
DALL·E

Cooperation via Shared Memory

- Multiple agents are allowed to access the same memory cells or variables, and communication is handled through them.
- Used in most prominent (imperative) programming languages (Java, C, C++, Rust, ...) exclusively or jointly with message passing.

Shared memory is a very simple and natural extension to imperative programming languages.

At least on first sight...

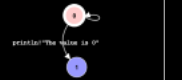
Why are DATA RACES a Problem?

And in pseuCo?

- We are **banning** data races in pseuCo:
  A pseuCo program that has a data race is **always** considered incorrect, for any specification.
- The CCS semantics of pseuCo is not aware of this, and neither is the pseuCo Debugger: They only show the possible interleavings of reads and writes, including any race conditions this may cause, but they do not flag that the program is *fundamentally wrong*.
  The reason for this is simply that it's complicated. Notably, deciding absence of data race is undecidable in general.
- PseuCo Book is aware, and its programming exercises will reject programs with data races.

Concurrent Countdown (safe)

```
int n = 6;
lock guard_n;

void counter() {
    for (int i = 0; i < 3; i++) {
        lock(guard_n);
        n = n - 1;
        unlock(guard_n);
    }
}

mainAgent {
    agent a1 = start(counter());
    agent a2 = start(counter());
    join(a1); join(a2);
    println("The value is " + n);
}
```



Basic Rules of Locks

- Each shared variable must be *guarded* by exactly one lock.
  Using one lock for multiple variables is OK, but using multiple locks for one variable is a mistake.
- For shared variables, any access, **whether reading or writing**, is only allowed while the agent attempting access **holds the corresponding lock**.
- Recommendation: Write down which lock guards a variable:
  `int n; // guarded by guard_n`

The Art of Locking

```
for (int i = 0; i < 3; i++) {
    lock(guard_n); tmp = n; unlock(guard_n);
    tmp--;
    lock(guard_n); n = tmp; unlock(guard_n);
}
```

small

```
for (int i = 0; i < 3; i++) {
    lock(guard_n);
    n--;
    unlock(guard_n);
}
```

large

full interleaving race conditions (but no data races)

sweet spot?

sequential execution

Locking Critical Sections

When a program enters a critical section, it must first acquire all relevant locks for the shared variables it wants to touch.
It can only release them when it leaves the critical section.

Hopefully, this means only a single lock.

If multiple variables are often used together in the same critical section, consider guarding both with the same lock.

Deadlocks, Revisited

Terminal states are called **deadlocks**, but we usually reserve the term for unintended termination. Just as is the case here!

Deadlocks manifest in many ways, but the most common one is a **classical deadlock**, characterized by four properties:

- **Mutual Exclusion:** Only one agent can use a resource at any given time.
- **No Preemption:** Resources are released only voluntarily by the respective agent holding them. No lockpicking!
- **Hold and Wait:** An agent holding a resource is requesting an additional resource.
- **Circular Wait:** There is a circularity in needed resources – agents are waiting for resources which are being held by other agents.

easiest to avoid

One More Thing

```
lock key;

void angel() {
    lock(key);
    println("I'm in heaven!");
    unlock(key);
}

void devil() {
    while(true) {
        lock(key);
        // (devilish laugh)
        unlock(key);
    }
}

mainAgent {
    start(angel());
    start(devil());
}
```

Is our angel guaranteed to reach heaven?

They can ride the τ-carousel forever.

Life is not fair!
Neither are locks.
Write your programs in a way that ensures they make progress regardless of which agent obtains the next lock!
But you can assume each agent will get to move eventually.

Introducing Monitors

This data structure is a **monitor**.

Monitors are abstract data structures that **themselves** protect their (per-instance) internal data by

- making it accessible exclusively through their methods,
- ensuring mutual exclusion for their methods,
- and do so using a single lock.

There's more to monitors, which we'll see soon!

```
struct Counter {
    int n; // not directly accessible
    lock g;

    void set(int x) {
        lock(g);
        n = x;
        unlock(g);
    }

    int get() {
        lock(g);
        int res = n;
        unlock(g);
        return res;
    }

    void decrement() {
        lock(g);
        n--;
        unlock(g);
    }
}
```

The Promise of Monitors

A well-defined monitor guarantees that there are no data races on any of its fields and that all of its methods behave atomically.

This allows the user of a monitor not to worry about its internals, *even when calling its methods concurrently*.

Recommendation: Make all shared data structures monitors!

Introducing Monitors

```
Counter c;

void counter() {
    for (int i = 0; i < 3; i++) {
        c.set(c.get() - 1);
    }
}

mainAgent {
    c.set(6);
    agent a1 = start(counter());
    agent a2 = start(counter());
    join(a1);
    join(a2);
    println("The value is " + c.get());
}
```

No data race, but race condition!

```
/* a monitor */
struct Counter {
    int n; // not directly accessible
    lock g;

    void set(int x) {
        lock(g);
        n = x;
        unlock(g);
    }

    int get() {
        lock(g);
        int res = n;
        unlock(g);
        return res;
    }
}
```

Introducing Monitors

```
Counter c; lock g;

void counter() {
    for (int i = 0; i < 3; i++) {
        lock(g);
        c.set(c.get() - 1);
        unlock(g);
    }
}

mainAgent {
    c.set(6);
    agent a1 = start(counter());
    agent a2 = start(counter());
    join(a1);
    join(a2);
    println("The value is " + c.get());
}
```

client-side locking

```
/* a monitor */
struct Counter {
    int n; // not directly accessible
    lock g;

    void set(int x) {
        lock(g);
        n = x;
        unlock(g);
    }

    int get() {
        lock(g);
        int res = n;
        unlock(g);
        return res;
    }
}
```

Works here, but extremely dangerous. What if someone else accesses the counter without using the lock g? Negates the benefits of the monitor!

Later on, we'll see even more things that can go wrong in similar situations.

Introducing Monitors

```
Counter c;

void counter() {
    for (int i = 0; i < 3; i++) {
        Counter newCounter;
        newCounter.set(c.get() - 1);
        c = newCounter;
    }
}

mainAgent {
    c.set(6);
    agent a1 = start(counter());
    agent a2 = start(counter());
    join(a1);
    join(a2);
    println("The value is " + c.get());
}
```

```
/* a monitor */
struct Counter {
    int n; // not directly accessible
    lock g;

    void set(int x) {
        lock(g);
        n = x;
        unlock(g);
    }

    int get() {
        lock(g);
        int res = n;
        unlock(g);
        return res;
    }
}
```

Monitors Are Not Perfect

Monitors are great! But even if you store all data in monitors, you have to be careful:

- Monitors only protect their internal data, nothing else.
  Notably, the variables that store (references to) monitors are not protected!
- In-between method calls to the monitor, other agents can change the monitor!
  If you have critical sections spanning multiple monitor calls, consider extending the monitor to offer the critical operation as a single method.
  As seen previously in the version of the counter that had a `decrement()` method.

Enter Through the Back Door

```
struct Counter {
    int n; // not directly accessible
    lock g;

    void set(int x) {
        lock(g);
        n = x;
        unlock(g);
    }

    int get() {
        lock(g);
        int res = n;
        unlock(g);
        return res;
    }

    void decrement() {
        lock(g);
        set(get() - 1);
        unlock(g);
    }
}
```

What happens here?

Reentrant Locks

Locks can only be locked once.

This is inconvenient because it blocks calling other methods using the same lock (e.g. recursion).

And it's unnecessary – memory accesses from the same agent are always safe.

To alleviate this, `lock` has a special feature: Any agent holding it is allowed to lock it multiple times.

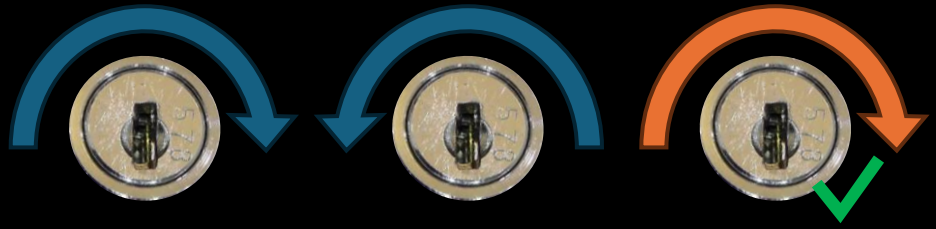
How does that work?


```
struct Counter {
    int n; // not directly accessible
    lock g;

    void set(int x) {
        lock(g);
        n = x;
        unlock(g);
    }

    int get() {
        lock(g);
        int res = n;
        unlock(g);
        return res;
    }

    void decrement() {
        lock(g);
        set(get() - 1);
        unlock(g);
    }
}
```

Locks in CCS

```
Proc_Counter_$set[a, i, $x] := env_class_Counter_get_g(i)?$0
                             . lock($0)!a
                             . env_class_Counter_set_n(i)!$x
                             . env_class_Counter_get_g(i)?$1
                             . unlock($1)!a.1


Mutex[i, c, a] :=
  when (c == 0)  lock(i)? a .Mutex[i, 1, a]
+ when (c > 0) ( lock(i)?(a).Mutex[i, c + 1, a]
               + multilock(i)?r.Mutex[i, c + r, a]
               + fullunlock(i)!c.Mutex[i, 0, a]
               + unlock(i)?a2.( when (a == a2) Mutex[i, c - 1, a]
                              + when (a != a2) exception!.0))

Mutex_cons[next_i] := mutex_create!next_i .
                      (Mutex_cons[next_i + 1] | Mutex[next_i, 0, 0])
```

Reentrant Locks in pseuCo



Lock was locked once, then unlocked.
Other agent may lock it.

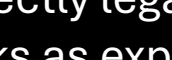
Lock was locked by blue agent twice but only unlocked once.
Orange agent can’t lock it, has to wait.

Orange agent tries to unlock a lock they did not lock.
This is illegal and indicates a programming error.

Reentrant Locks

```
struct Counter {
    int n; // not directly accessible
    lock g;

    void set(int x) {
        lock(g);
        n = x;
        unlock(g);
    }

    int get() {
        lock(g);
        int res = n;
        unlock(g);
        return res;
    }

    void decrement() {
        lock(g);
        set(get() - 1);
        unlock(g);
    }
}
```

Perfectly legal,
works as expected!



Writing all these `lock()` and `unlock()` calls seems really cumbersome.
If only there was a better way…

Introducing the `monitor` Feature

```
struct Counter {
    int n; // not directly accessible
    lock g;

    void set(int x) {
        lock(g);
        n = x;
        unlock(g);
    }

    int get() {
        lock(g);
        int res = n;
        unlock(g);
        return res;
    }

    void decrement() {
        lock(g);
        n--;
        unlock(g);
    }
}
```

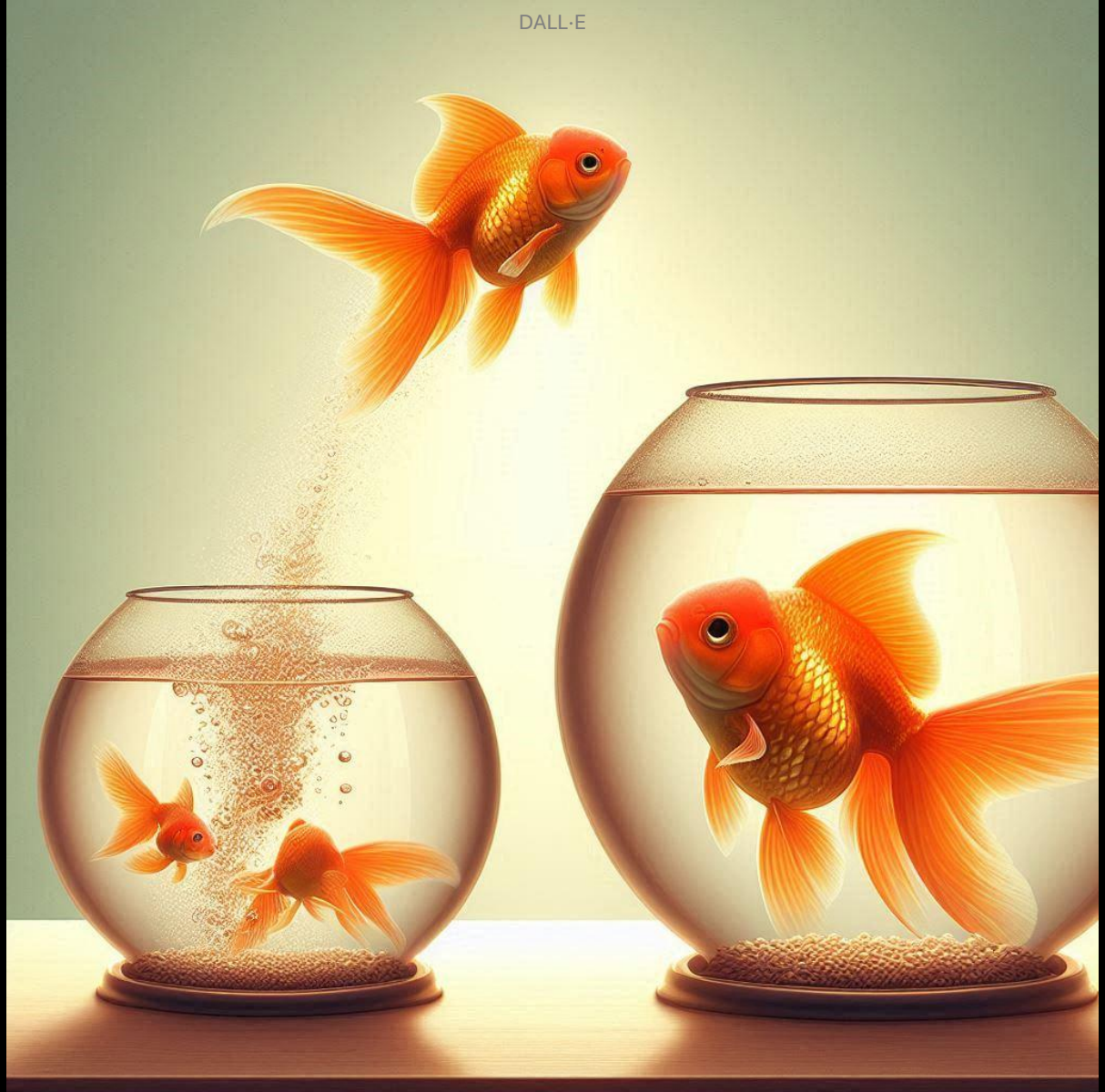
```
monitor Counter {
    int n; // not directly accessible


    void set(int x) {

        n = x;

    }

    int get() {



        return n;
    }

    void decrement() {

        n--;

    }
}
```

Introducing the `monitor` Feature

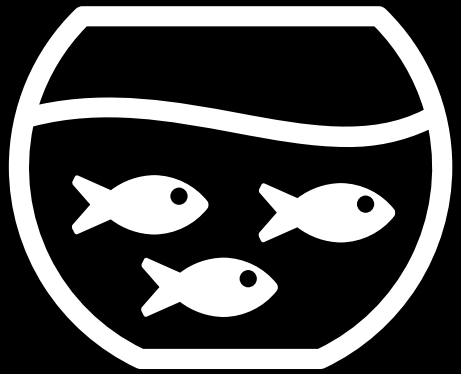
- syntax like a `struct`, just replace the keyword
- implicit (and not directly accessible) lock
- automatically locked and unlocked
- automatic handling of `return` statements

To be continued!

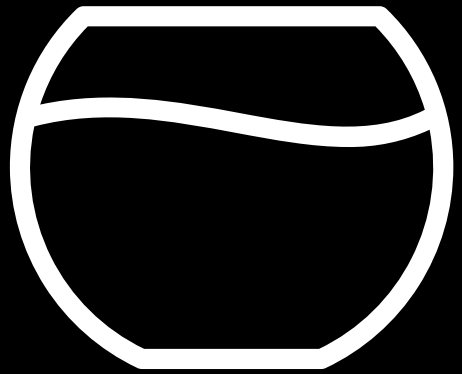
```
monitor Counter {
    int n; // not directly accessible

    void set(int x) {
        n = x;
    }

    int get() {
        return n;
    }

    void decrement() {
        n--;
    }
}
```



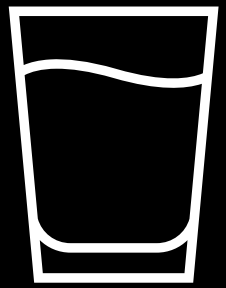
Monitors: A Fishy Analogy



ocean



waiting
for lock



monitor

Laziness is a Virtue

How to Wait Properly

Channels in pseuCo-SM

Normal channels are banned in pseuCo-SM. Why not build our own? Here's an `intchan3`:

```
monitor Channel3 {
    int[3] n;
    int used = 0;

    void put(int x) {
        if (used < 3) {
            n[used] = x;
            used++;
        }
    }

    int get() {
        if (used > 0) {
            int res = n[0];
            for (int i = 1; i < used; i++) n[i-1] = n[i];
            used--;
            return res;
        }
    }
}
```

Does that work?

Generally yes, with two problems:

- Channel full? New values are dropped!
  Expected: Sender has to wait
- Channel empty? Receiving does not return a value!
  Expected: Receiver has to wait

How to wait?

Channels in pseuCo-SM

```
struct Channel3 {
    lock g;
    int[3] n;
    int used = 0;

    void put(int x) {
        lock(g);
        while (used >= 3) { unlock(g); lock(g); }
        n[used] = x;
        used++;
        unlock(g);
    }

    int get() {
        lock(g);
        while (used <= 0) { unlock(g); lock(g); }
        int res = n[0];
        for (int i = 1; i < used; i++) n[i-1] = n[i];
        used--;
        unlock(g);
        return res;
    }
}
```

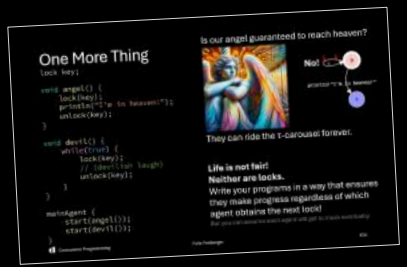
Super duper ugly. But does it work?

Not really. Other agents have a brief chance to act, but with bad luck (or an evil scheduler), this can cycle forever.

And if no-one else is here, it'll heat the planet.

We call this a **busy wait**.

Busy waits are fundamental programming errors.



Patience: A Manual

When a method of a monitor has to await a certain condition,
i.e. data being available, or free space being available,
it must proceed as follows:

- check whether the condition is satisfied, if not:
- release the lock,
- wait for someone else to make a change to the monitor,
- re-acquire the lock, and
- return to step 1.

We cannot implement this using the pseuCo features we have seen so far. Also, details matter a lot – it's easy to build a deadlock.

Patience: A Manual

These features come with the `monitor` feature in pseuCo:

Declares a condition that may be awaited later on.

Waits for the condition to be satisfied.

Notifies others that this condition *may* be satisfied.

```
monitor Channel3 {
    int[3] n;
    int used = 0;

    condition roomAvailable with (used < 3);
    condition dataAvailable with (used > 0);

    void put(int x) {
        waitForCondition(roomAvailable);
        n[used] = x;
        used++;
        signal(dataAvailable);
    }

    int get() {
        waitForCondition(dataAvailable);
        int res = n[0];
        for (int i = 1; i < used; i++) n[i-1] = n[i];
        used--;
        signal(roomAvailable);
        return res;
    }
}
```

Condition Synchronization: The Fine Print

- **`signal(c)`**
  Notify *one* agent waiting for condition `c` that they should check the condition again.
  Nonblocking, destructive multicast: If no one is listening, the signal is lost.
- **`waitForCondition(c)`**
  Does the Boolean condition for `c` already hold? If so, go on. If not, 1 leave the monitor and 2 wait to be notified. Once notified, start over.

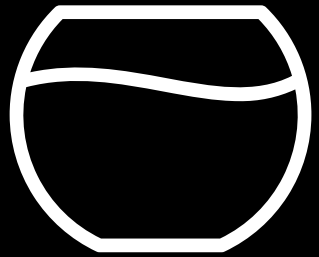
Is the order of 1 and 2 OK?

No! Signals may be missed.

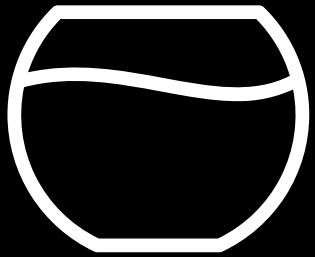
Scenario: Channel is empty, agent 1 tries to receive, leaves monitor. Before they start waiting, agent 2 fills up the channel, tries to insert a fourth element, goes to sleep. Only then, agent 1 starts waiting. Now, both agents wait for a signal!

Solution: Make this atomic – start listening for signals before leaving the monitor!

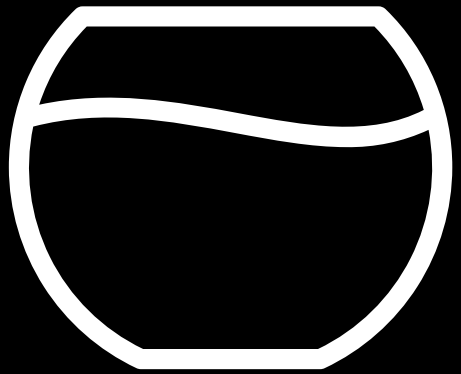
Fishy Conditions



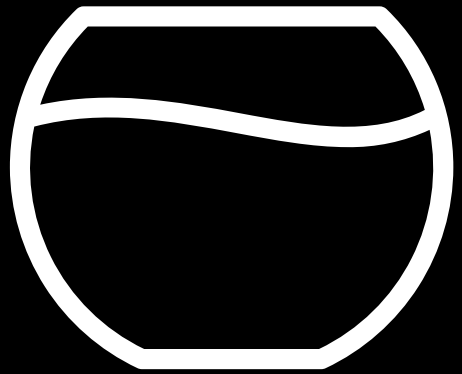
waiting for
`roomAvailable`



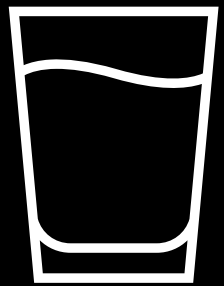
waiting for
`dataAvailable`



ocean

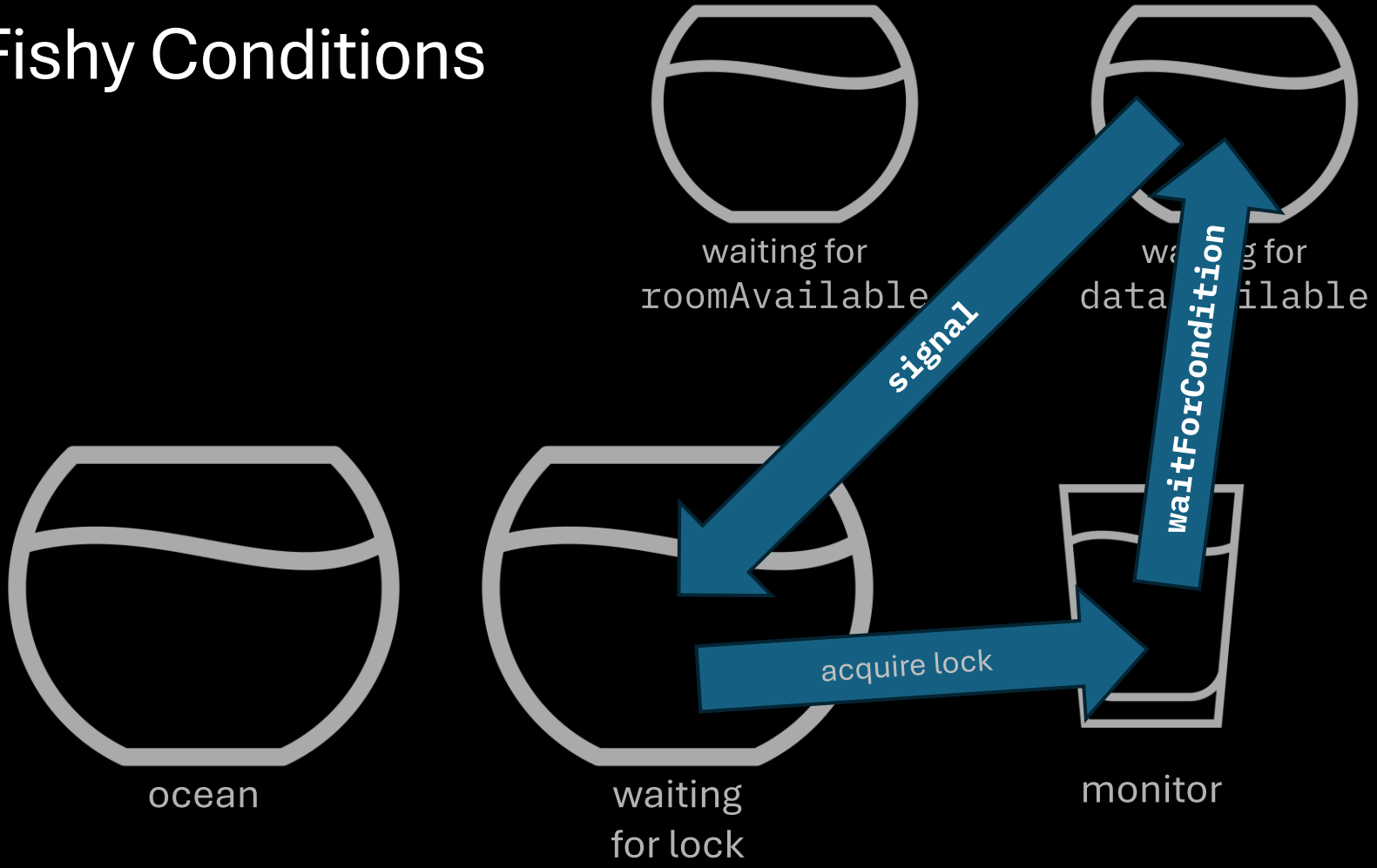


waiting
for lock

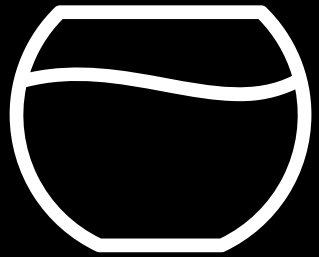


monitor

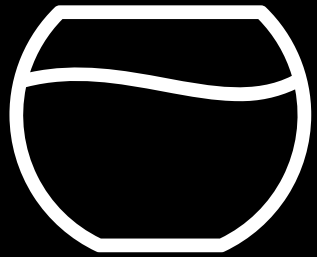
Fishy Conditions



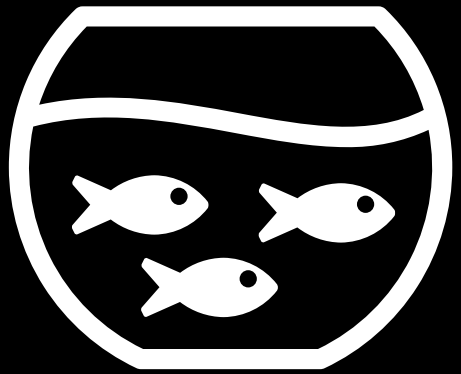
Fishy Conditions



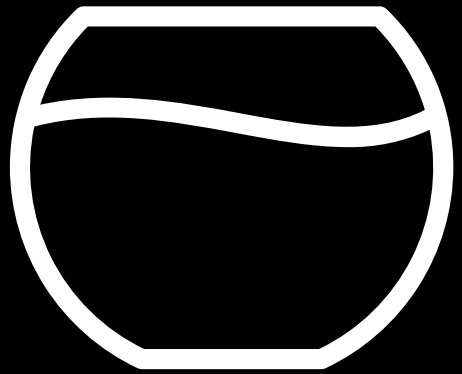
waiting for `roomAvailable`



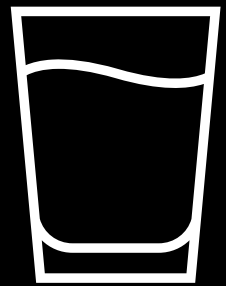
waiting for `dataAvailable`



ocean

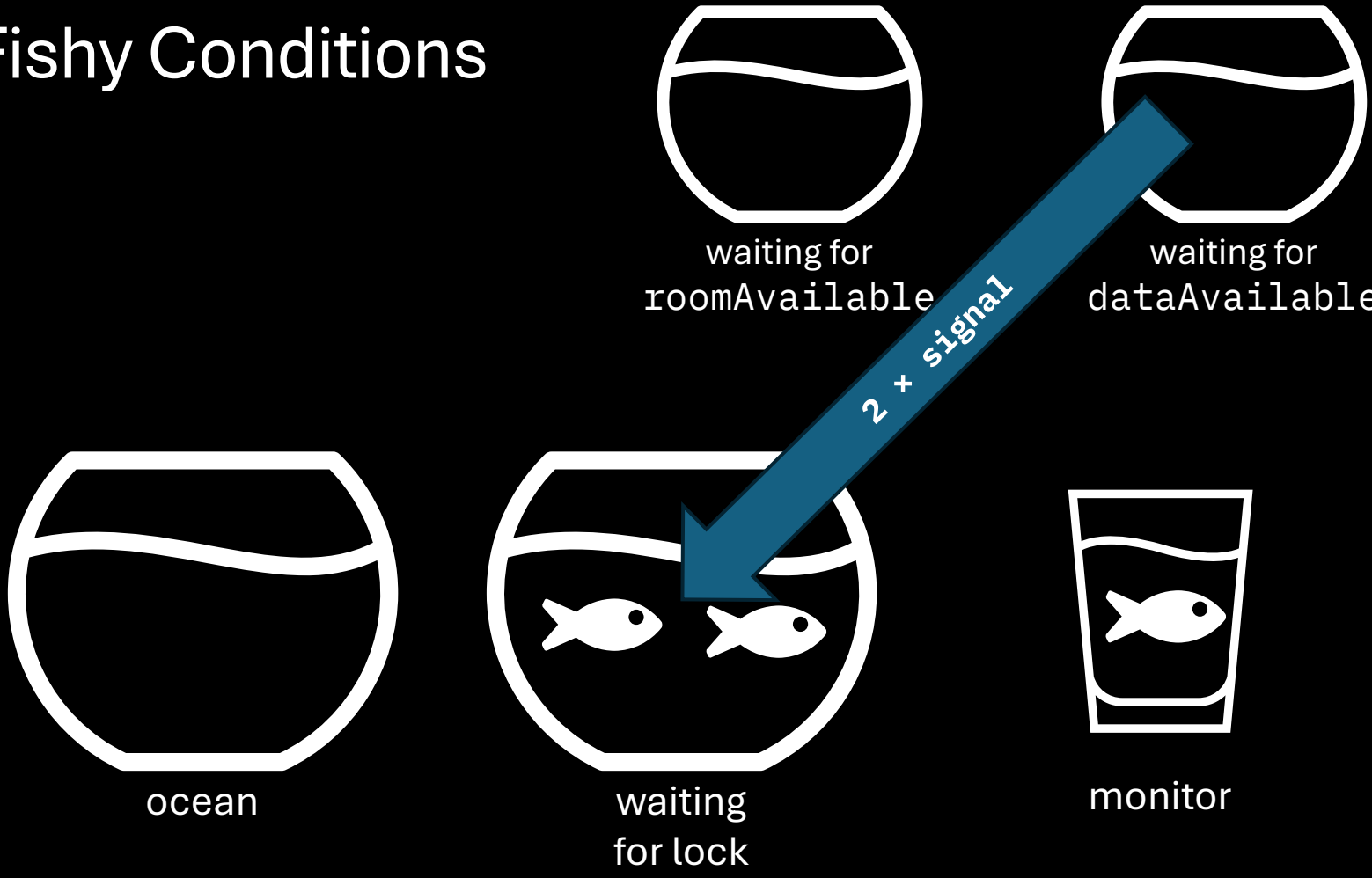


waiting for lock



monitor

Fishy Conditions



Condition Synchronization: The Fine Print

- **`signal(c)`** nondeterministic choice
  Notify *one* agent waiting for condition `c` that they should check the condition again.
- **`signalAll(c)`**
  Notify *all* agents waiting for condition `c` that they should check the condition again.

Do we need a `signalAll` here?
No.

```
monitor Channel3 {
    int[3] n;
    int used = 0;

    condition roomAvailable with (used < 3);
    condition dataAvailable with (used > 0);

    void put(int x) {
        waitForCondition(roomAvailable);
        n[used] = x;
        used++;
        signal(dataAvailable);
    }

    int get() {
        waitForCondition(dataAvailable);
        int res = n[0];
        for (int i = 1; i < used; i++) n[i-1] = n[i];
        used--;
        signal(roomAvailable);
        return res;
    }
}
```

`signal` vs. `signalAll`

`signalAll` is slower: More agents need to be woken up and will fight for the lock to re-check the condition. At least if multiple agents are waiting.

Using `signal` is risky: It can cause deadlocks if used inappropriately! When is it safe to use?


Use your brain!

→ When you know that *any one* of the agents waiting can be picked, but their actions will make the condition false again.

For example, in our `Channel3`, when agents are waiting for `dataAvailable`, if someone puts in one item, any agent would be happy to take it – and afterwards, the channel will be empty again.

There are other special cases, e.g. when agents “forward” signals.

Confused? If in doubt, use `signalAll`!

DALL·E

Shared Memory is a Minefield

```
monitor Channel3 {
    int[3] n;
    int used = 0;

    condition roomAvailable with (used < 3);
    condition dataAvailable with (used > 0);

    void put(int x) {
        waitForCondition(roomAvailable);
        n[used] = x;
        used++;
        signal(dataAvailable);
    }

    int get() {
        waitForCondition(dataAvailable);
        int res = n[0];
        for (int i = 1; i < used; i++) n[i-1] = n[i];
        used--;
        signal(roomAvailable);
        return res;
    }
}
```

Good monitor ✓

Shared Memory is a Minefield

```
monitor Channel3 {
    MyBoolean[3] n;
    int used = 0;

    condition roomAvailable with (used < 3);
    condition dataAvailable with (used > 0);

    void put(MyBoolean x) {
        waitForCondition(roomAvailable);
        n[used] = x;
        used++;
        signal(dataAvailable);
    }

    MyBoolean get() {
        waitForCondition(dataAvailable);
        MyBoolean res = n[0];
        for (int i = 1; i < used; i++) n[i-1] = n[i];
        used--;
        signal(roomAvailable);
        return res;
    }
}
```

monitor

```
    struct MyBoolean {
        bool b;

        void set(bool x) {
            b = x;
        }

        bool get() {
            return b;
        }
    }
```

Surely this is fine?

No! `MyBooleans` turn into shared objects when travelling through the channel to another agents.

DATA RACE!

Putting structures into monitors
does not make them safe!
All shared data structures should be monitors.

Semaphores

Monitors have been coined by Tony Hoare and Per Brinch-Hansen.

Semaphores are a related concept, coined by Edsger Dijkstra.

A semaphore is a data structure managing a finite number of **tickets**.

An agent may obtain a ticket, provided one is available.

If no ticket is available, attempting to obtain a ticket blocks until another agent returns a ticket.

Semaphores

```
monitor Semaphore {
    int n;

    condition ticketAvailable with (n > 0);

    void up() {
        n++;
        signalAll(ticketAvailable);
    }

    void down() {
        waitForCondition(ticketAvailable);
        n--;
    }

    void init(int v) {
        n = v;
    }
}
```

necessary?

Semaphores

```
monitor Semaphore {
    int n;

    condition ticketAvailable with (n > 0);

    void up() {
        n++;
        signal(ticketAvailable);
    }

    void down() {
        waitForCondition(ticketAvailable);
        n--;
    }

    void init(int v) {
        n = v;
    }
}
```

```
void worker(Semaphore A, int n) {
    while (true){
        A.down();
        println("Worker " + n);
        A.up();
    }
}

mainAgent {
    Semaphore A;
    A.init(2);
    start(worker(A, 1));
    start(worker(A, 2));
    start(worker(A, 3));
}
```

Semaphorous Channel3

```
monitor Channel3 {
    Semaphore full, empty;
    int[3] n;
    int used = 0;

    void put(int x) {
        empty.down();
        n[used] = x;
        used++;
        full.up();
    }

    int get() {
        full.down();
        int res = n[0];
        for (int i = 1; i < used; i++) n[i-1] = n[i];
        used--;
        empty.up();
        return res;
    }
}
```

```
void init() {
    full.init(0);
    empty.init(3);
}
```

Idea: Two semaphores manage space and data. No conditions needed!

Good idea?
Let's try in the debugger!

Deadlock!
"Nested Monitor"

This attempt is unsalvageable.

Shared Memory Survival Guide

Hints to maximize your chance of survival:

- Minimize the amount of shared data, maximize local work.
  Shared memory is most performant if accessed rarely!
- Guard all shared data with locks.
- Whenever possible, encapsulate shared data in monitors.
  Provide all critical operations as methods of the monitor.
- Use mutexes and monitors consistently. Prefer simplicity.
  Do not get creative. This notoriously goes wrong.
- Double-check that you aren't missing a `signal(All)`, and if in doubt, use `signalAll`.

Structures in CCS

```
// Structs and monitors

Env_class_Semaphore[i, guard, $n, $ticketAvailable] :=
   env_class_Semaphore_guard(i)!guard
    . Env_class_Semaphore[i, guard, $n, $ticketAvailable]
 + env_class_Semaphore_get_n(i)!$n
    . Env_class_Semaphore[i, guard, $n, $ticketAvailable]
 + env_class_Semaphore_set_n(i)?$n
    . Env_class_Semaphore[i, guard, $n, $ticketAvailable]
 + env_class_Semaphore_get_ticketAvailable(i)!$ticketAvailable
    . Env_class_Semaphore[i, guard, $n, $ticketAvailable]
 + env_class_Semaphore_set_ticketAvailable(i)?$ticketAvailable
    . Env_class_Semaphore[i, guard, $n, $ticketAvailable]

Env_class_Semaphore_cons[next_i] := class_Semaphore_create!next_i
 . mutex_create?$0 . wait_create?$1
 . class_Semaphore_initialized(next_i)!
 . (Env_class_Semaphore[next_i, $0, 0, $1] | Env_class_Semaphore_cons[next_i + 1])
```

Just a bunch of cells!

Condition Synchronization in CCS

```
// Procedures

Proc_Semaphore_$down[a, i] := env_class_Semaphore_guard(i)?$0
                            . lock($0)!a
                            . Proc_Semaphore_$down_1[a, i]

Proc_Semaphore_$down_1[a, i] := env_class_Semaphore_get_n(i)?$0
 . when (!($0 > 0)) env_class_Semaphore_get_ticketAvailable(i)?$1
     . add($1)!
     . env_class_Semaphore_guard(i)?$2
     . fullunlock($2)?$3
     . wait($1)!
     . lock($2)!a . multilock($2)!($3 - 1) . Proc_Semaphore_$down_1[a, i]
 + when ($0 > 0) env_class_Semaphore_get_n(i)?$1
    . env_class_Semaphore_set_n(i)!($1 - 1)
    . env_class_Semaphore_guard(i)?$2
    . unlock($2)!a . 1
```

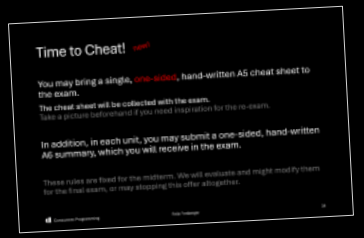
Condition Synchronization in CCS

```
// System processes

Mutex[i, c, a] :=
  when (c == 0) lock(i)?a . Mutex[i, 1, a]
+ when (c > 0) ( lock(i)?(a) . Mutex[i, c + 1, a]
               + multilock(i)?r . Mutex[i, c + r, a]
               + fullunlock(i)!c . Mutex[i, 0, a]
               + unlock(i)?a2 . ( when (a == a2) Mutex[i, c - 1, a]
                                + when (a != a2) exception!.0))
Mutex_cons[next_i] := mutex_create!next_i
                    . (Mutex_cons[next_i + 1] | Mutex[next_i, 0, 0])
```

Condition Synchronization in CCS

```
WaitRoom[i, c] :=
  signal(i)? . (when (c == 0) WaitRoom[i, c]
              + when (c  > 0) wait(i)? . WaitRoom[i, c - 1])
  + add(i)? . WaitRoom[i, c + 1]
  + signal_all(i)? . WaitDistributor[i, c] ; WaitRoom[i, 0]

WaitDistributor[i, c] := when (c <= 0) 1
                       + when (c > 0) wait(i)? . WaitDistributor[i, c - 1]

WaitRoom_cons[next_i] := wait_create!next_i
                       . (WaitRoom_cons[next_i + 1] | WaitRoom[next_i, 0])
```

Summary / Cheat Sheets Update



We're continuing with summaries and cheat sheets as planned:

- You can bring a fresh cheat sheet to the re-midterm, final exam, and reexam, respectively.
  same terms and conditions: hand-written, one-sided, A5
- You can submit summaries E, F, G, and H, and will receive them at at the final exam.
  Summaries A-D will not be available in the final exam.
- For the re-midterm and reexam, you will receive summaries A-D and E-H, respectively.

2	3	4	5	6
Colloquium E 14:15 **Lecture F1**	Colloquium E 10:15 **Lecture F2**	08:15 **Tutorial E** 10:15 **Tutorial E**	14:00 **Concurrency Café**	14:00 **Concurrency Café**
9	10	11	12	13
Colloquium F	Colloquium F	08:15 **Tutorial F** 10:15 **Tutorial F**	14:00 **Concurrency Café**	14:00 **Concurrency Café**
16	17	18	19	20
14:15 **Lecture G1**	10:15 **Lecture G2**			14:00 **Concurrency Café**