
            inUse = false;
        }
        lock.release(); // always release lock
    }
};

```



- Which task is scheduled next from the list of blocked tasks?
- Has the busy wait been removed completely?
- An alternative approach is not to release the spin lock if there is a waiting task, hence:
 - the mutual exclusion is transferred from the releasing task to the unblocking task
 - and the critical section is not bracketed by the spin lock.

4.3.1.2 uOwnerLock Details

- $\mu\text{C++}$ only provides an owner lock, `uOwnerLock`, which subsumes a mutex lock.

```

class uOwnerLock {
  public:
    uOwnerLock();
    void acquire();
    bool tryacquire();
    void release();
};

```

- The operations are the same as for uLock but with blocking instead of spinning for acquire.

4.3.1.3 Stream Locks

- Concurrent use of C++ streams can produce unpredictable and undesirable results; e.g., if two tasks execute:

```

task1 : cout << "abc " << "def " << endl;
task2 : cout << "uvw " << "xyz " << endl;

```

any of the outputs can appear:

abc def	abc uvw xyz	uvw abc xyz def	abuvwc dextf	uvw abc def
uvw xyz	def		yz	xyz

- $\mu\text{C++}$ uses owner locks to provide mutual exclusion for streams: `osacquire` for output streams and `isacquire` for input streams.
- Most common usage is to create as anonymous stream lock for a cascaded I/O expression:

```
task1 : osacquire( cout ) << "abc " << "def " << endl;
task2 : osacquire( cout ) << "uvw " << "xyz " << endl;
```

constraining the output to two different lines in either order:

```
abc def | uvw xyz
uvw xyz | abc def
```

- Multiple I/O statements can be protected using block structure:

```
{ // acquire the lock for stream cout for block duration
  osacquire acq( cout ); // named stream lock
  cout << "abc";
  osacquire( cout ) << "uvw " << "xyz " << endl; // OK?
  cout << "def";
} // implicitly release the lock when "acq" is deallocated
```

4.3.2 Synchronization Lock

- A synchronization lock is used solely for synchronization, for the same reasons as a mutex lock is used only for mutual exclusion.
- Often called a **condition lock**, with wait / signal(notify) for acquire / release.

4.3.2.1 Synchronization Lock Implementation

- Synchronization lock implementation has two forms:
 - external locking** uses an external mutex lock to protect its state,
 - internal locking** uses an internal mutex lock to protect its state.
- Like the mutex lock, the blocking task waits on a list and the releasing task performs the necessary cooperation.
- external locking

```

class SyncLock {
    Task *list;
public:
    SyncLock() : list( NULL ) {}
    void acquire( MutexLock &mutexlock ) {
        // add self to lock's blocked list
        mutexlock.release();
        // yield
    }
    void release() {
        if ( list != NULL ) {
            // remove task from blocked list and make ready
        }
    }
};

```

- The protecting mutex-lock is passed to acquire to be released by the blocking task.
- Alternatively, the protecting mutex-lock is bound at synchronization-lock creation and used implicitly.
- internal locking

```

class SyncLock {
    spinlock mlock;           // nonblocking lock
    Task *list;              // blocked tasks
public:
    SyncLock() : list( NULL ) {}
    void acquire( MutexLock &userlock ) {
        mlock.acquire();
        // add self to task list
        userlock.release();   // release before blocking
        mlock.release();
        // yield
    }
    void release() {
        mlock.acquire();
        if ( list != NULL ) {
            // remove task from blocked list and make ready
        }
        mlock.release();
    }
};

```

- It is still useful (necessary) to be able to unlock an external mutex lock before blocking.
- Otherwise, there is a race between releasing the mutex lock and

blocking on the synchronization lock.

4.3.2.2 uCondLock Details

- μ C++ only provides an internal synchronization lock, uCondLock.

```
class uCondLock {  
    public:  
        uCondLock();  
        bool empty();  
        void wait( uOwnerLock &lock );  
        void signal();  
        void broadcast();  
};
```

- empty() returns **false** if there are tasks blocked on the queue and **true** otherwise.
- wait and signal are used to block a thread on and unblock a thread from the queue of a condition, respectively.
- wait atomically blocks the calling task and releases the argument owner-lock;

- wait re-acquires its argument owner-lock before returning.
- signal releases tasks in FIFO order.
- broadcast is the same as the signal routine, except all waiting tasks are unblocked.

```
bool done = false;
```

```
_Task T1 {
    uOwnerLock &mlk;
    uCondLock &clk;

    void main() {
        mlk.acquire();
        if ( ! done ) clk.wait( mlk );
        else mlk.release();
        S2;
    }
public:
    T1( uOwnerLock &mlk,
        uCondLock &clk ) :
        mlk(mlk), clk(clk) {}
};
```

```
_Task T2 {
    uOwnerLock &mlk;
    uCondLock &clk;

    void main() {
        S1;
        mlk.acquire();
        done = true;
        clk.signal();
        mlk.release();
    }
public:
    T2( uOwnerLock &mlk,
        uCondLock &clk ) :
        mlk(mlk), clk(clk) {}
};
```

```
void uMain::main() {  
    uOwnerLock mlk;  
    uCondLock clk;  
    T1 t1( mlk, clk );  
    T2 t2( mlk, clk );  
}
```

4.3.3 Barrier

- A **barrier** is used to repeatedly coordinate a group of tasks performing a concurrent operation surrounded by a series of sequential operations.
- Hence, a barrier is specifically for synchronization and cannot be used to build mutual exclusion.
- Unlike previous synchronization locks, a barrier retains some state about the events it manages.
- Since manipulation of this state requires mutual exclusion, most barriers use internal locking.
- E.g., 3 tasks must execute a section of code in a particular order: S1, S2 and S3 must *all* execute before S5, S6 and S7.

```

X::main() {
    ...
    S1
    b.block();
    S5
    ...
}
Y::main() {
    ...
    S2
    b.block();
    S6
    ...
}
Z::main() {
    ...
    S3
    b.block();
    S7
    ...
}

void uMain::main() {
    uBarrier b( 3 );
    X x( b );
    Y y( b );
    Z z( b );
}

```

- The barrier is initialized to control 3 tasks and passed to each task by reference (not copied).
- The barrier works by blocking each task at the call to block until all 3 tasks have reached their call to block on barrier b.
- The last task to call block detects that all the tasks have arrived at the barrier, and releases all the tasks (cooperation).
- Hence, all 3 tasks continue execution together after they have all arrived

at the barrier.

- Notice, it is necessary to know in advance the total number of block operations executed before the tasks are released.
- Why not use termination synchronization and create new tasks for each computation?
- The reason is that the creation and deletion of computation tasks may be an unnecessary expense that can be eliminated by using a barrier.

4.3.4 Semaphore

- A **semaphore** lock provides synchronization *and* mutual exclusion (Edsger W. Dijkstra).

```
semaphore lock(0); // 0 => closed, 1 => open, default 1
```

- Like the barrier, a semaphore retains state (counter) to “remember” releases.
- names for acquire and release from Dutch terms
- acquire is P
 - *passeren* \Rightarrow to pass

– prolagen \Rightarrow (proberen) to try (verlagen) to decrease

```
lock.P();           // wait to enter
```

P decrements the semaphore counter and waits if it is zero.

- release is V

– vrijgeven \Rightarrow to release

– verhogen \Rightarrow to increase

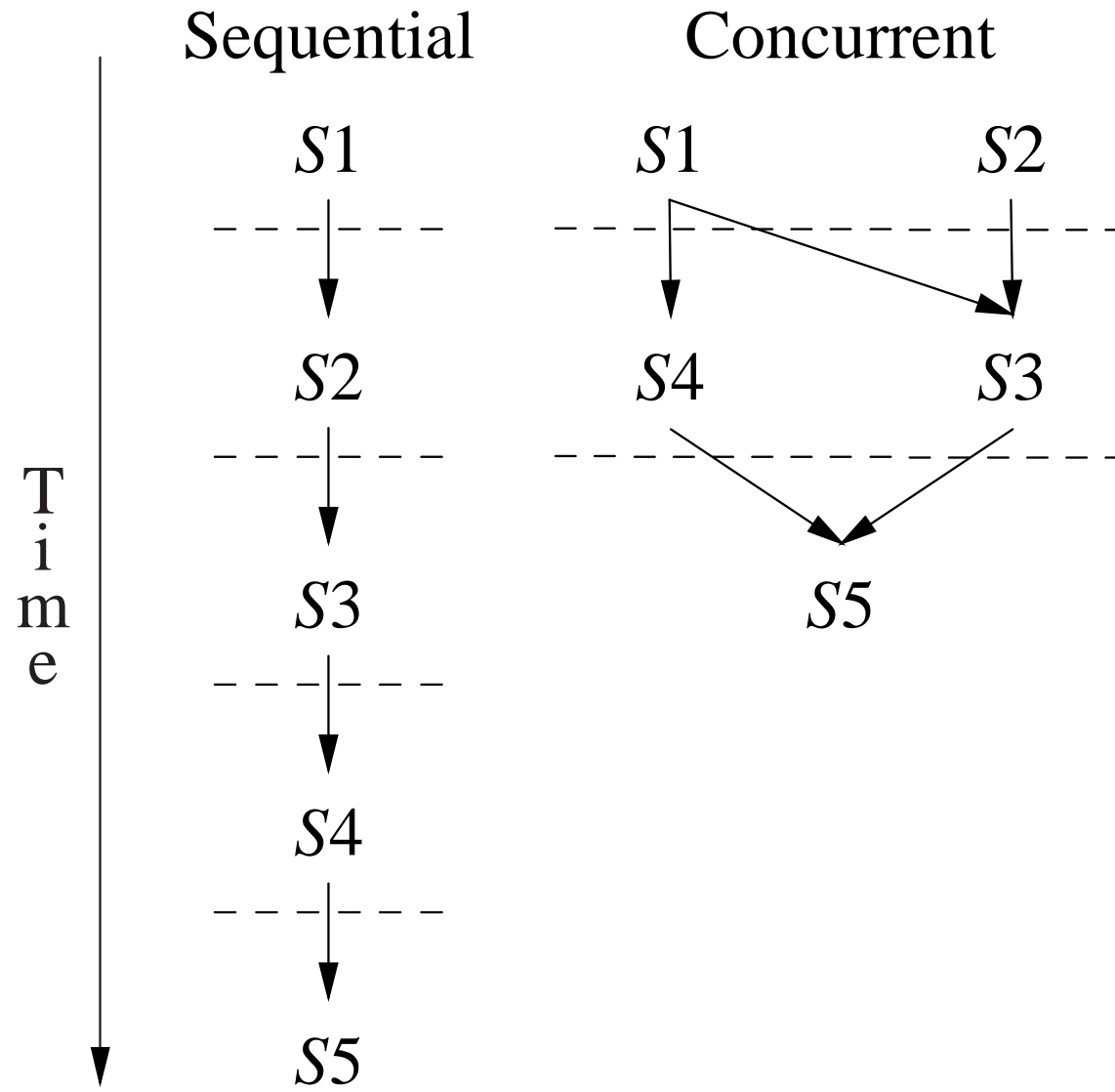
```
lock.V();           // release lock
```

V increases the counter and unblocks a waiting task (if present).

- When the semaphore counter has only two values (0, 1), it is called a **binary semaphore**.
- Semaphore implementation is similar to mutex/synchronization locks, with the counter.
- For example, P and V in conjunction with COBEGIN are as powerful as START and WAIT.
- E.g., execute statements so the result is the same as serial execution but concurrency is maximized.

S1: $a := 1$
S2: $b := 2$
S3: $c := a + b$
S4: $d := 2 * a$
S5: $e := c + d$

- Analyse which data and code depend on each other.
- I.e., statement S1 and S2 are independent \Rightarrow can execute in either order or at the same time.
- Statement S3 is dependent on S1 and S2 because it uses both results.
- Display dependences graphically in a **precedence graph** (which is different from a process graph).

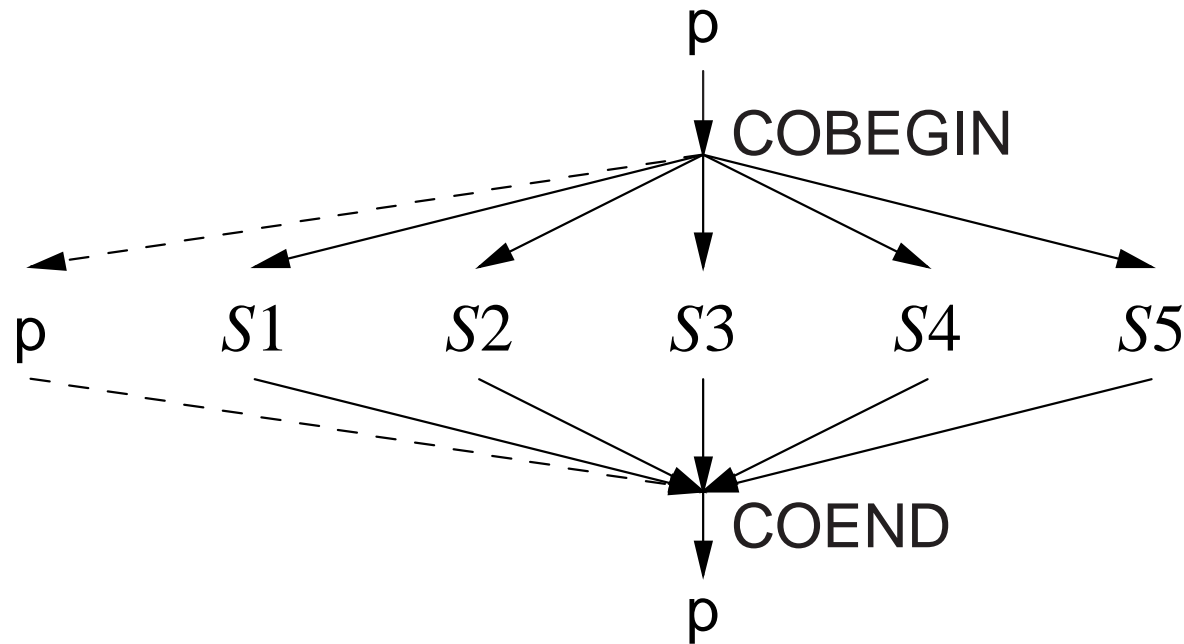


```

semaphore L1(0), L2(0), L3(0), L4(0);
COBEGIN
  BEGIN a := 1; V(L1); END;
  BEGIN b := 2; V(L2); END;
  BEGIN P(L1); P(L2); c := a + b; V(L3); END;
  BEGIN P(L1); d := 2 * a; V(L4); END;
  BEGIN P(L3); P(L4); e := c + d; END;
COEND

```

- process graph



- Does this solution work?

4.3.5 Counting Semaphore

- Augment the definition of P and V to allow a multi-valued semaphore.
- Augment V to allow increasing the counter an arbitrary amount.
- E.g. Three tasks must execute a section of code in a particular order. S2 and S3 only execute after S1 has completed.

```

X::main() {          Y::main() {          Z::main() {
    ...              ...                  S1
    s.P();           s.P();              s.V(); // s.V(2)
    S2               S3                  s.V();
    ...              ...                  ...
}                   }                   }

```

```

void uMain::main() {
    uSemaphore s(0);
    X x(s);
    Y y(s);
    Z z(s);
}

```

- The problem is that you must know in advance the total number of P's

on the semaphore (like a barrier).

4.3.5.1 uSemaphore Details

- μ C++ only provides a counting semaphore, uSemaphore, which subsumes a binary semaphore.

```
class uSemaphore {  
    public:  
        uSemaphore( unsigned int count = 1 );  
        void P();  
        bool TryP();  
        void V( unsigned int times = 1 );  
        int counter() const;  
        bool empty() const;  
};
```

- P decrements the semaphore counter; if the counter is greater than or equal to zero, the calling task continues, otherwise it blocks.
- TryP returns **true** if the semaphore is acquired and **false** otherwise (never blocks).

- V wakes up the task blocked for the longest time if there are tasks blocked on the semaphore and increments the semaphore counter.
- If V is passed a positive integer value, the semaphore is V ed that many times.
- The member routine `counter` returns the value of the semaphore counter:
 - negative means $\text{abs}(N)$ tasks are blocked waiting to acquire the semaphore, and the semaphore is locked;
 - zero means no tasks are waiting to acquire the semaphore, and the semaphore is locked;
 - positive means the semaphore is unlocked and allows N tasks to acquire the semaphore.
- The member routine `empty` returns **false** if there are threads blocked on the semaphore and **true** otherwise.

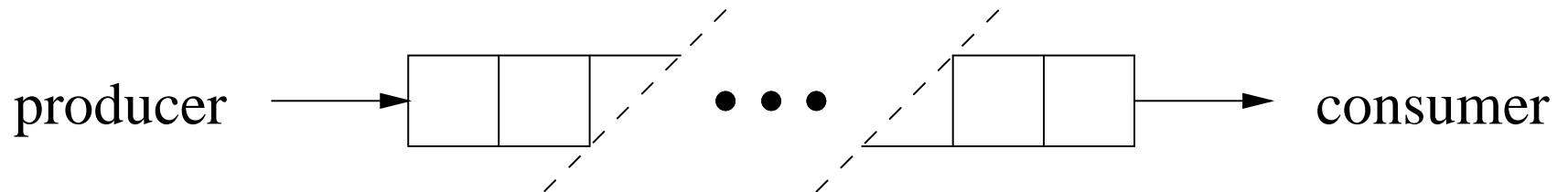
4.4 Semaphore Programming

- Semaphores are used in two distinct ways:
 1. For synchronization, if the semaphore starts at 0 \Rightarrow waiting for an event to occur.

2. For mutual exclusion, if the semaphore starts at 1(N) \Rightarrow used to control a critical section.

- E.g. Unbounded-Buffer Problem

- Two tasks are communicating unidirectionally through a queue of unbounded length.
- Producer adds results to the end of a queue of unbounded length; consumer removes items from the front of the queue unless the queue is empty.



- Because they work at different speeds, the producer may get ahead of the consumer.
- The producer never has to wait as the buffer is infinitely long, but the consumer may have to wait if the buffer is empty.
- The queue is shared between the producer/consumer, and a counting semaphore controls access.

```
#define QueueSize  $\infty$ 

int front = 0, back = 0;
int Elements[QueueSize];
uSemaphore signal(0);

void Producer::main() {
    for (;;) {
        // produce an item
        // add to back of queue
        signal.V();
    }
    // produce a stopping value
}

void Consumer::main() {
    for (;;) {
        signal.P();
        // take an item from the front of the queue
        if ( stopping value ? ) break;
        // process or consume the item
    }
}
```

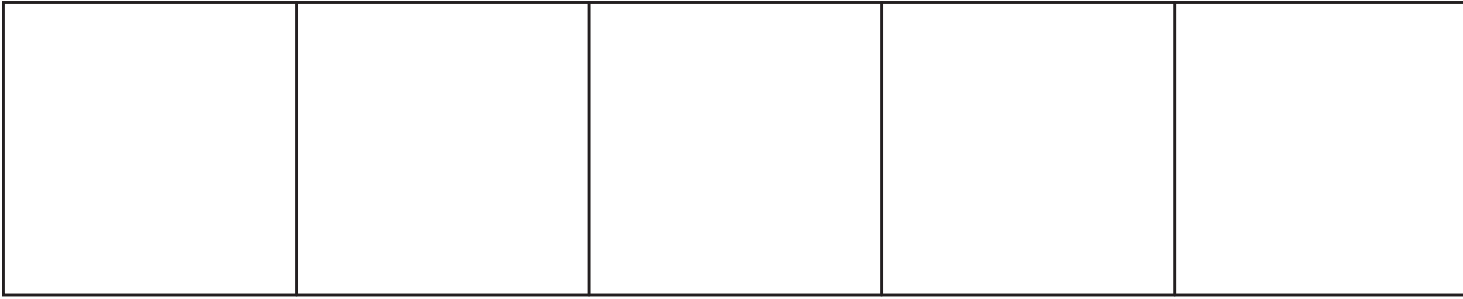
Is there a problem adding and removing items from the shared queue?

- Is the signal semaphore used for mutual exclusion or synchronization?
- E.g. Bounded-Buffer Problem
 - Two tasks are communicating unidirectionally through a buffer of bounded length.
 - Because of the bounded length, the producer may have to wait for the consumer to empty it. As well, the buffer may be empty so the consumer may have to wait until items are produced.
 - Use counting semaphores to account for the finite length of the shared queue.

```
uSemaphore full(0), empty(QueueSize);
void Producer::main() {
    for ( ... ) {
        // produce an item
        empty.P();
        // add to back of queue
        full.V();
    }
    // produce a stopping value
}

void Consumer::main() {
    for ( ... ) {
        full.P();
        // take an item from the front of the queue
        if ( stopping value ? ) break;
        // process or consume the item
        empty.V();
    }
}
```

- Does this produce maximum concurrency?
- Can it handle multiple producers/consumers?



full

0

empty

5

34				
----	--	--	--	--

full

empty

~~0~~

~~5~~

1

4

34	13			
----	----	--	--	--

full

empty

~~0~~

~~5~~

~~1~~

~~4~~

2

3

34	13	9		
----	----	---	--	--

full

empty

~~0~~

~~5~~

~~1~~

~~4~~

~~2~~

~~3~~

3

2

34	13	9	10	
----	----	---	----	--

full

empty

~~0~~~~5~~~~1~~~~4~~~~2~~~~3~~~~3~~~~2~~

4

1

34	13	9	10	-3
----	----	---	----	----

full

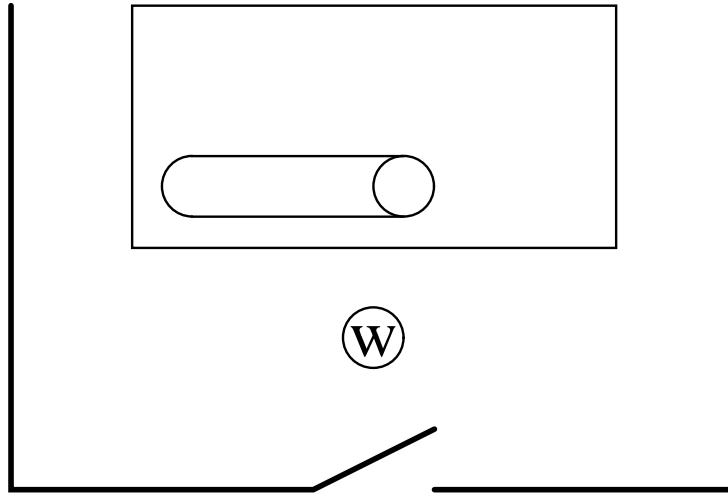
empty

~~0~~~~5~~~~1~~~~4~~~~2~~~~3~~~~3~~~~2~~~~4~~~~1~~

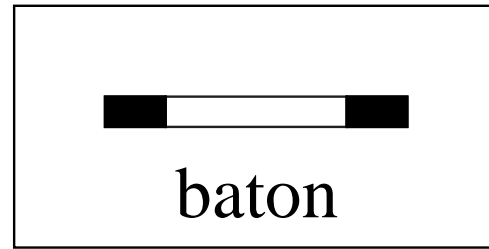
5

0

- E.g. Readers and Writer Problem (Solution 1)
 - Multiple tasks are sharing a resource, and some of them want to read the resource and some want to write or modify the resource.
 - Allow multiple concurrent reader tasks simultaneous access, but serialize access for writer tasks (a writer may read).
 - A **split binary semaphore** is a collection of semaphores where at most one of the collection has the value 1, i.e., the sum of the semaphores is always less than or equal to one.
 - Split binary semaphores can be used to solve complicated mutual-exclusion problems by a technique called **baton passing**.
 - The rules of baton passing are:
 - * there is exactly one baton
 - * nobody moves in the entry/exit code unless they have it
 - * once the baton is released, cannot read/write variables in entry/exit



(r) (r) (r)
Readers



(W) (W) (W)
Writers

(r)
(r)
(W) Arrivers
(r)
(W)

```

uSemaphore entry_q(1);      // split binary semaphore
uSemaphore read_q(0), write_q(0);
int r_cnt = 0, w_cnt = 0;   // auxiliary
int r_del = 0, w_del = 0;

void Reader::main() {
    entry_q.P();             // entry protocol
    if ( w_cnt > 0 ) {
        r_del += 1; entry_q.V(); read_q.P();
    }
    r_cnt += 1;
    if ( r_del > 0 ) {
        r_del -= 1; read_q.V(); // pass baton
    } else {
        entry_q.V();
    }

    yield();                // pretend to read

    entry_q.P();            // exit protocol
    r_cnt -= 1;
    if ( r_cnt == 0 && w_del > 0 ) {
        w_del -= 1; write_q.V(); // pass baton
    } else {
        entry_q.V();
    }
}

```

```

void Writer::main() {
    entry_q.P();           // entry protocol
    if ( r_cnt > 0 || w_cnt > 0 ) {
        w_del += 1; entry_q.V(); write_q.P();
    }
    w_cnt += 1;
    entry_q.V();

    yield();              // pretend to write

    entry_q.P();          // exit protocol
    w_cnt -= 1;
    if ( r_del > 0 ) {
        r_del -= 1; read_q.V(); // pass baton
    } else if ( w_del > 0 ) {
        w_del -= 1; write_q.V(); // pass baton
    } else {
        entry_q.V();
    }
}

```

– Problem: continuous stream of readers \Rightarrow no writer can get in, so starvation.

- E.g. Readers and Writer Problem (Solution 2)

- Give writers priority and make the readers wait.
- Change entry protocol for reader to the following:

```

entry_q.P();                // entry protocol
if ( w_cnt > 0 || w_del > 0 ) {
    r_del += 1; entry_q.V(); read_q.P();
}
r_cnt += 1;
if ( r_del > 0 ) {
    r_del -= 1; read_q.V();
} else {
    entry_q.V();
}

```

- Also, change writer's exit protocol to favour writers:

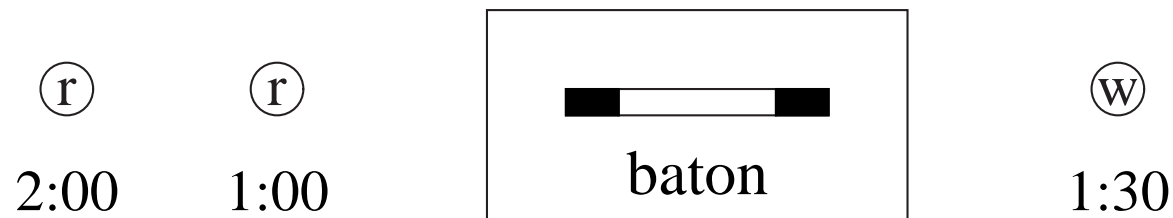
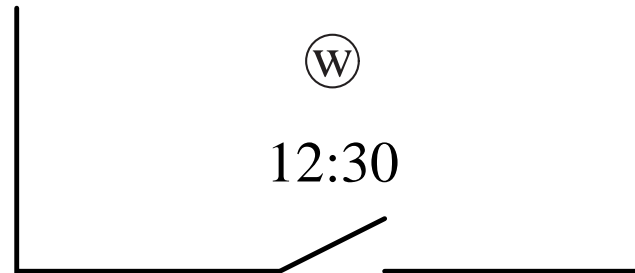
```

entry_q.P();                // exit protocol
w_cnt -= 1;
if ( w_del > 0 ) {          // check writers first
    w_del -= 1; write_q.V();
} else if ( r_del > 0 ) {
    r_del -= 1; read_q.V();
} else {
    entry_q.V();
}

```

– Now readers can starve.

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



- Readers wait if there is a waiting writer, all readers are started after a writer (i.e. alternate group of readers then a writer)
- There is a problem: staleness/freshness.

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

- Service readers and writers in a first-in first-out (FIFO) fashion, but allow multiple concurrent readers.
- Must put readers and writers on same condition variable to maintain temporal order but then lose what kind of task.

- Maintain a shadow list to know the kind of task is waiting at front of condition variable:

semaphore	$t_0 \rightarrow t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow t_4 \rightarrow \dots$
shadow-list	$w \rightarrow r \rightarrow w \rightarrow r \rightarrow r \rightarrow \dots$