Monitors — Some History

In all my work on the formalisation of proof methods for sequential programming languages, I knew that I was only preparing the way for a much more serious challenge, which was to extend the proof technology into the realm of concurrent program execution. I took as my first model of concurrency a kind of quasi-parallel programming (co-routines), which was introduced by Ole-Johan Dahl and Kristen Nygaard into Simula (and later Simula 67) for purposes of discrete event simulation [28, 29]. I knew the Simula concept of an object as a replicable structure of data, declared in a class together with the methods which are allowed to update its attributes. [...]

As in the case of proof-driven program development, it is the obligation of correctness that should drive the design of a good programming language feature. Of course, efficiency of implementation is also important. A correct implementation of the abstraction has to prevent more than one process from updating the concrete representation at the same time. This is efficiently done by use of Dijkstra's semaphores protecting critical regions [32]; the resulting structure was called a monitor [33, 34]. The idea was simultaneously put forward and successfully tested by Per Brinch Hansen in his efficient implementation of Concurrent PASCAL [35]. The monitor has since been adopted for the control of concurrency by the more recently fashionable language Java [36], but with extensions that prevent the use of the original simple proof rules.

From C.A.R. Hoare, 'Assertions: a personal perspective'

Monitors

Abstract data types (classes) that ensure mutual exclusion between operations (methods).

- Can’t access ’permanent’ (member) variables except through operations.
- Operations cannot access variables outside monitor (i.e., can only access permanent variables, local variables and parameters).
- Permanent variables are initialized before any operation can be invoked (constructor).
- Mutual exclusion is implicit. *At most 1 thread occupies the monitor.*
- Implementation: Threads are delayed on an “entry queue” until the monitor is unoccupied.
- Conditional Synchronization through *condition* variables.

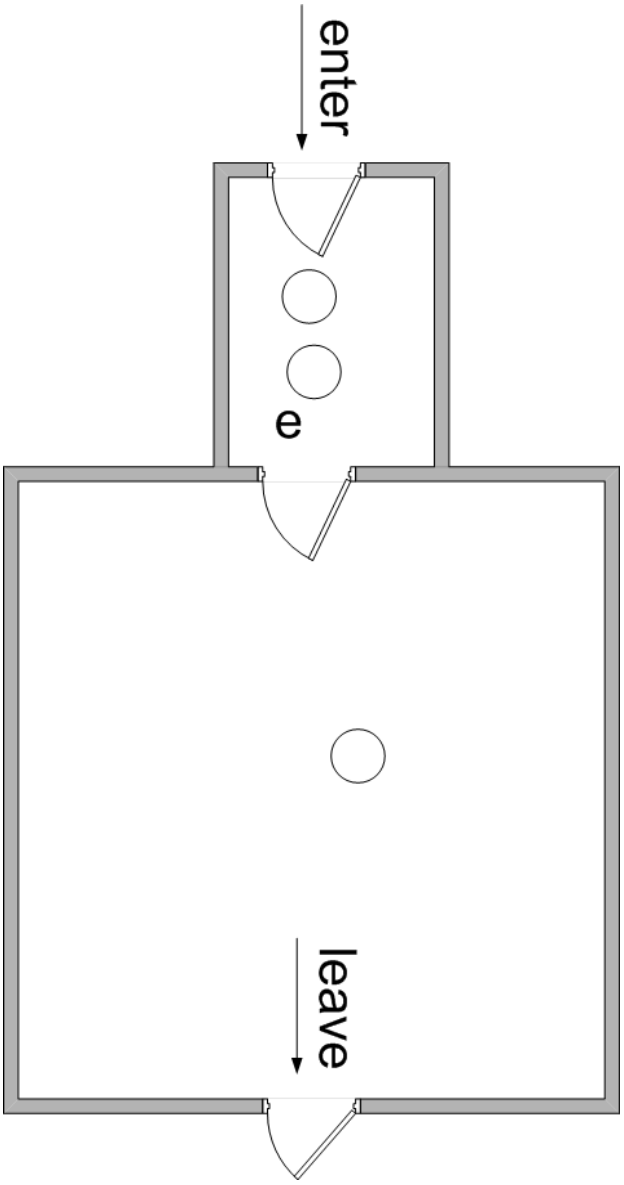
Example: Time Of Day

```
monitor TOD {
    int hr := 0, min := 0, sec := 0 ;
    ## Inv: 0 _< hr < 24
    ## Inv: 0 _< min < 60
    ## Inv: 0 _< sec < 60

    procedure set( int h, int m, int s ) {
    ## Pre: 0 _< h < 24 and 0 _< m < 60
    ## and 0 _< s < 60
        hr := h ; min := m ; sec := s ; }

    procedure get( int &h, int &m, int &s ) {
        h := hr ; m := min ; s := sec ; }

    procedure tick() {
        sec += 1 ;
        min += sec / 60 ; sec := sec % 60 ;
        hr += min / 60 ; min := min % 60 ;
        hr := hr % 24 ; } }
```



A monitor with no condition variables.

Monitor Invariant

M — an assertion

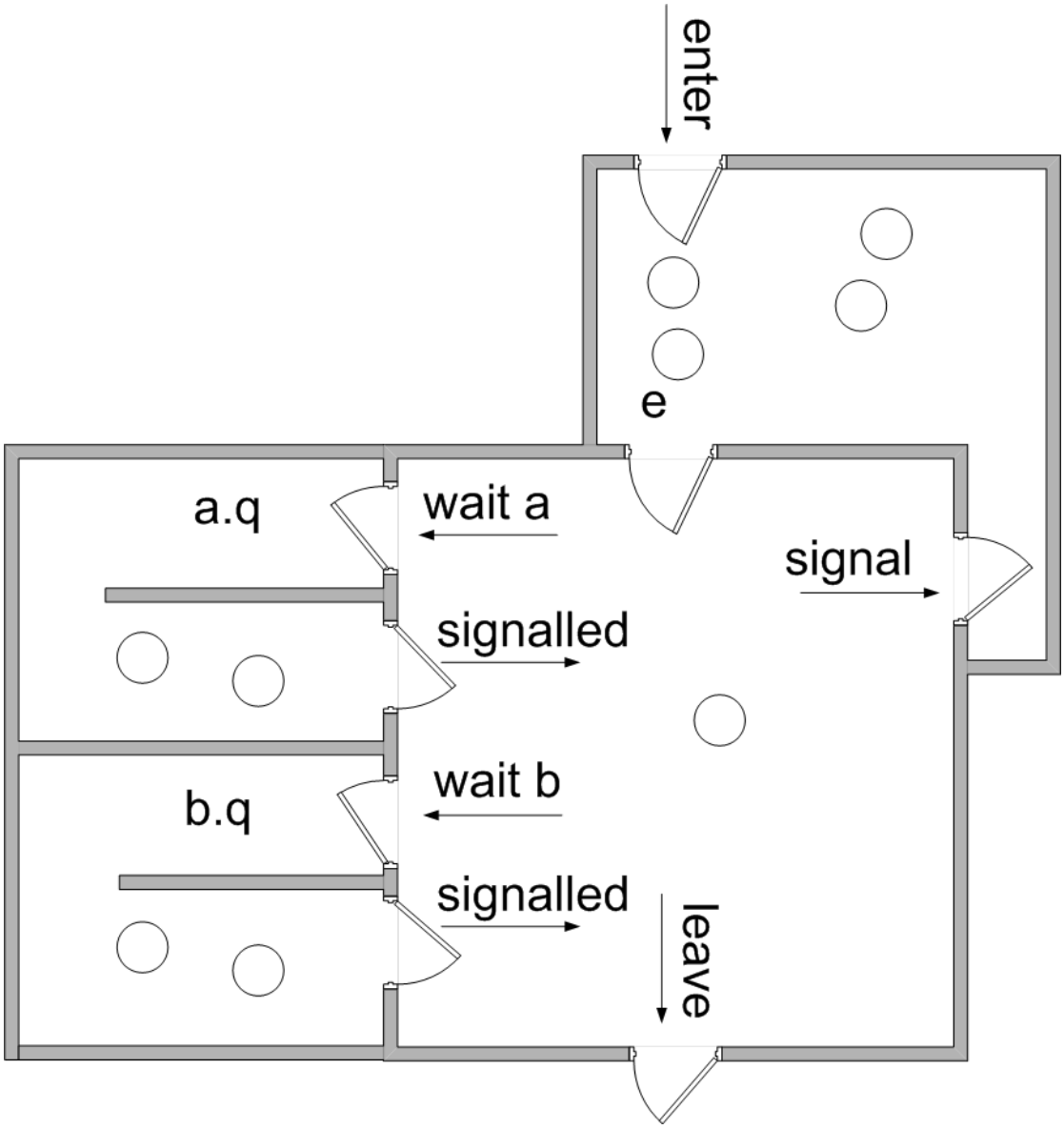
- M should be true when the monitor is “unoccupied”.
- Captures consistency (sanity, invariant) properties of data.
- In terms of permanent (thread global) variables only.
- Ensure that initialization makes it true.
- Ensure that $\{M\}\ op\ \{M\}$ (all public methods keep it true)
- Ensure that it is true at any wait points.

Condition variables — Signal and Wait (SW) version

cond c; — only used within monitor.

— associated with a "condition queue"

- **wait** c ; Leave the monitor and wait on c's condition queue.
- **signal** c;
  - Pass occupancy to (and wake up) some thread waiting on c's condition queue (if any).
  - Wait on the monitor's entry queue.
- Since occupancy of the monitor is passed seamlessly from the signaller to the signalee, any facts about the monitor's data will remain true between the start of the signal and the end of the wait.



Signal and Wait Discipline

Example: Bounded Buffer

Pseudo-code

---

module Bounded_buffer {

char $buf[N]$; # buffer

int $front := 0$; # first full slot

int $count := 0$; # number of full slots

Inv: $0 \leq front < N$ and $0 \leq count\_ < N$

procedure deposit(**char** $data$) {

$\langle$**await**$(count < N)$;

Inv and $count < N$

$buf[(front + count)\%N] := data$;

$count+ = 1$; ## $count > 0$

$\rangle$ }

procedure fetch(**char** $data$) {

$\langle$**await**$(count > 0)$;

Inv and $count > 0$

$result := buf[front]$;

$front := (front + 1)\%N$;

$count- = 1$; ## $count < N$

$\rangle$} }

Example: Bounded Buffer

---

monitor Bounded_buffer { # Signal and Wait version
    char $buf[N]$; # buffer
    int $front := 0$; # first full slot
    int $count := 0$; # number of full slots
    ## Inv: $0 \leq front < N$ and $0 \leq count\_ < N$
    cond not_full; # signaled only when $count < N$
    cond not_empty; # signaled only when $count > 0$
    procedure deposit(**char** $data$) {
        if ($count == N$) **wait**(not_full);
        ## Inv and $count < N$
        $buf[(front + count)\%N] := data;$
        $count+ = 1$; ## $count > 0$
        signal(not_empty); }
    procedure fetch(**char** $data$) {
        if ($count == 0$) **wait**(not_empty);
        ## Inv and $count > 0$
        $result := buf[front];$
        $front := (front + 1)\%N;$
        $count- = 1$; ## $count < N$
        signal(not_full); } }

Example: Semaphore

```
monitor Semaphore {
    int s := 0 ; ## Inv: s ≥ 0
    cond not_zero ; # signaled only when s > 0

    procedure init( int new_s ) {
        ## Pre: new_s ≥ 0
        s := new_s; }

    procedure V() {
        s+ = 1 ;
        signal( not_zero ) ; }

    procedure P() {
        if( s = 0 ) wait( not_zero ) ;
        ## s > 0
        s− = 1 ; }
}
```

Simple semantics—signal and wait (SW)

We assume that the programmer has associated an assertion P_c with each condition variable c.

M is the monitor invariant, respected by all public operations.

P_c and M must not depend on the "state" of the queues.

L_0 and L_1 are assertions that only involves variables local to the thread (and may differ for each occurence of **signal** or **wait**).

$$\text{signal axiom} : \{P_c \wedge M \wedge L_0\}\ \mathbf{signal}(c)\ \{M \wedge L_0\}$$
$$\text{wait axiom} : \{M \wedge L_1\}\ \mathbf{wait}(c)\ \{P_c \wedge M \wedge L_1\}$$

- Since occupancy of the monitor passes from the signaller to the waiter without interruption, if P_c is true prior to every $\mathsf{signal}(c)$ it will also be true after each $\mathsf{wait}(c)$.
- Local variables of the threads are unaffected.
- Since waiting yields occupancy, we must ensure M is true before waiting.
- Since the signaller reenters when the monitor becomes unoccupied, it can assume M after the signal is complete.
- Since signalling leaves the monitor unoccupied, if the queue is empty, we should ensure M is true before signalling.

Obligations and benefits

signal axiom : $\{P_c \wedge M \wedge L_0\}$ $\mathbf{signal}(c)$ $\{M \wedge L_0\}$

wait axiom : $\{M \wedge L_1\}$ $\mathbf{wait}(c)$ $\{P_c \wedge M \wedge L_1\}$

	Obligation of monitor	Benefit to monitor
$\mathbf{signal}(c)$	Ensure M and P_c before	M is true after
$\mathbf{wait}(c)$	Ensure M is true before	P_c and M are true after

New Operation

- **conditional_wait(** c **)** $\triangleq$ **{ if(** !(P_c) **) wait(** c **) ; }**

Length and empty

Sometimes we want to query the queues.

Often want to know if the queue is empty.

New Variable

- **length(** c **)** The number of threads on c's queue

New Operation

- **empty(** c **)** abbreviates $\mathbf{length}(c) == 0$

Example: In Semaphore we might want an invariant

$$\text{Inv}: \ s > 0 \Rightarrow \textbf{empty}(\textit{not\_zero}) \ ;$$

to show that threads do not wait without necessity.

Improved semantics—signal and wait (SW)

We allow M and P_c to mention $\mathsf{length}(d)$ for any condition variable d.

If the signaller knows that the queue is not empty, then it does not need to establish M.

If the signaller knows that the queue is empty, then it does not need to establish P_c.

$$\begin{array}{rl} \text{signal axiom}: & \{(\textit{if } \mathbf{empty}(c) \textit{ then } M \textit{ else } P'_c) \wedge L_0\} \\ & \mathbf{signal}(c) \\ & \{M \wedge L_0\} \\ \text{wait axiom}: & \{M' \wedge L_1\}\ \mathbf{wait}(c)\ \{P_c \wedge L_1\} \end{array}$$

where

$$\begin{aligned} P'_c &\triangleq (P_c)_{\mathsf{length}(c)\leftarrow\mathsf{length}(c)-1} \\ M' &\triangleq M_{\mathsf{length}(c)\leftarrow\mathsf{length}(c)+1} \end{aligned}$$

(Recall that $Q_{x\leftarrow E}$ means Q with each unbound occurrence of x replaced by E)

Sufficient signalling

The rules above only guarantee safety. There is still the worry that a thread will be stuck forever on a wait

Sufficient signalling: For each wait, there will be a subsequent signal (unique to that wait).

We can ensure sufficient signalling if M implies that there are no threads waiting that could be signalled.

For each c we should have $M \wedge P_c \Rightarrow \mathsf{length}(c) = 0$.

Sufficient signalling for semaphores

(Optional example.)

[After Howard 'Monitor Proofs', CACM 1976]

Here is a semaphore monitor again. We count:

- successful increments v
- attempts to decrement a
- successful decrements p
- Note that $s = v - p$

Pseudo code

---

module Semaphore {

```
    int v := 0 , a := 0 , p := 0 ;

    procedure V() {
        ⟨v+ = 1;⟩
    }

    procedure P() {
        ⟨a+ = 1;⟩
        ⟨await(v > p) p+ = 1;⟩
    }
}
```

The invariant:

- $a \geq p$ — attempts to decrement $\geq$ successful decrements
- $v \geq p$ — increments $\geq$ decrements. (I.e. $s \geq 0$)
- So far, we have $\min(a, v) \geq p$.
- $a > p \Rightarrow v \leq p$ — If any decrementing thread is waiting, it is because s can not be decremented. Rewriting this we get $a \leq p \vee v \leq p$, which is $\min(a, v) \leq p$
- Sumarizing we have $M = (\min(a, v) = p)$.

The last condition ($\min(a, v) \leq p$) is not required for safety, but is required for progess. It says that no thread waits in the P operation if it could proceed.

Progress: For P_c take $a > p = v - 1$. ($a > p$ means that there is a waiting thread —hence $\neg$empty(c). And $p = v - 1$ means $s = 1$ and so we signal as soon as possible, but no sooner.) Now M and P_c contradict each other and so (trivially)

$$M \wedge P_c \Rightarrow \text{ anything you like}$$

```
monitor Semaphore {
    int v := 0 , a := 0 , p := 0 ;
    ## inv: min(a, v) = p
    cond c ; # signaled only when a > p = v − 1

    procedure V() {
        ## min(a, v) = p
        v+ = 1 ;
        ## min(a, v − 1) = p
        if( p < a )
            ## a > p = v − 1
            signal( c ) ;
        else ## p = a ≤ v − 1, therefore p = a < v
```

```
            skip
        ## min(a, v) = p
    }

    procedure P() {
        ## min(a, v) = p
        a+ = 1 ;
        ## min(a − 1, v) = p
        if( v < a )
            ## v < a ∧ p = v, therefore min(a, v) = p
            wait( c ) ; ## a > p = v − 1
        else ## v > p = a − 1
            skip
        ## min(a − 1, v − 1) = p,
        ## therefore min(a, v) = p + 1
        p+ = 1 ;
        ## min(a, v) = p
    }
}
```

Exercise: Does our earlier implementation of semaphores have sufficient signalling? Can you dis/prove it?

Monitors versus Semaphores.

Passing a P operation indicates that something happened in the past — i.e. that a semaphore was incremented. It does not directly say much about the current state of the program. Consider

---

init: sem $s := 0$
P0:
    {B}
    V(s)
P1:
    P(s)
    {B was true in the past}

Passing a wait operation means that something is true of the state right now. Data encapsulation ensures that no other thread can interfere with the assertion.

Monitors are not more powerful than semaphores. But they are easier to use.

Rendezvous

We need a Rendezvous object to mediate communications between

- Several clients and
- Several servers.

1 or more clients call Reply submitRequest(Request) ;

Each server behaves as follows

---

while(true) {
    Request req := getRequest() ;
    Reply rep ;
    compute rep *from* req
    setReply(rep) ; }

The protocol cycles through 4 states.

---

enum State {$READY$, REQ_SENT, $REQ\_RECEIVED$, $REPLY\_SET$} ;

Pseudo-code solution

```
module Rendezvous {
    Request request ;
    Reply reply ;
    State state := READY ;

    procedure Reply submitRequest( Request req ) {
        ⟨await( state == READY )
         request := req; state := REQ_SENT;⟩
        ⟨await( state == REPLY_SET )
         Reply rep := reply; state := READY;
         return rep ; ⟩ }

    procedure Request getRequest() {
        ⟨await( state == REQ_SENT );
         Request req := request ;
         state := REQ_RECEIVED ;
         return req ; ⟩ }

    procedure void setReply( Reply rep ) {
        ⟨reply := rep; state := REPLY_SET;⟩ } }
```

Moving to a condition based solution. Introduce three new variables.

---

cond $Ready$; # signalled only when $state == READY$
cond $ReplySet$; # s.o.w. $state == REPLY\_SET$
cond $ReqSent$; # s.o.w. $state == REQ\_SENT$

Replace the three awaits with

---

conditional_wait($Ready$ **)**
conditional_wait($ReplySet$ **)**
conditional_wait($ReqSent$ **)**

Replace the assignments to state with

$state := READY$; **signal(** $Ready$ **)** ;
$state := REPLY\_SET$; **signal(** $ReplySet$ **)** ;
$state := REQ\_SENT$; **signal(** $ReqSent$ **)** ;

```
monitor Rendezvous {
    Request request ;
    Reply reply ;
    State state := READY ;
    cond Ready ; # s.o.w. state == READY
    cond ReplySet ; # s.o.w. state == REPLY_SET
    cond ReqSent ; # s.o.w. state == REQ_SENT

    procedure Reply submitRequest( Request req ) {
        conditional_wait( Ready )
        request := req ;
        state := REQ_SENT ; signal( ReqSent ) ;
        conditional_wait( ReplySet )
        Reply rep := reply ;
        state := READY ; signal( Ready ) ;
        return rep ;}

    procedure Request getRequest() {
        conditional_wait( ReqSent )
        Request req := request ;
        state := REQ_RECEIVED ;
        return req ; }
```

procedure void setReply(Reply rep) {
    $reply := rep;$
    $state := REPLY\_SET$; **signal**($ReplySet$) ; } }

Daisy-Chaining

Waking up a number of waiting threads.

Example Problem: Voters. N threads reach a barrier and must agree on a bit.

```
monitor Vote { # INCORRECT ATTEMPT!
    int for := 0, against := 0 ;
    ## Inv: 0 ≤ for ∧ 0 ≤ against ∧ for + against < N
    cond allDone ; #  s. o. w. for + against = N

    procedure bool cast( bool vote ) {
    if( vote ) ++for ; else ++against ;
    bool result
    if( for + against < N ) {
        wait( allDone ) ;
        result := for > against ; }
    else {
        result := for > against ;
        while( ! empty(allDone) ) signal( allDone ) ;
        for := against := 0 ; }
    return result ; } }
```

Idea is that last thread to vote wakes up all the others and resets the monitor.

But threads are allowed to leave prior to monitor being reset.

(Note violation of invariant.)

A correct solution delays each thread until the monitor is reset. Each thread wakes up the next.

Last thread awakened is the first one out and resets the monitor.

Note that *the invariant is not true* when the **signal** occurs — Must use 'improved semantics'.

```
monitor Vote {
    int for := 0, against := 0 ;
    ## Inv. 0 ≤ for ∧ 0 ≤ against ∧ for + against < N
    cond allDone ; # s. o. w. for + against == N
    procedure bool cast( bool vote ) {
        if( vote ) ++for ; else ++against ;
        conditional_wait( allDone ) ;
        ## for + against == N
        bool result := for > against;
        if( ! empty( allDone ) ) {
            signal( allDone ) ; }
        else {
            for := against := 0 }
        return result ;} }
```

Implementing Monitors With Semaphores (Signal and Wait)

Use ‘passing the baton’

- **sem** e :=1 ; # Semaphore for mutual exclusion (baton)
- **sem** q_c := 0 ;# Semaphore for each condition c
- **int** δ_c := 0 ; # Count of threads waiting for c

Now

- At the start of each public procedure: P(e)
- On return from each public procedure: V(e)
- **wait** c $\longrightarrow$ $\delta_c += 1$; V(e); P(q_c); $\delta_c -= 1$;
- **signal** c $\longrightarrow$ **if**($\delta_c > 0$) { V(q_c); P(e); }

Why it works:

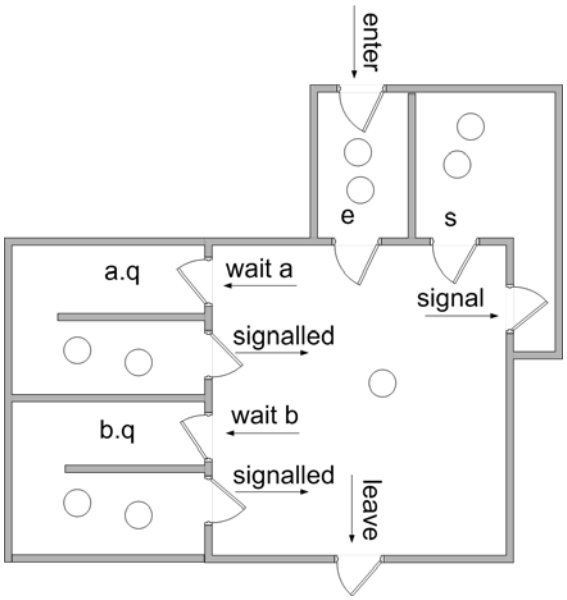
- The e semaphore enforces exclusive access to the monitor.
- The thread that has exclusive access can
  - give up access by returning from a public method, calling wait or calling signal when no thread is waiting on c, or
  - pass the exclusive access to another thread by calling signal when some thread is waiting on c.

Monitor Variants

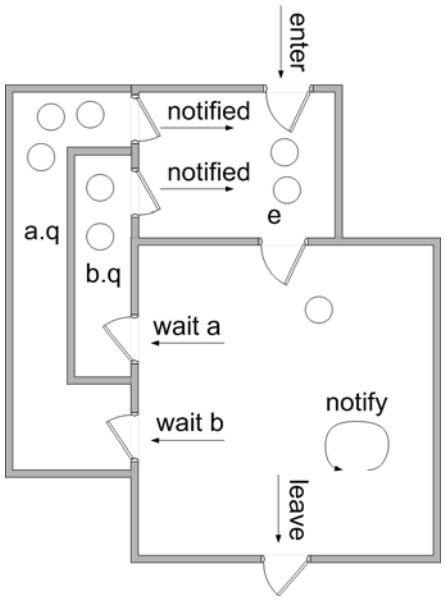
Signaling Disciplines

What happens to thread that calls signal?

- **Signal and Wait (SW):** Signaller moves to entry queue, signalled thread immediately enters the monitor.
- **Signal and Urgent Wait (SU):** signaller is put on the entry queue with high priority.
- **Signal and Exit (SX):** signaller leaves the monitor immediately
- **Signal and Continue (SC):**
  - Signaller retains occupancy, signaled thread is moved to the entry queue.
  - signal(c) is often written notify(c).



SU — Signal and Urgent wait.



SC — Signal and Continue

Programming with Signal and Continue

Since the signalled thread must compete with other threads on the entry queue, there is no reason to believe that P_c is true after wait. We only know it was "recently" true.

To ensure P_c is true we check it after awaking:

do { wait(c) ; } **while(** $\neg P_c$) ; ## P_c

Therefore:

conditional_wait(c) $\triangleq$ $\{$**while**($\neg P_c$) wait(c) ; $\}$ ## P_c

Sufficient signalling is now more difficult since a thread may wait multiple times, it may need to be signalled more times.

Control: It becomes harder to control the order of thread execution..

Semantics for Signal and Continue (SC)

Since P_c is not guaranteed on returning from , it need not be ensured prior to signalling.

$$\text{signal axiom (SC)} : \{L'_0\}\ \mathbf{signal}(c)\ \{L_0\}$$
$$\text{wait axiom (SC)} : \{M' \wedge L_1\}\ \mathbf{wait}(c)\ \{M \wedge L_1\}$$

where L_1 depends only on local variables and M is the monitor invariant.

$$L'_0 \triangleq (L_0)_{\mathsf{length}(c) \leftarrow \max(0, \mathsf{length}(c)-1)}$$
$$M' \triangleq M_{\mathsf{length}(c) \leftarrow \mathsf{length}(c)+1}$$

Example: Bounded Buffer—Signal and Continue (SC)

---

monitor Bounded_buffer {
    char $buf[n]$; # buffer
    int $front := 0$; # first full slot
    int $count := 0$; # number of full slots
    cond not_full; # signaled only when $count < n$
    cond not_empty; # signaled only when $count > 0$

    procedure deposit(**char** $data$) {
        while ($count == n$) **wait**(not_full);
        $buf[(front + count)\%n] := data$;
        $count := count + 1$;
        signal(not_empty); }

    procedure fetch(**char** &$data$) {
        while($count == 0$) **wait**(not_empty);
        $data := buf[front]$;
        $front := (front + 1)\%n$;
        $count := count - 1$
        signal(not_full); } }

Signal-all

signal_all(c) move all threads waiting on c to the entry queue.

Makes sense for SC (not for SW)

$$\text{signal all axiom (SC)} : \{L_2'\}\ \mathbf{signalAll}(c)\ \{L_2\}$$
$$L_2' \triangleq (L_2)_{\mathsf{length}(c)\leftarrow 0}$$

Variants: Priority waiting.

Better control of which thread is signalled.

Operation	Semantics
$\mathsf{wait}(c, rank)$	Priority wait, lowest $rank$ awakened first
$\mathsf{minrank}(c)$	Value of lowest rank waiting

Example disk controller.

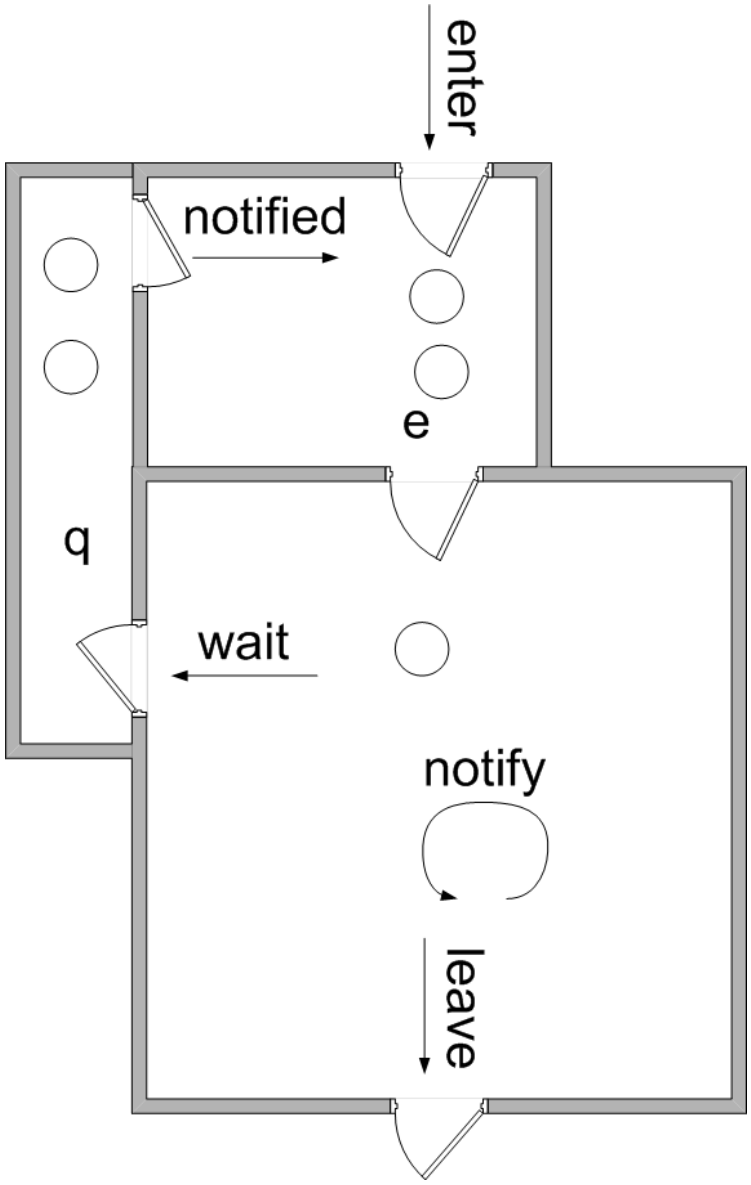
---

wait until my turn
move disk head to required cylinder
read or write data
signal next waiter

For efficiency, threads are awakened in order of increasing cylinder (for out-swing) or decreasing cylinder (for in-swing).

Java's built-in "Monitors"

- Keyword `synchronized` declares object (method, section of code) to be critical section.
- Class with private data and all public methods `synchronized` is a monitor.
- Signal and continue discipline.
- No explicit condition variables — just call `wait()`.
- Only one wait queue per object. Effectively one condition variable.
- Signal with `notify()` or `notifyAll()`.
- `wait()`, `notify()`, and `notifyAll()` belong to class `Object` — every object has an entry queue and a wait queue.



Java style monitors. One queue is shared by all conditions.

Example: Bounded Buffer:

```
class BoundedBuffer {
        private char[] buf;
        private int front = 0, count = 0, n;

        public BoundedBuffer(int n) {
                this.n = n ; buf = new char[n] ; }

        public synchronized void deposit(char data)
        throws InterruptedException {
                while (count == n) wait() ;
                buf[(front+count) % n] = data ; count++ ;
                notifyAll(); }

        public synchronized char fetch()
        throws InterruptedException {
                while (count == 0) wait();
                char result = buf[front];
                front = (front+1)%n ; count-- ;
                notifyAll();
                return result; } }
```