

Embedded Software

Thread synchronization

Agenda

- Why synchronization - Shared data problem revisited
- Cases
 - ▶ Sharing data between threads
 - ▶ The Producer / Consumer problem
 - ▶ Park-A-Lot 2000
- Types of synchronization methods

The shared data problem revisited

The shared data problem revisited

```
unsigned int shared;
void taskfunc()
{
    for(;;)
    {
        shared++;           // Increment i, then wait
        sleep(ONE_SECOND);  // 1 second
    }
}

int main()
{
    shared = 0;
    createThread(taskFunc); // Start two identical threads
    createThread(taskFunc); // that run the same function
    for(;;) sleep(ONE_SECOND); // 1 second
}
```

The shared data problem

Task 1

```
LOAD R1, shared
INC R1
STORE shared, R1
```

Task 2

```
LOAD R1, shared
INC R1
STORE shared, R1
```

What value is shared
suppose to have?

Non-interleaved instructions

```
LOAD R1, shared // shared = 0
INC R1
STORE shared, R1 // shared = 1
LOAD R1, shared // shared = 1
INC R1
STORE shared, R1 // shared = 2
```

Which values can shared
in fact have?

Interleaved instructions

```
LOAD R1, shared // shared = 0
LOAD R1, shared // shared = 0
INC R1
STORE shared, R1 // shared = 1
INC R1
STORE shared, R1 // shared = 1
```

The shared data problem

```
Position pos { 10, 20, 30 };
```

Task 1

```
void newPos(Position& ps, float x, float y, float z)
{
    pos.x = x;
    pos.y = y;
    pos.z = z;
}
```

Task 2

```
void printPos(Position& pos)
{
    std::cout << "X: " << pos.x << std::endl;
    std::cout << "Y: " << pos.y << std::endl;
    std::cout << "Z: " << pos.z << std::endl;
}
```

Non-interleaved instructions

```
T1 pos.x = x;
T1 pos.y = y;
T1 pos.z = z;

T2 std::cout << "X: " << pos.x << std::endl;
T2 std::cout << "Y: " << pos.y << std::endl;
T2 std::cout << "Z: " << pos.z << std::endl;
```

newPos called with 11, 22, 33

std::cout prints
X: 10
Y: 20
Z: 30

Interleaved instructions

```
T1 pos.x = x;
T2 std::cout << "X: " << pos.x << std::endl;
T2 std::cout << "Y: " << pos.y << std::endl;
T2 std::cout << "Z: " << pos.z << std::endl;
T1 pos.y = y;
T1 pos.z = z;
```

Pseudo'ish code

std::cout prints
X: 11
Y: 20
Z: 30

Hey WHAT???

The Challenge

- We need a way to ensure that access to **shared** is mutually exclusive
 - ▶ When T_1 is using **shared**, T_2 must be denied access
 - ▶ When T_2 is using **shared**, T_1 must be denied access

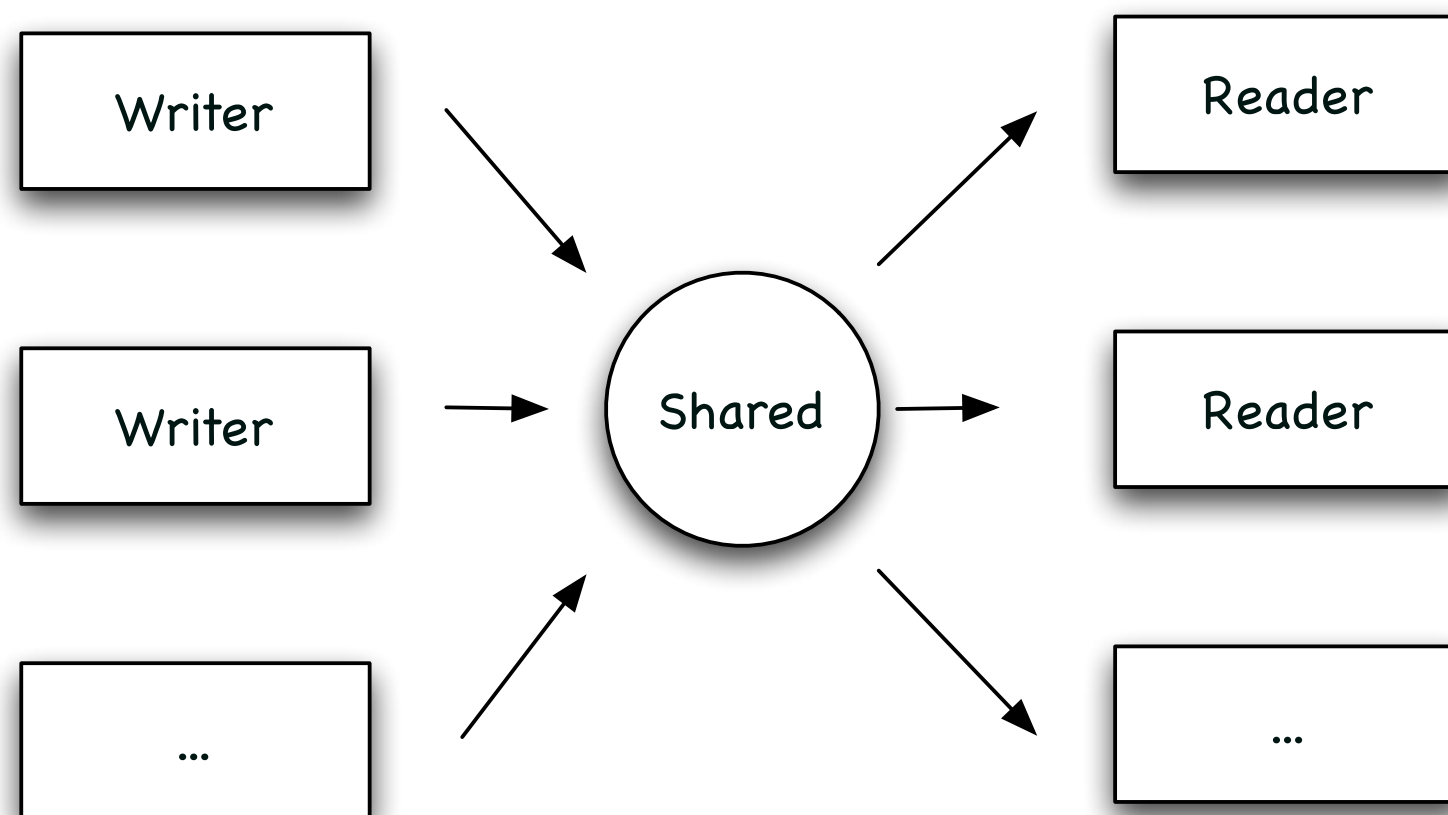
Cases

Case - Ensuring shared data integrity

Case - Ensuring shared data integrity

- Multiple thread entities read/write to a common data structure (maybe as simple as a single variable)

‣ ***Classic problem***



Case - Ensuring shared data integrity

- Problem
 - ▶ Common *shared* variable
- Solution “a mutex”

```
unsigned int shared;
Mutex m = MUTEX_INITIALIZER;

void threadFunc()
{
    for(;;)
    {
        lock(m);
        shared++;
        unlock(m);
        sleep(ONE_SECOND);
    }
}

main()
{
    createThread(threadFunc); // Start two identical threads
    createThread(threadFunc); // that run the same function
    for(;;) sleep(100);
}
```

The mutex

Take the mutex
(or block if needed)

// Increment i...

// then wait 1 second

Release the mutex

The shared data problem - solution

```
Position pos { 10, 20, 30 };  
Mutex m;
```

Task 1

```
void newPos(Position& ps, float x, float y, float z)  
{  
    lock(m);  
    pos.x = x;  
    pos.y = y;  
    pos.z = z;  
    unlock(m);  
}
```

Task 2

```
void printPos(Position& pos)  
{  
    lock(m);  
    std::cout << "X: " << pos.x << std::endl;  
    std::cout << "Y: " << pos.y << std::endl;  
    std::cout << "Z: " << pos.z << std::endl;  
    unlock(m);  
}
```

- Its the programmers responsibility to “lock/unlock” the appropriate places in the code.
- The compiler won't help you
- Ex. - bad solution

```
void newPos(Position& ps, ...)  
{  
    pos.x = x;  
    lock(m);    // Bad position x is not part  
    pos.y = y;  
    pos.z = z;  
    unlock(m);  
}
```

CANNOT be interleaved because of the mutex

```
T1  lock(m);  
T1  pos.x = x;  
T1  pos.y = y;  
T1  pos.z = z;  
T1  unlock(m);  
  
T2  lock(m);  
T2  std::cout << "X: " << pos.x << std::endl;  
T2  std::cout << "Y: " << pos.y << std::endl;  
T2  std::cout << "Z: " << pos.z << std::endl;  
T2  unlock(m);
```

Mutexes

- Mutexes are used to enforce MUTual EXclusion
- Mutexes are owned by one thread at a time - only the “**taker**” can release!
- Two operations on a mutex:
 - **lock(m)**
 - **unlock(m)**

```
lock(Mutex m)
{
    wait until m==1, then m=0;  // ATOMIC operation
}
```

If m=0, calling thread is **BLOCKED** until m==1
If m==1, calling thread proceeds

```
unlock(Mutex m)
{
    m=1;  // ATOMIC operation
}
```

Now m==1 so a **BLOCKED** thread
is made **READY**

Case - Ensuring shared data integrity

- Problem
 - ▶ Common *shared* variable
- Solution “a semaphore”

```
unsigned int shared;
SEM_ID s;

void threadFunc()
{
    for(;;)
    {
        take(s);
        shared++;
        release(s);
        sleep(ONE_SECOND);
    }
}

main()
{
    s = createSem(1);
    createThread(threadFunc); // Start two identical threads
    createThread(threadFunc); // that run the same function
    for(;;) sleep(100);
}
```

Take the semaphore (or block if needed)

// Increment i...

// then wait 1 second

Release the semaphore

Initializing the sem to 1

Semaphores

- Semaphores are used to enforce mutual exclusion or rather signaling
- Semaphores are NOT owned by one thread at a time - “**all**” can release!
- Two operations on a semaphore:
 - ▶ **take(s)** (A.K.A. get(s), pend(s), P(s), wait(s)...)
 - ▶ **release(s)** (A.K.A. give(s), post(s), V(s), signal(s)...)

```
take(Semaphore s)
{
    wait until s>0, then s=s-1;  // ATOMIC operation
}
```

If s=0, calling thread is **BLOCKED** until s>0
If s>0, calling thread proceeds

```
release(Semaphore s)
{
    s=s+1;  // ATOMIC operation
}
```

Now s>0 so a **BLOCKED** thread
is made **READY**

Mutexes & Semaphores: FAQ

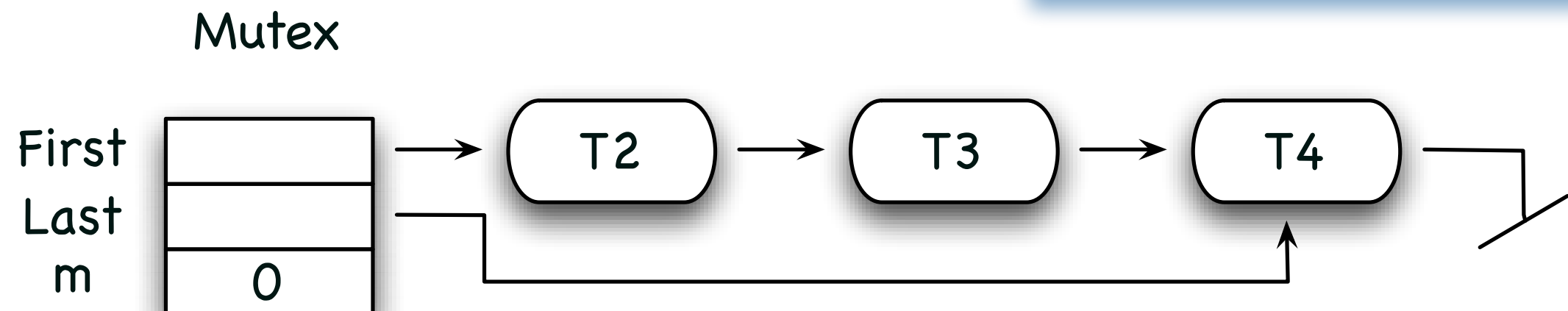
- “Can more than one thread wait for a mutex/semaphore at a time?”
 - ▶ Yes. The threads are queued – see next slide
- “Which of the blocked threads are made ready?”
 - ▶ FIFO: The thread that has waited the longest
 - ▶ Priority: The highest-priority thread

Mutexes & Semaphores: The queue

- Each mutex/semaphore is associated with a waiting queue (FIFO/priority)
- When a thread takes a mutex:
 - ▶ $m=0$: the next incoming thread is added to the mutex's queue
 - ▶ $m=1$: running thread done, next thread activated

```
lock(Mutex m)
{
    wait until m==1, then m=0;
}
```

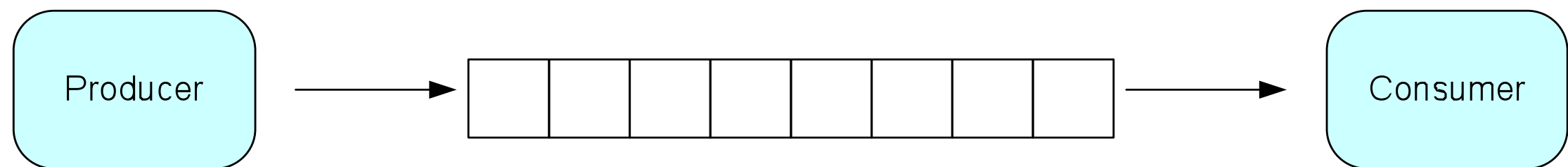
```
unlock(Mutex m)
{
    m=1;
}
```



The Producer-Consumer problem

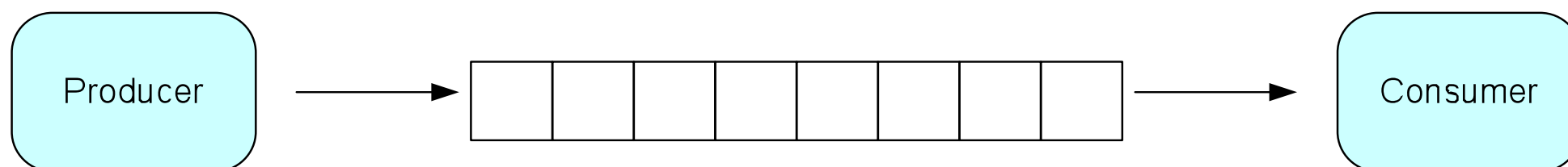
The Producer-Consumer problem

- A producer produces elements and *puts* them in a **buffer**, from which the consumer *retrieves* an element at a time



The producer-consumer problem

- What happens if...
 - ▶ The producer **put()**'s into a full buffer?
 - ▶ The consumer **get()**'s from an empty buffer?
- How can this be handled?
 - ▶ Checking **insert** and **remove** before insertion?
 - ▶ ...and what if the buffer is full/empty? Sleep? How long?
 - ▶ Use 2 counting semaphores!



The producer-consumer problem

- **Explain how to extend this implementation to use 2 counting semaphores to prevent buffer over/underrun**

```
template<typename T>
class Buffer
{
public:
    Buffer(size_t buffserSize) : buffer_(new T[bufferSize]),
        bufferSize_(bufferSize), insert_(0), remove_(0) { }

    void put(const T& x) {
        buffer_[insert_] = x;
        insert_ = (insert_+1)%bufferSize_;
    }

    T get() {
        T tmp = buffer_[remove_];
        remove_ = (remove_+1)%bufferSize_;
        return tmp;
    }

private:
    T* buffer_;
    size_t bufferSize_;
    CountingSemaphore emptySlotsLeft_;
    CountingSemaphore usedSlotsLeft_;
    size_t insert_;
    size_t remove_;
};
```

Implemented as **circular buffer**
insert_ : Insertion pointer
remove_ : Remove pointer

Only **one** producer & **one** consumer

Simplified construction
Possible exceptions are NOT handled

The producer-consumer problem

```
template<typename T>
class Buffer
{
public:
    Buffer(size_t bufferSize) : buffer_(new T[bufferSize]), bufferSize_(bufferSize),
                              insert_(0), remove_(0)
    {
        emptySlotsLeftSem_ = createCountingSem(bufferSize_);
        usedSlotsLeftSem_ = createCountingSem(0);
    }

    void put(const T& x) {
        take(emptySlotsLeftSem_);
        buffer_[insert_] = x;
        insert_ = (insert_ + 1) % bufferSize_;
        release(usedSlotsLeftSem_);
    }

    T get() {
        take(usedSlotsLeftSem_);
        T tmp = buffer_[remove_];
        remove_ = (remove_ + 1) % bufferSize_;
        release(emptySlotsLeftSem_);
    }

private:
    T* buffer_;
    size_t bufferSize_;
    SEM_ID emptySlotsLeftSem_;
    SEM_ID usedSlotsLeftSem_;
    size_t insert_;
    size_t remove_;
};
```

Semaphores are init with size
bufferSize_

The producer thread will automatically block
if buffer is full on **put()**

The consumer thread will be auto-
matically blocked if buffer is empty
on **get()**

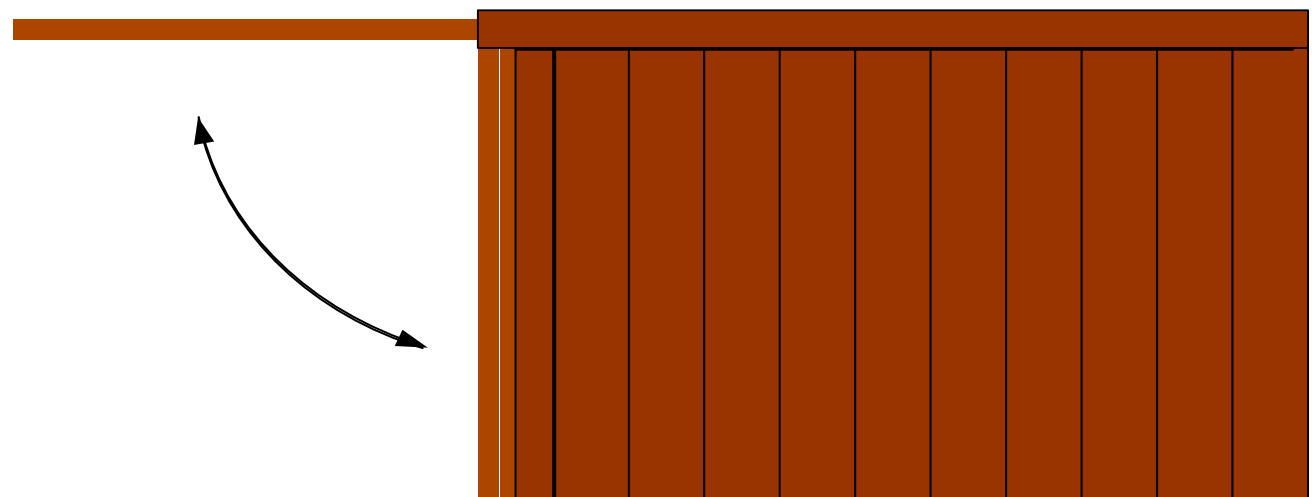
Both consumer and producer threads
will be **automatically unblocked** if data is ready
or buffer not full any longer

*Simplified construction
Possible exceptions are NOT handled*

Case - Park-a-lot 2000

Case - Park-a-lot 2000

- Example: Park-a-lot 2000: An automated car parking system
 - ▶ One thread steers the car
 - ▶ Another thread steers the garage door opener
- ***Coordination how?***



Signalling mechanism - Which?

- We need Conditionals... But How?
 - ▶ Fundamental point is that we have a
 - ▶ Receiver/Waiter who waits on a conditional variable
 - ▶ Sender/Indicator who signals this particular conditional variable at some point

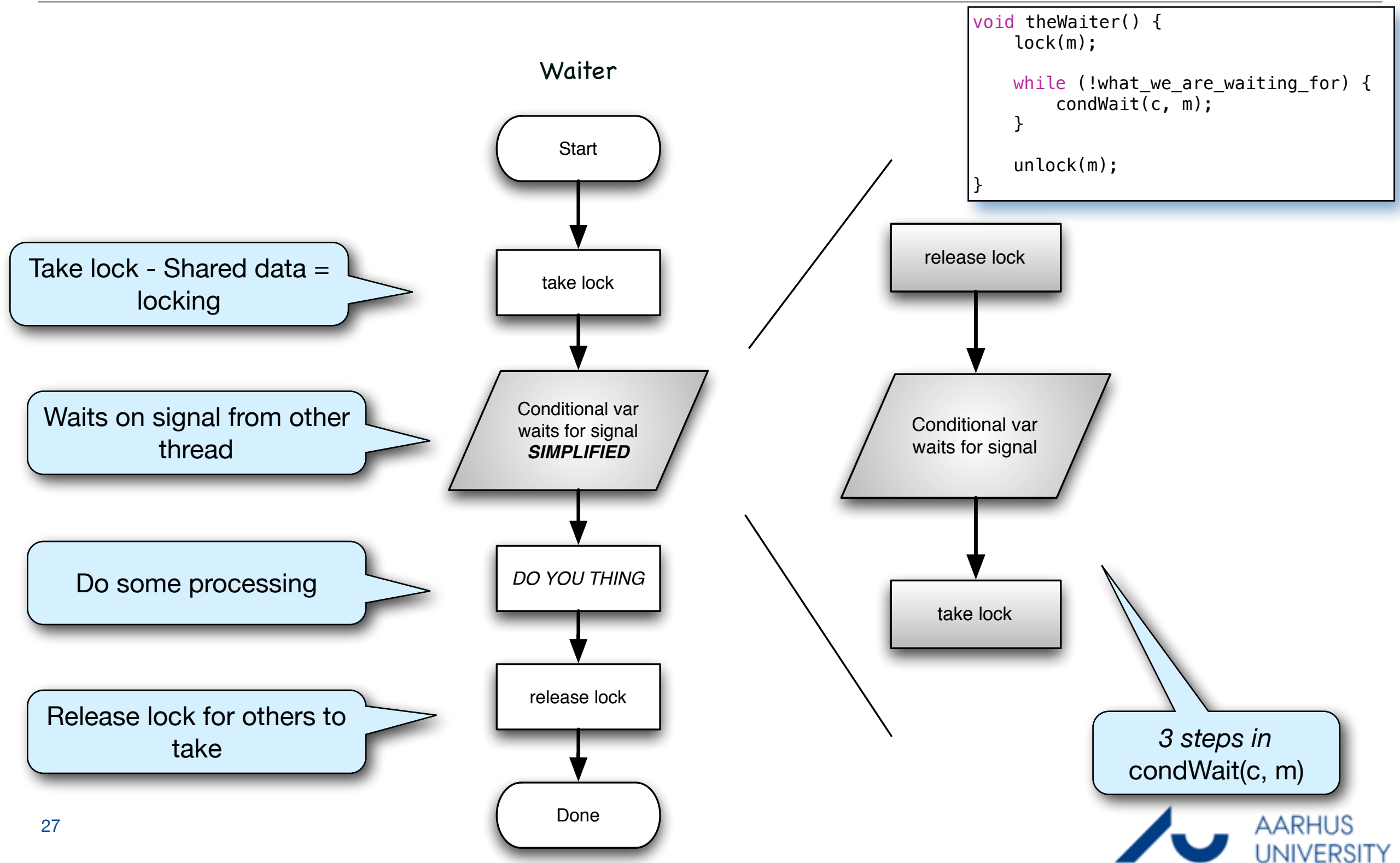
Conditionals - How do you code it?

```
Mutex m;  
Conditional c;
```

```
void theWaiter()  
{  
    lock(m);  
    while (!what_we_are_waiting_for)  
    {  
        condWait(c, m);  
    }  
    unlock(m);  
}
```

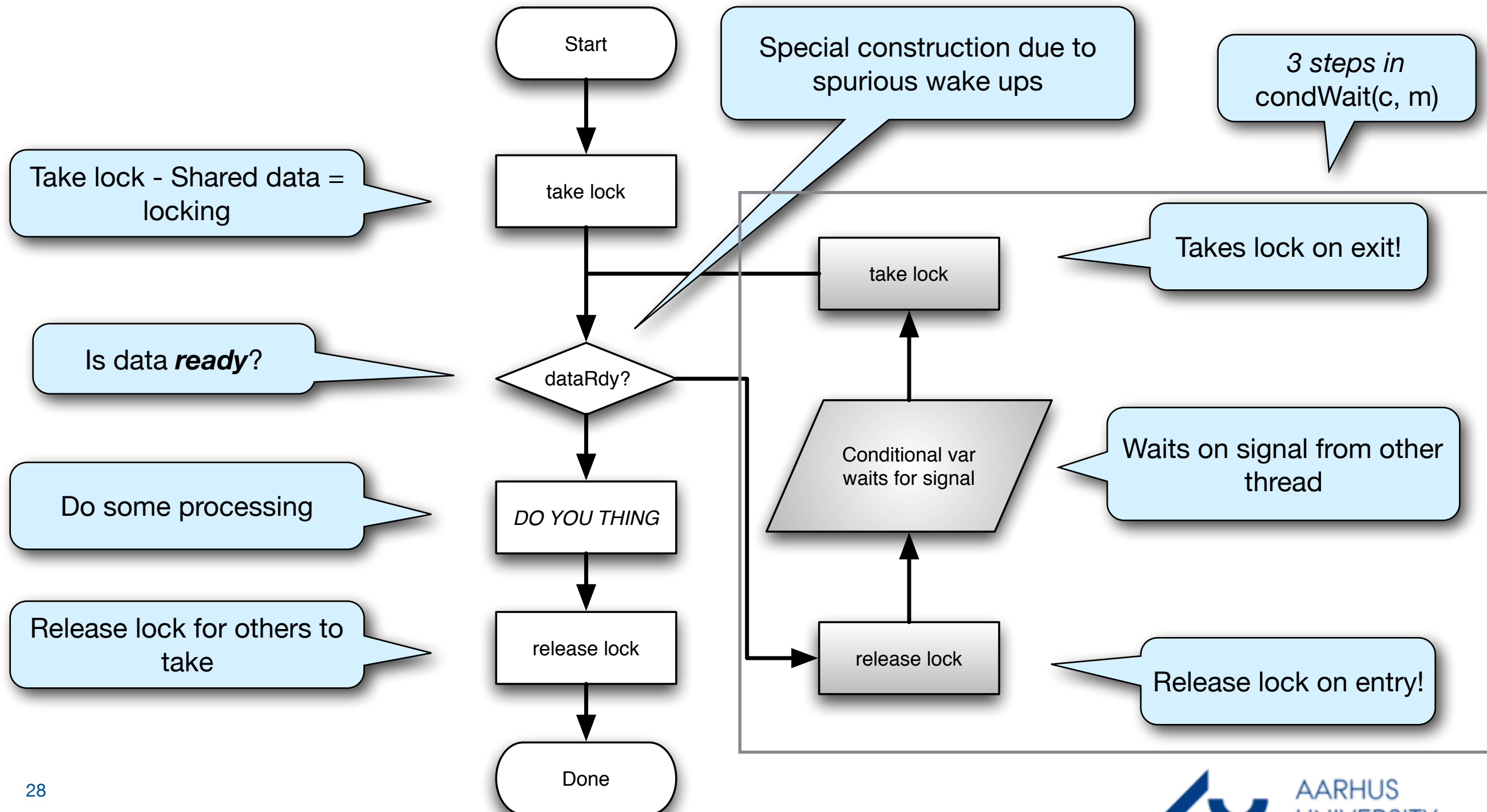
```
void theIndicator()  
{  
    lock(m);  
    // Do something...  
    what_we_are_waiting_for = true;  
    // unlock(m) here - is also okay  
    condSignal(c);  
    unlock(m);  
}
```

How you understand conditionals... - Receiver/Waiter

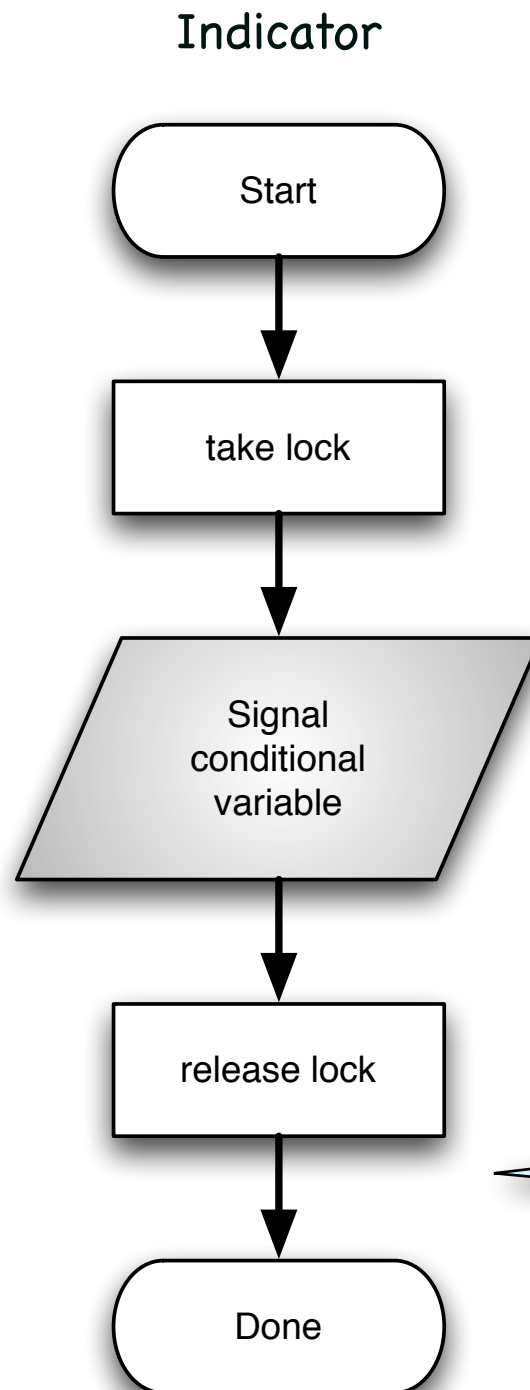


We need Conditionals... - Receiver/Waiter

Waiter



We need Conditionals... - Sender/Indicator



```
void theIndicator()  
{  
    lock(m);  
    // Do something...  
    what_we_are_waiting_for = true;  
    // unlock(m) here - is also okay  
    condSignal(c);  
    unlock(m);  
}
```

When some common data has been updated -> The conditional variable is signaled

Does **NOT** need to be protected. **WHY?**

Park-a-lot 2000 - Feeble attempt

- Our first attempt:
- “hope”...another word system engineers don't like!

```
carDriverThread()  
{  
    driveUpToGarageDoor()  
    sleep(GARAGE_DOOR_OPEN_TIME);  
    // Let's hope the door is open!  
    driveIntoGarage();  
}
```

```
garageDoorControllerThread()  
{  
    openGarageDoor()  
    sleep(CAR_ENTER_GARAGE_TIME);  
    // Let's hope the car is in!  
    closeGarageDoor();  
}
```

- We need to be sure that...
 - ▶ The door is open before we move the car (car sync with garage door)
 - ▶ The car is in before we close the door (garage door sync with car)

Park-a-lot 2000

- Our second attempt: Two-way synchronization

```
carDriverThread()  
{  
    driveUpToGarageDoor():  
    lock(mut);  
    carWaiting = true;  
    condSignal(entry);  
    while(!garageDoorOpen)  
        condWait(entry, mut);  
    driveIntoGarage();  
    carWaiting = false;  
    condSignal(entry);  
    unlock(mut);  
}
```

```
garageDoorControllerThread()  
{  
    lock(mut);  
    while(!carWaiting)  
        condWait(entry, mut);  
    openGarageDoor();  
    garageDoorOpen = true;  
    condSignal(entry);  
    while(carWaiting)  
        condWait(entry, mut);  
    closeGarageDoor();  
    garageDoorOpen = false;  
    unlock(mut);  
}
```

- This works!
 - ▶ 2-way synchronization
 - ▶ All waits are matched with signals

Types of synchronization methods

Types of synchronization methods

- Generally, there are three different types of semaphores for three different purposes:
 - ▶ Mutex: $s=0$ or $s=1$, belongs to one thread at a time
 - ▶ Conditionals Signaling facility used with a mutex
 - ▶ *Read/writable locks* *Multiple readers - Exclusive writer*
 - ▶ Counting semaphore: $s \geq 0$, shared among threads
 - ▶ Binary semaphore: $s=0$ or $s=1$, shared among threads

POSIX Synchronization mechanisms (Not all included)

```
#include<pthread.h>

int pthread_mutex_init(pthread_mutex_t* mutex, pthread_mutex_attr_t *mutexattr);
int pthread_mutex_lock(pthread_mutex_t* mutex);
int pthread_mutex_unlock(pthread_mutex_t* mutex);
int pthread_mutex_destroy(pthread_mutex_t* mutex);

int pthread_rwlock_init(pthread_rwlock_t* mutex, pthread_rwlockattr_t *mutexattr);
int pthread_rwlock_rdlock(pthread_rwlock_t* mutex);
int pthread_rwlock_wrlock(pthread_rwlock_t* mutex);
int pthread_rwlock_unlock(pthread_rwlock_t* mutex);
int pthread_rwlock_destroy(pthread_rwlock_t* mutex);

int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_destroy(pthread_cond_t *cond)
```

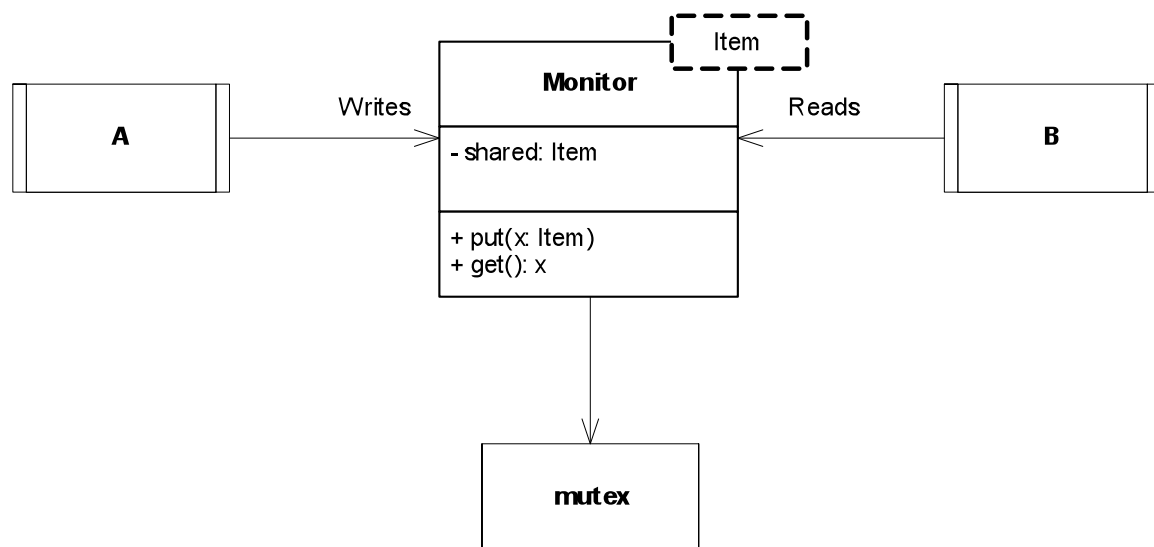
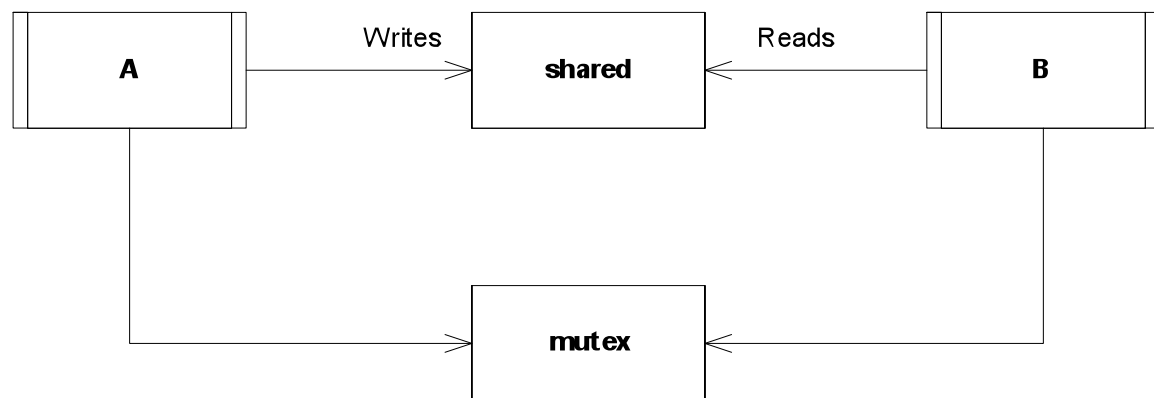
```
#include<semaphore.h>

int sem_init(sem_t* sem,          // Semaphore ID
             int pshared,         // Unsupported. Set to 0
             unsigned int value   // Initial value , >=0
             );

int sem_destroy( sem_t* sem);      // Semaphore ID to destroy
int sem_wait(sem_t* sem);         // Wait for sem
int sem_post(sem_t* sem);         // Post (signal) sem
```

Aids / Tools

The Monitor



- Monitor: A template class
 - ▶ When accessed, the Monitor
 1. takes mutex,
 2. accesses shared,
 3. releases mutex
 - ▶ Responsibility for mutual exclusion: Programmer → monitor
- Any drawbacks/consequences?
 - ▶ Complete copy of shared returned – takes time
 - ▶ Exception between lock() and unlock()?

The Scoped Locking idiom

- A idiom pattern to ensure proper mutex clean-up, even on errors
- The idea: Create an object that automatically takes and releases a mutex at proper times – how?
 - ▶ lock() → constructor
 - ▶ unlock() → destructor
- ***How does this ensure clean-up?***
 - ▶ ***Generalized idiom called RAII - Learn IT!!!***