THREAD

SYNCHRONIZATION I

AARHUS
UNIVERSITY
AARHUS UNIVERSITY SCHOOL OF ENGINEERING

AGENDA

- Synchronisation
- Cases & solutions
  - Sharing data between threads
  - The Producer / Consumer problem
  - Park-A-Lot 2000
- Types of synchronisation methods

SYNCHRONISATION

SYNCHRONISATION

- Race conditions can occur
  - We need tools to handle
- The basic tools are
  - Mutex
  - Semahores
  - Conditionals
- Each *tool* is discussed using cases

CASES & SOLUTIONS

SHARING DATA BETWEEN THREADS

SHARING DATA BETWEEN THREADS

CASE EXAMPLE 1

```
01  unsigned int shared;
02  void taskfunc()
03  {
04    for(;;)
05    {
06      shared++;               // Increment i, then wait
07      sleep(ONE_SECOND);      // 1 second
08    }
09  }
10
11  int main()
12  {
13    shared = 0;
14    createThread(taskFunc);  // Start two identical threads
15    createThread(taskFunc);  // that run the same function
16    for(;;) sleep(ONE_SECOND);      // 1 second
17  }
```

SHARED DATA PROBLEM

- Whats the difference between two execution scenarios?
- Who controls which scenario plays out?
- Which values can `shared` be?

Task 1

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

Task 2

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

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
```

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
```

SHARED DATA PROBLEM - SOLUTION

- Problem
  - Common shared variable
- Solution "a mutex"
  - (or a semaphore)

```
01  unsigned int shared;
02  Mutex m = MUTEX_INITIALIZER;
03
04
05
06  void threadFunc()
07  {
08       for(;;)
09       {
10         lock(m);     // Taking lock
11         shared++;
12         unlock(m);   // Release lock
13         sleep(ONE_SECOND);
14       }
15  }
16
17  main()
18  {
19       createThread(threadFunc);  // Start two identical threads
20       createThread(threadFunc);  // that run the same function
21       for(;;) sleep(100);
22  }
```

SHARING DATA BETWEEN THREADS

CASE EXAMPLE 2

```
01  struct Position
02  {
03    double x, y, z;
04  };
```

SHARING DATA BETWEEN THREADS

CASE EXAMPLE 2

```
01  struct Position
02  {
03    double x, y, z;
04  };
```

```
01  void newPos(Position&pos, float x, float y, float z)
02  {
03    pos.x = x;
04    pos.y = y;
05    pos.z = z;
06  }
```

SHARING DATA BETWEEN THREADS

CASE EXAMPLE 2

```
01  struct Position
02  {
03    double x, y, z;
04  };
```

```
01  void newPos(Position&pos, float x, float y, float z)
02  {
03    pos.x = x;
04    pos.y = y;
05    pos.z = z;
06  }
```

```
01  void printPos(Position& pos)
02  {
03    std::cout << "X: " << pos.x << std::endl;
04    std::cout << "Y: " << pos.y << std::endl;
05    std::cout << "Z: " << pos.z << std::endl;
06  }
```

SHARING DATA BETWEEN THREADS

CASE EXAMPLE 2

```
01  struct Position
02  {
03    double x, y, z;
04  };
```

```
01  void newPos(Position&pos, float x, float y, float z)
02  {
03    pos.x = x;
04    pos.y = y;
05    pos.z = z;
06  }
```

```
01  void printPos(Position& pos)
02  {
03    std::cout << "X: " << pos.x << std::endl;
04    std::cout << "Y: " << pos.y << std::endl;
05    std::cout << "Z: " << pos.z << std::endl;
06  }
```

GLOBAL COMMON VARIABLE

```
01  Position p { 10, 20, 30 };
```

TASK 1 (T1)

```
01  newPos(p, 11, 22, 33);
```

TASK 2 (T2)

```
01  printPos(p);
```

NON-INTERLEAVED

```
01  T1  pos.x = x;
02  T1  pos.y = y;
03  T1  pos.z = z;
04
05  T2  std::cout << "X: " << pos.x << std::endl;
06  T2  std::cout << "Y: " << pos.y << std::endl;
07  T2  std::cout << "Z: " << pos.z << std::endl;
```

INTERLEAVED

```
01  T1  pos.x = x;
02
03  T2  std::cout << "X: " << pos.x << std::endl; // X: 11
04  T2  std::cout << "Y: " << pos.y << std::endl; // Y: 20
05  T2  std::cout << "Z: " << pos.z << std::endl; // Z: 30
06
07  T1  pos.y = y;
08  T1  pos.z = z;
```

SHARING DATA BETWEEN THREADS

SOLUTION

```
01  struct Position
02  {
03    double x, y, z;
04  };
```

SHARING DATA BETWEEN THREADS

SOLUTION

```
01  struct Position
02  {
03    double x, y, z;
04  };
```

```
01  void newPos(Position&pos, float x, float y, float z)
02  {
03    lock(m);
04    pos.x = x;
05    pos.y = y;
06    pos.z = z;
07    unlock(m);
08  }
```

SHARING DATA BETWEEN THREADS

SOLUTION

```
01  struct Position
02  {
03    double x, y, z;
04  };
```

```
01  void newPos(Position&pos, float x, float y, float z)
02  {
03    lock(m);
04    pos.x = x;
05    pos.y = y;
06    pos.z = z;
07    unlock(m);
08  }
```

```
01  void printPos(Position& pos)
02  {
03    lock(m);
04    std::cout << "X: " << pos.x << std::endl;
05    std::cout << "Y: " << pos.y << std::endl;
06    std::cout << "Z: " << pos.z << std::endl;
07    unlock(m);
08  }
```

SHARING DATA BETWEEN THREADS

SOLUTION

```
01  struct Position
02  {
03    double x, y, z;
04  };
```

```
01  void newPos(Position&pos, float x, float y, float z)
02  {
03    lock(m);
04    pos.x = x;
05    pos.y = y;
06    pos.z = z;
07    unlock(m);
08  }
```

```
01  void printPos(Position& pos)
02  {
03    lock(m);
04    std::cout << "X: " << pos.x << std::endl;
05    std::cout << "Y: " << pos.y << std::endl;
06    std::cout << "Z: " << pos.z << std::endl;
07    unlock(m);
08  }
```

GLOBAL COMMON VARIABLE

```
01  Position p { 10, 20, 30 };
02  Mutex m;
```

TASK 1 (T1)

```
01  newPos(p, 11, 22, 33);
```

TASK 2 (T2)

```
01  printPos(p);
```

CANNOT BE INTERLEAVED DUE TO MUTEX

```
01  T1  lock(m);
02  T1  pos.x = x;
03  T1  pos.y = y;
04  T1  pos.z = z;
05  T1  unlock(m);
06
07  T2  lock(m);
08  T2  std::cout << "X: " << pos.x << std::endl;
09  T2  std::cout << "Y: " << pos.y << std::endl;
10  T2  std::cout << "Z: " << pos.z << std::endl;
11  T2  unlock(m);
```

SHARING DATA BETWEEN THREADS

NOT A SOLUTION

- Its the programmers responsibility to “lock/unlock” the appropriate places in the code.
  - Called critical sections
- The compiler won’t help you!!!
- Example is NOT solution

```
01  void newPos(Position& ps, ...)
02  {
03    pos.x = x;
04    lock(m);     // Bad position x is not part
05    pos.y = y;
06    pos.z = z;
07    unlock(m);
08  }
```

USEABLE TOOL - MUTEX

- 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)

```
01  lock(Mutex m)
02  {
03    wait until m==1, then m=0; /* ATOMIC operation */
04  }
```

```
01  unlock(Mutex m)
02  {
03    m=1;   /* ATOMIC operation */
04  }
```

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

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

USEABLE TOOL - SEMAPHORE

- Semaphores are used to signal, but can be used to enforce mutual exclusion
- 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)...)

```
01  take(Semaphore s)
02  {
03    wait until s>0, then s=s-1;/* ATOMIC operation */
04  }
```

```
01  release(Semaphore s)
02  {
03    s=s+1;  /* ATOMIC operation */
04  }
```

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

- 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
- "Which of the blocked threads are made ready?"
  - Indeterminate: Depends on native system...
  - Priority: The highest-priority thread

CASES & SOLUTIONS

THE PRODUCER / CONSUMER PROBLEM

CASES & SOLUTIONS

THE PRODUCER / CONSUMER PROBLEM

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

CASES & SOLUTIONS

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?

CASES & SOLUTIONS

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?

A solution

Use 2 counting semaphores!

FIRST CUT - WITHOUT SEMAPHORE

- ImpIemented as a *Circular buffer*
  - `insert_` - Insertion pointer
  - `remove_` - Remove pointer
- Design focus
  - Only *1* producer & *1* consumer

```
01  class Buffer
02  {
03  public:
04    Buffer(size_t bufferSize) :
05      buffer_(new uint8_t[bufferSize]),
06      bufferSize_(bufferSize),
07      insert_(0), remove_(0)
08    {
09    }
10
11    void put(uint8_t x)  {
12      buffer_[insert_] = x;
13      insert_ = (insert_+1)%bufferSize_; // Using modulus to achieve what?
14    }
15
16    uint8_t get()  {
17      uint8_t tmp = buffer_[remove_];
18      remove_ = (remove_+1)%bufferSize_;
19    }
20
21  private:
22    uint8_t* buffer_;
23    size_t   bufferSize_, insert_, remove_;
24    SEM_ID   emptySlotsLeftSem_;
25    SEM_ID   usedSlotsLeftSem_;
26  };
```

A SOLUTION

- Create two semaphores
  1. number of empty slots
  2. used slots

```
01  class Buffer
02  {
03  public:
04    Buffer(size_t bufferSize) :
05      buffer_(new uint8_t[bufferSize]),
06      bufferSize_(bufferSize),
07      insert_(0), remove_(0)
08    {
09      emptySlotsLeftSem_ = createCountingSem(bufferSize_);
10      usedSlotsLeftSem_ = createCountingSem(0);
11    }
12
13    void put(uint8_t x)  {
14      take(emptySlotsLeftSem_);
15      buffer_[insert_] = x;
16      insert_ = (insert_+1)%bufferSize_;
17      release(usedSlotsLeftSem_);
18    }
19
20    uint8_t get()  {
21      take(usedSlotsLeftSem_);
22      uint8_t tmp = buffer_[remove_];
23      remove_ = (remove_+1)%bufferSize_;
24      release(emptySlotsLeftSem_);
25    }
26
27  private:
28    uint8_t* buffer_;
29    size_t   bufferSize_, insert_, remove_;
30    SEM_ID   emptySlotsLeftSem_;
31    SEM_ID   usedSlotsLeftSem_;
32  };
```

A SOLUTION

- Create two semaphores
  1. number of empty slots
  2. used slots
- **`put()`** adds an uint8_t
  - takes a semaphore
    - from *empty slots*
  - release a semaphore
    - to *used slots*

```
01  class Buffer
02  {
03  public:
04    Buffer(size_t bufferSize) :
05      buffer_(new uint8_t[bufferSize]),
06      bufferSize_(bufferSize),
07      insert_(0), remove_(0)
08    {
09      emptySlotsLeftSem_ = createCountingSem(bufferSize_);
10      usedSlotsLeftSem_ = createCountingSem(0);
11    }
12
13    void put(uint8_t x)  {
14      take(emptySlotsLeftSem_);
15      buffer_[insert_] = x;
16      insert_ = (insert_+1)%bufferSize_;
17      release(usedSlotsLeftSem_);
18    }
19
20    uint8_t get()  {
21      take(usedSlotsLeftSem_);
22      uint8_t tmp = buffer_[remove_];
23      remove_ = (remove_+1)%bufferSize_;
24      release(emptySlotsLeftSem_);
25    }
26
27  private:
28    uint8_t* buffer_;
29    size_t   bufferSize_, insert_, remove_;
30    SEM_ID   emptySlotsLeftSem_;
31    SEM_ID   usedSlotsLeftSem_;
32  };
```

A SOLUTION

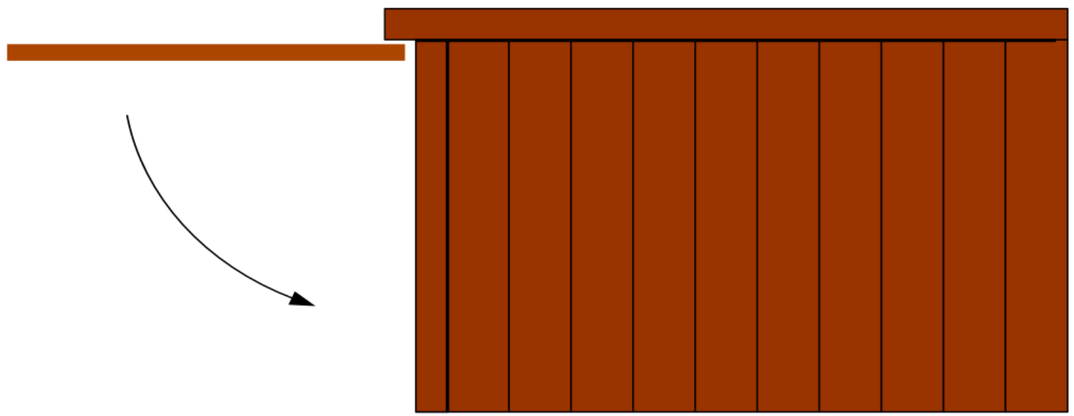
- Create two semaphores
  1. number of empty slots
  2. used slots
- **`put()`** adds an uint8_t
  - takes a semaphore
    - from *empty slots*
  - release a semaphore
    - to *used slots*
- **`get()`** gets an uint8_t
  - takes a semaphore
    - from *used slots*
  - release a semaphore
    - to *empty slots*

```
01  class Buffer
02  {
03  public:
04    Buffer(size_t bufferSize) :
05      buffer_(new uint8_t[bufferSize]),
06      bufferSize_(bufferSize),
07      insert_(0), remove_(0)
08    {
09      emptySlotsLeftSem_ = createCountingSem(bufferSize_);
10      usedSlotsLeftSem_ = createCountingSem(0);
11    }
12
13    void put(uint8_t x)  {
14      take(emptySlotsLeftSem_);
15      buffer_[insert_] = x;
16      insert_ = (insert_+1)%bufferSize_;
17      release(usedSlotsLeftSem_);
18    }
19
20    uint8_t get()  {
21      take(usedSlotsLeftSem_);
22      uint8_t tmp = buffer_[remove_];
23      remove_ = (remove_+1)%bufferSize_;
24      release(emptySlotsLeftSem_);
25    }
26
27  private:
28    uint8_t* buffer_;
29    size_t   bufferSize_, insert_, remove_;
30    SEM_ID   emptySlotsLeftSem_;
31    SEM_ID   usedSlotsLeftSem_;
32  };
```

CASES & SOLUTIONS

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 open/close mechanism
- *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

LETS SEE SOME PSEUDO CODE :-)

Define a mutex, conditional and flag

```
01  Mutex     m;
02  Condition c;
03
04  bool whatWeAreWaitingFor = false;
```

THE WAITER

```
01  void theWaiter()
02  {
03    lock(m);
04
05    while (!whatWeAreWaitingFor)
06    {
07        condWait(c, m);
08    }
09    whatWeAreWaitingFor = false;
10
11    unlock(m);
12  }
```

THE SIGNALER

```
01  void theIndicator()
02  {
03    lock(m);
04    /* Do something... */
05    whatWeAreWaitingFor = true;
06
07    condSignal(c);
08    unlock(m);
09  }
```

LETS SEE SOME PSEUDO CODE :-)

Define a mutex, conditional and flag

```
01  Mutex     m;
02  Condition c;
03
04  bool whatWeAreWaitingFor = false;
```

THE WAITER

```
01  void theWaiter()
02  {
03    lock(m);
04
05    while (!whatWeAreWaitingFor)
06    {
07        condWait(c, m);
08    }
09    whatWeAreWaitingFor = false;
10
11    unlock(m);
12  }
```

THE SIGNALER

```
01  void theIndicator()
02  {
03    lock(m);
04    /* Do something... */
05    whatWeAreWaitingFor = true;
06
07    condSignal(c);
08    unlock(m);
09  }
```

HEY WAIT, WHY THE LOOP IN `theWaiter()`???

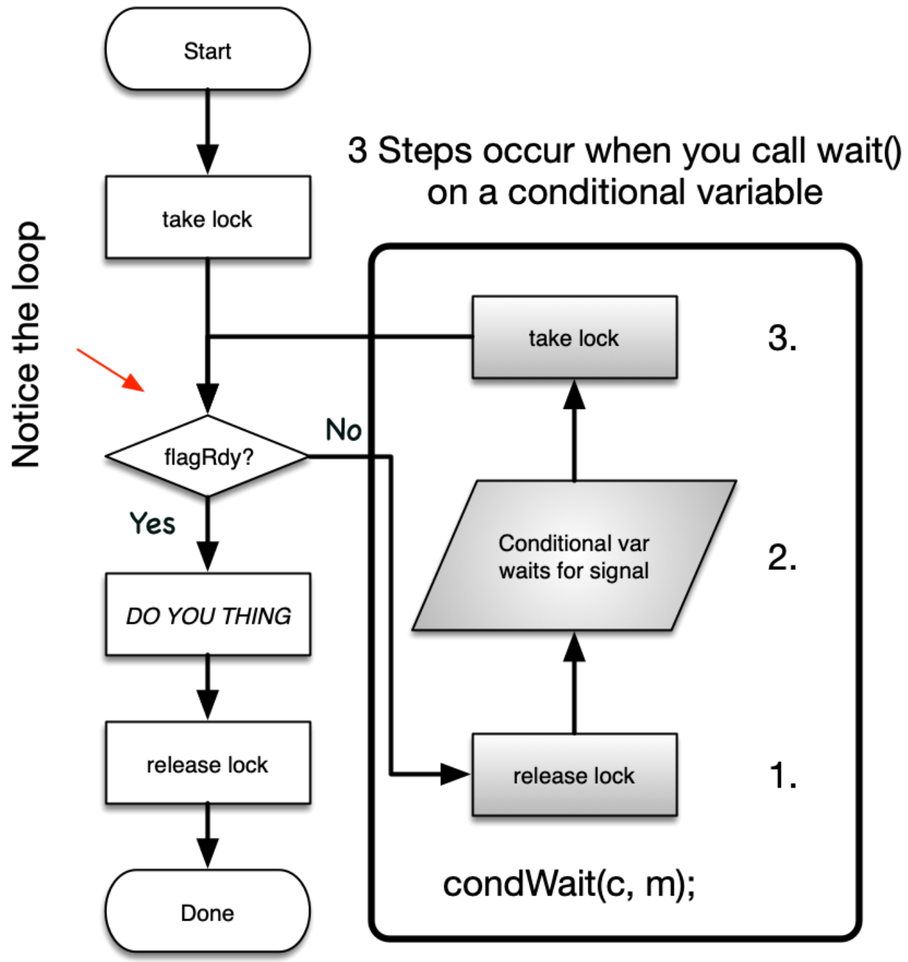
THE WAITER

WHY THE LOOP?

- To ensure that the condition to continue is there
  - Might have changed
  - Might have been woken by mistake

INSIDE WAIT

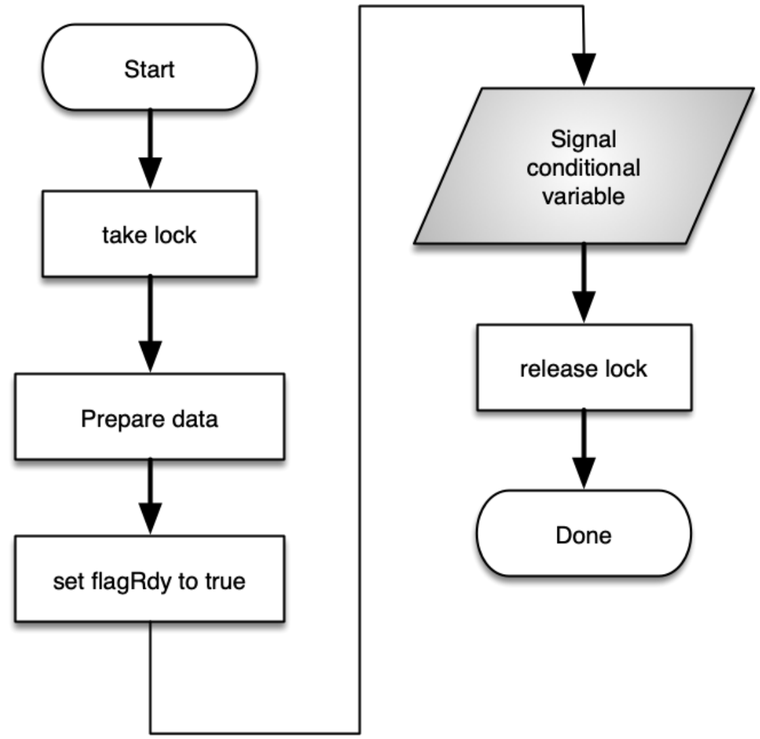
1. Release lock, we dont need lock
   - Enables the signaler to take lock and *do stuff*
2. Do the actual *waiting*
3. Take lock back, need it back before we return



THE SIGNALER

SIGNALLING

- Take lock, critical section
- Do what we need to do
- Set flag
- Signal conditional to wake *waiter*
- release lock



PARK-A-LOT 2000 - FEEBLE ATTEMPT

Our first attempt

"We hope it works" ...'hope' is a word system engineers don't like!

```
01  carDriverThread()
02  {
03    driveUpToGarageDoor()
04    sleep(GARAGE_DOOR_OPEN_TIME);
05    // Let's hope the door is open!
06    driveIntoGarage();
07  }
```

```
01  garageDoorControllerThread()
02  {
03    openGarageDoor()
04    sleep(CAR_ENTER_GARAGE_TIME);
05    // Let's hope the car is in!
06    closeGarageDoor();
07  }
```

- 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)

OUR SECOND ATTEMPT: TWO-WAY SYNCHRONIZATION

```
01  carDriverThread()
02  {
03    driveUpToGarageDoor();
04    lock(mut);
05    carWaiting = true;
06    condSignal(entry);
07
08    while(!garageDoorOpen)
09        condWait(entry, mut);
10
11    driveIntoGarage();
12    carWaiting = false;
13    condSignal(entry);
14    unlock(mut);
15  }
```

```
01  garageDoorControllerThread()
02  {
03    lock(mut);
04    while(!carWaiting)
05      condWait(entry, mut);
06
07    openGarageDoor();
08    garageDoorOpen = true;
09    condSignal(entry);
10    while(carWaiting)
11        condWait(entry, mut);
12
13    closeGarageDoor();
14    garageDoorOpen = false;
15    unlock(mut);
16  }
```

OUR SECOND ATTEMPT: TWO-WAY SYNCHRONIZATION

```
01  carDriverThread()
02  {
03    driveUpToGarageDoor();
04    lock(mut);
05    carWaiting = true;
06    condSignal(entry);
07
08    while(!garageDoorOpen)
09         condWait(entry, mut);
10
11    driveIntoGarage();
12    carWaiting = false;
13    condSignal(entry);
14    unlock(mut);
15  }
```

```
01  garageDoorControllerThread()
02  {
03    lock(mut);
04    while(!carWaiting)
05      condWait(entry, mut);
06
07    openGarageDoor();
08    garageDoorOpen = true;
09    condSignal(entry);
10    while(carWaiting)
11         condWait(entry, mut);
12
13    closeGarageDoor();
14    garageDoorOpen = false;
15    unlock(mut);
16  }
```

GUARD

- *Take lock and wait until a car wants to enter*

OUR SECOND ATTEMPT: TWO-WAY SYNCHRONIZATION

```
01  carDriverThread()
02  {
03    driveUpToGarageDoor();
04    lock(mut);
05    carWaiting = true;
06    condSignal(entry);
07
08    while(!garageDoorOpen)
09        condWait(entry, mut);
10
11    driveIntoGarage();
12    carWaiting = false;
13    condSignal(entry);
14    unlock(mut);
15  }
```

```
01  garageDoorControllerThread()
02  {
03    lock(mut);
04    while(!carWaiting)
05      condWait(entry, mut);
06
07    openGarageDoor();
08    garageDoorOpen = true;
09    condSignal(entry);
10    while(carWaiting)
11        condWait(entry, mut);
12
13    closeGarageDoor();
14    garageDoorOpen = false;
15    unlock(mut);
16  }
```

CAR

- *Drive to garage door*

OUR SECOND ATTEMPT: TWO-WAY SYNCHRONIZATION

```
01  carDriverThread()
02  {
03    driveUpToGarageDoor();
04    lock(mut);
05    carWaiting = true;
06    condSignal(entry);
07
08    while(!garageDoorOpen)
09        condWait(entry, mut);
10
11    driveIntoGarage();
12    carWaiting = false;
13    condSignal(entry);
14    unlock(mut);
15  }
```

```
01  garageDoorControllerThread()
02  {
03    lock(mut);
04    while(!carWaiting)
05      condWait(entry, mut);
06
07    openGarageDoor();
08    garageDoorOpen = true;
09    condSignal(entry);
10    while(carWaiting)
11        condWait(entry, mut);
12
13    closeGarageDoor();
14    garageDoorOpen = false;
15    unlock(mut);
16  }
```

CAR

- *Enter critical section*
- *Indicate that car is waiting*

OUR SECOND ATTEMPT: TWO-WAY SYNCHRONIZATION

```
01  carDriverThread()
02  {
03    driveUpToGarageDoor();
04    lock(mut);
05    carWaiting = true;
06    condSignal(entry);
07
08    while(!garageDoorOpen)
09        condWait(entry, mut);
10
11    driveIntoGarage();
12    carWaiting = false;
13    condSignal(entry);
14    unlock(mut);
15  }
```

```
01  garageDoorControllerThread()
02  {
03    lock(mut);
04    while(!carWaiting)
05      condWait(entry, mut);
06
07    openGarageDoor();
08    garageDoorOpen = true;
09    condSignal(entry);
10    while(carWaiting)
11        condWait(entry, mut);
12
13    closeGarageDoor();
14    garageDoorOpen = false;
15    unlock(mut);
16  }
```

CAR

- *Inform guard that a car has arrived*

OUR SECOND ATTEMPT: TWO-WAY SYNCHRONIZATION

```
01  carDriverThread()
02  {
03    driveUpToGarageDoor();
04    lock(mut);
05    carWaiting = true;
06    condSignal(entry);
07
08    while(!garageDoorOpen)
09        condWait(entry, mut);
10
11    driveIntoGarage();
12    carWaiting = false;
13    condSignal(entry);
14    unlock(mut);
15  }
```

```
01  garageDoorControllerThread()
02  {
03    lock(mut);
04    while(!carWaiting)
05      condWait(entry, mut);
06
07    openGarageDoor();
08    garageDoorOpen = true;
09    condSignal(entry);
10    while(carWaiting)
11        condWait(entry, mut);
12
13    closeGarageDoor();
14    garageDoorOpen = false;
15    unlock(mut);
16  }
```

CAR

- *Wait until garage door has been opened!*

OUR SECOND ATTEMPT: TWO-WAY SYNCHRONIZATION

```
01  carDriverThread()
02  {
03    driveUpToGarageDoor();
04    lock(mut);
05    carWaiting = true;
06    condSignal(entry);
07
08    while(!garageDoorOpen)
09        condWait(entry, mut);
10
11    driveIntoGarage();
12    carWaiting = false;
13    condSignal(entry);
14    unlock(mut);
15  }
```

```
01  garageDoorControllerThread()
02  {
03    lock(mut);
04    while(!carWaiting)
05      condWait(entry, mut);
06
07    openGarageDoor();
08    garageDoorOpen = true;
09    condSignal(entry);
10    while(carWaiting)
11        condWait(entry, mut);
12
13    closeGarageDoor();
14    garageDoorOpen = false;
15    unlock(mut);
16  }
```

GUARD

- *Wake up and check to see if supposed to wakeup*

OUR SECOND ATTEMPT: TWO-WAY SYNCHRONIZATION

```
01  carDriverThread()
02  {
03    driveUpToGarageDoor();
04    lock(mut);
05    carWaiting = true;
06    condSignal(entry);
07
08    while(!garageDoorOpen)
09        condWait(entry, mut);
10
11    driveIntoGarage();
12    carWaiting = false;
13    condSignal(entry);
14    unlock(mut);
15  }
```

```
01  garageDoorControllerThread()
02  {
03    lock(mut);
04    while(!carWaiting)
05      condWait(entry, mut);
06
07    openGarageDoor();
08    garageDoorOpen = true;
09    condSignal(entry);
10    while(carWaiting)
11        condWait(entry, mut);
12
13    closeGarageDoor();
14    garageDoorOpen = false;
15    unlock(mut);
16  }
```

GUARD

- *Open door*
- *Indicate that door is opened*
- *Signal car*

OUR SECOND ATTEMPT: TWO-WAY SYNCHRONIZATION

```
01  carDriverThread()
02  {
03    driveUpToGarageDoor();
04    lock(mut);
05    carWaiting = true;
06    condSignal(entry);
07
08    while(!garageDoorOpen)
09        condWait(entry, mut);
10
11    driveIntoGarage();
12    carWaiting = false;
13    condSignal(entry);
14    unlock(mut);
15  }
```

```
01  garageDoorControllerThread()
02  {
03    lock(mut);
04    while(!carWaiting)
05      condWait(entry, mut);
06
07    openGarageDoor();
08    garageDoorOpen = true;
09    condSignal(entry);
10    while(carWaiting)
11        condWait(entry, mut);
12
13    closeGarageDoor();
14    garageDoorOpen = false;
15    unlock(mut);
16  }
```

GUARD

- *Wait until car has entered*

OUR SECOND ATTEMPT: TWO-WAY SYNCHRONIZATION

```
01  carDriverThread()
02  {
03    driveUpToGarageDoor();
04    lock(mut);
05    carWaiting = true;
06    condSignal(entry);
07
08    while(!garageDoorOpen)
09        condWait(entry, mut);
10
11    driveIntoGarage();
12    carWaiting = false;
13    condSignal(entry);
14    unlock(mut);
15  }
```

```
01  garageDoorControllerThread()
02  {
03    lock(mut);
04    while(!carWaiting)
05      condWait(entry, mut);
06
07    openGarageDoor();
08    garageDoorOpen = true;
09    condSignal(entry);
10    while(carWaiting)
11        condWait(entry, mut);
12
13    closeGarageDoor();
14    garageDoorOpen = false;
15    unlock(mut);
16  }
```

CAR

- *Wake up and check to see if supposed to wakeup*

OUR SECOND ATTEMPT: TWO-WAY SYNCHRONIZATION

```
01  carDriverThread()
02  {
03    driveUpToGarageDoor();
04    lock(mut);
05    carWaiting = true;
06    condSignal(entry);
07
08    while(!garageDoorOpen)
09        condWait(entry, mut);
10
11    driveIntoGarage();
12    carWaiting = false;
13    condSignal(entry);
14    unlock(mut);
15  }
```

```
01  garageDoorControllerThread()
02  {
03    lock(mut);
04    while(!carWaiting)
05      condWait(entry, mut);
06
07    openGarageDoor();
08    garageDoorOpen = true;
09    condSignal(entry);
10    while(carWaiting)
11        condWait(entry, mut);
12
13    closeGarageDoor();
14    garageDoorOpen = false;
15    unlock(mut);
16  }
```

CAR

- *Drive into garage*
- *Indicate that car has entered*
- *Signal guard*

OUR SECOND ATTEMPT: TWO-WAY SYNCHRONIZATION

```
01  carDriverThread()
02  {
03    driveUpToGarageDoor();
04    lock(mut);
05    carWaiting = true;
06    condSignal(entry);
07
08    while(!garageDoorOpen)
09        condWait(entry, mut);
10
11    driveIntoGarage();
12    carWaiting = false;
13    condSignal(entry);
14    unlock(mut);
15  }
```

```
01  garageDoorControllerThread()
02  {
03    lock(mut);
04    while(!carWaiting)
05      condWait(entry, mut);
06
07    openGarageDoor();
08    garageDoorOpen = true;
09    condSignal(entry);
10    while(carWaiting)
11        condWait(entry, mut);
12
13    closeGarageDoor();
14    garageDoorOpen = false;
15    unlock(mut);
16  }
```

CAR

- *Release lock*

OUR SECOND ATTEMPT: TWO-WAY SYNCHRONIZATION

```
01  carDriverThread()
02  {
03    driveUpToGarageDoor();
04    lock(mut);
05    carWaiting = true;
06    condSignal(entry);
07
08    while(!garageDoorOpen)
09        condWait(entry, mut);
10
11    driveIntoGarage();
12    carWaiting = false;
13    condSignal(entry);
14    unlock(mut);
15  }
```

```
01  garageDoorControllerThread()
02  {
03    lock(mut);
04    while(!carWaiting)
05      condWait(entry, mut);
06
07    openGarageDoor();
08    garageDoorOpen = true;
09    condSignal(entry);
10    while(carWaiting)
11        condWait(entry, mut);
12
13    closeGarageDoor();
14    garageDoorOpen = false;
15    unlock(mut);
16  }
```

GUARD

- *Wake up and check to see if supposed to wakeup*

OUR SECOND ATTEMPT: TWO-WAY SYNCHRONIZATION

```
01  carDriverThread()
02  {
03    driveUpToGarageDoor();
04    lock(mut);
05    carWaiting = true;
06    condSignal(entry);
07
08    while(!garageDoorOpen)
09        condWait(entry, mut);
10
11    driveIntoGarage();
12    carWaiting = false;
13    condSignal(entry);
14    unlock(mut);
15  }
```

```
01  garageDoorControllerThread()
02  {
03    lock(mut);
04    while(!carWaiting)
05      condWait(entry, mut);
06
07    openGarageDoor();
08    garageDoorOpen = true;
09    condSignal(entry);
10    while(carWaiting)
11        condWait(entry, mut);
12
13    closeGarageDoor();
14    garageDoorOpen = false;
15    unlock(mut);
16  }
```

GUARD

- *Close garage door*
- *Indicate that door i closed*
- *Release lock*

OUR SECOND ATTEMPT: TWO-WAY SYNCHRONIZATION

```
01  carDriverThread()
02  {
03    driveUpToGarageDoor();
04    lock(mut);
05    carWaiting = true;
06    condSignal(entry);
07
08    while(!garageDoorOpen)
09        condWait(entry, mut);
10
11    driveIntoGarage();
12    carWaiting = false;
13    condSignal(entry);
14    unlock(mut);
15  }
```

```
01  garageDoorControllerThread()
02  {
03    lock(mut);
04    while(!carWaiting)
05      condWait(entry, mut);
06
07    openGarageDoor();
08    garageDoorOpen = true;
09    condSignal(entry);
10    while(carWaiting)
11        condWait(entry, mut);
12
13    closeGarageDoor();
14    garageDoorOpen = false;
15    unlock(mut);
16  }
```

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

TYPES OF SYNCHRONIZATION METHODS

TYPES OF SYNCHRONIZATION METHODS

Sync method	Behavour
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

```c
#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)
```

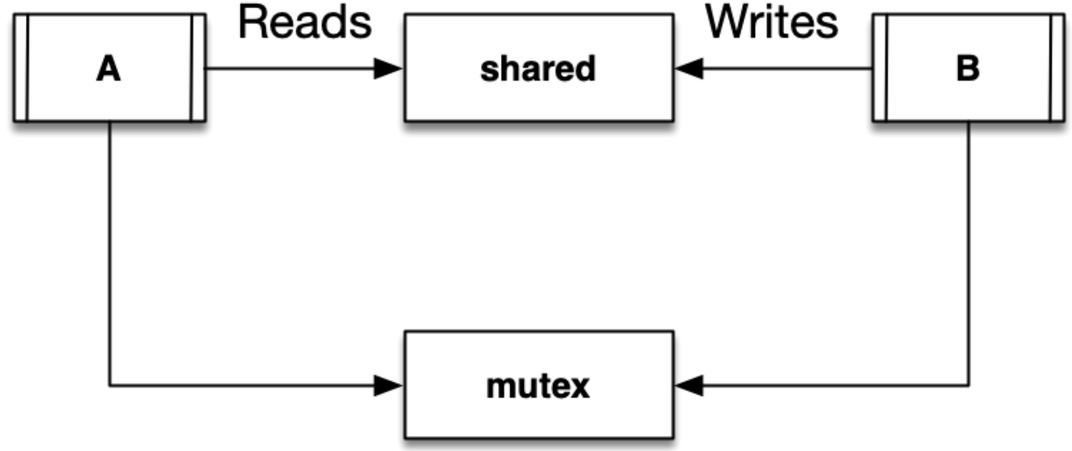
```c
#include<semaphore.h>

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

AIDS & TOOLS

THE MONITOR

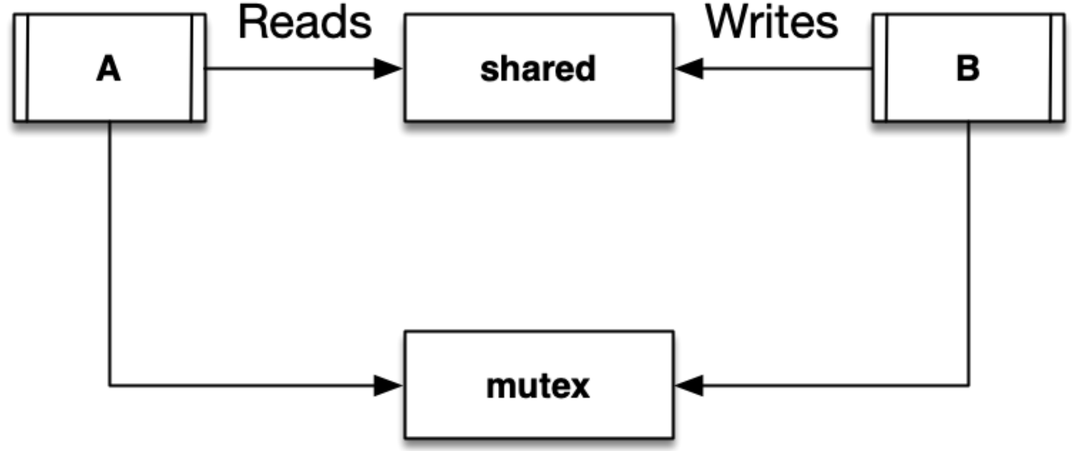
- Monitor: A template class
  - When accessed, the Monitor 1. takes mutex, 2. accesses shared, 3. releases mutex
  - Responsibility for mutual exclusion: Programmer -> monitor



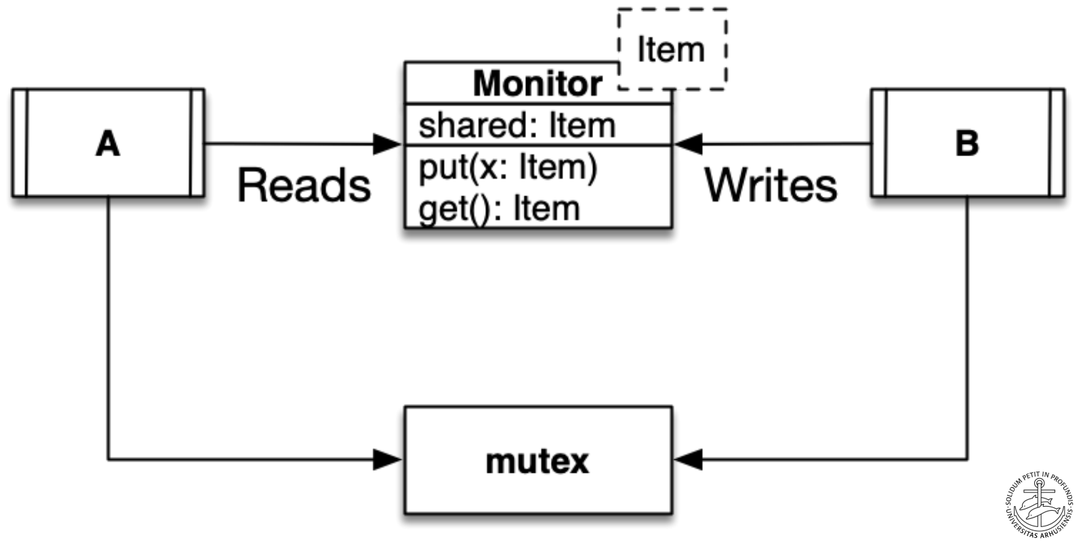
AIDS & TOOLS

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()`?



AIDS & TOOLS

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!!!*