Concurrent Programming in Harmony: Signaling and Conditional Critical Sections

CS 4410
Operating Systems

[Robbert van Renesse]
Remember the recruiter...

Asked >100 candidates if they could implement two threads, where one thread had to wait for a signal from the other

none of them were able to do it without hints
only some of them were able to do it with hints

(as far as I know, none of them were Cornell grads ;-)

Can be done with busy-waiting

def T0():
    while not done:
        pass;
    

def T1():
    done = True;

    spawn T0();
    spawn T1();
    done = False;
Can be done with busy-waiting

```python
def T0():
    while not done:
        pass;
    done = False;

def T1():
    done = True;
    spawn T0();
    spawn T1();
```

we don’t like busy waiting
Can be done with busy-waiting

def T0():
    await done;
;
def T1():
    done = True;
;
    done = False;
spawn T0();
spawn T1();

we don’t like busy waiting
Can be done with locks, awkwardly

```
import synch;

def T0():
    lock(?condition);
    assert done;  # make sure T1 sent signal
    # no unlock
;

def T1():
    # no lock
done = True;
    unlock(?condition);
;

done = False;
condition = Lock();
lock(?condition);  # weird stuff during init...
spawn T0();
spawn T1();
```
Can be done with locks, awkwardly

```python
import synch;

def T0():
    lock(?condition);
    assert done;   # make sure T1 sent signal
    # no unlock
;

def T1():
    # no lock
    done = True;
    unlock(?condition);
;
    done = False;
    condition = Lock();
    lock(?condition);
    # weird stuff during init…

spawn T0();
spawn T1();
```

locks should be nested
Enter *(binary)* semaphores

[Dijkstra 1962]
Binary Semaphore

- Two-valued counter: 0 or 1
- Two operations:
  - **P**(rocure)
    - waits until counter is 1, then sets the counter to 0. Akin to decrementing
  - **V**(ocate)
    - can only be called legally if the counter is 0. Sets the counter to 1. Akin to incrementing
- No operation to read the value of the counter!
## Difference with locks

<table>
<thead>
<tr>
<th>Locks</th>
<th>(Binary) Semaphores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initially “unlocked”</td>
<td>Can be initialized to 0 or 1</td>
</tr>
<tr>
<td>Usually locked, then unlocked by same process (although see R/W lock)</td>
<td>Can be procured and vacated by different processes</td>
</tr>
<tr>
<td>Either held or not</td>
<td>Can be easily generalized to counting semaphores</td>
</tr>
<tr>
<td>Mostly used to implement critical sections</td>
<td>Can be used to implement critical sections as well as waiting for special conditions</td>
</tr>
</tbody>
</table>

but both are much like “batons” that are being passed
Counting Semaphores?

• Book starts with counting semaphores
• We will start concentrate on binary semaphores...
Binary Semaphore interface and implementation

```python
def Semaphore(cnt):
    result = cnt;

    def P(sema):
        blocked = True:
        atomic:
            if (!sema) > 0:
                !sema -= 1;
                blocked = False;

    def V(sema):
        atomic:
            !sema += 1;
```

- `sema = Semaphore(0 or 1)`

  P(?sema) “procures” sema

  This means that it tries to decrement the semaphore, blocking if it is 0.

- V(?sema) “vacates” sema

  This means incrementing the semaphore.
import synch;

def T0():
    P(?condition);  # wait for signal
    assert done;
;
def T1():
    done = True;
    V(?condition);  # send signal
;

done = False;
condition = Semaphore(0);
spawn T0();
spawn T1();
Semaphores can be locks too

- $lk = \text{Semaphore}(1)$  \# 1-initialized
- $P(?lk)$  \# lock
- $V(?lk)$  \# unlock
Great, what else can one do with binary semaphores??
Conditional Critical Sections

• A critical section with a condition
• For example:
  • dequeue(), but wait until the queue is non-empty
    – don’t want two threads to run dequeue code at the same time, but also don’t want any thread to run dequeue code when queue is empty
  • print(), but wait until the printer is idle
  • acquire_rlock(), but only if there are no writers in the critical section
  • allocate 100 GPUs, when they become available
  • …

[Hoare 1973]
Multiple conditions

Some conditional critical sections can have multiple conditions:

- R/W lock: readers are waiting for writer to leave; writers are waiting for reader or writer to leave
- bounded queue: dequeuers are waiting for queue to be non-empty; enqueuers are waiting for queue to be non-full
- …
High-level idea: selective baton passing!

• When a process wants to execute in the critical section, it needs the one baton
• Processes can be waiting for different conditions
  • such processes do not hold the baton
• When a process with the baton leaves the critical section, it checks to see if there are processes waiting on a condition that now holds
• If so, it passes the baton to one such process
• If not, the critical section is vacated and the baton is free to pick up for another process that comes along
“Split Binary Semaphores” [Hoare 1973]

• Implement baton passing with multiple binary semaphores
• If there are $N$ conditions, you’ll need $N+1$ binary semaphores
  • one for each condition
  • one to enter the critical section in the first place
• At most one of these semaphores has value 1
  • If all are 0, baton held by some process
  • If one semaphore is 1, no process holds the baton
    – if it’s the “entry” semaphore, then no process is waiting on a condition that holds, and any process can enter
    – if it’s one of the condition semaphores, some process that is waiting on the condition can now enter the critical section
Bathroom humor…

3 processes want to enter critical section

- Bathroom: critical section
- Bedrooms: waiting conditions

At any time exactly one semaphore or process is green (and thus at most one semaphore is green)
This is a model of:

- **Reader/writer lock:**
  - Bathroom: critical section
  - Bedroom 1: readers waiting for writer to leave
  - Bedroom 2: writers waiting for readers or writers to leave

- **Bounded queue:**
  - Bathroom: critical section
  - Bedroom 1: dequeuers waiting for queue to be non-empty
  - Bedroom 2: enqueueuers waiting for queue to be non-full

- ...
Bathroom humor...

3 processes want to enter critical section

Bathroom: critical section
Bedrooms: waiting conditions

at any time exactly one semaphore or process is green
Bathroom humor…

1 process entered the critical section

Bathroom: critical section
Bedrooms: waiting conditions

at any time exactly one semaphore or process is green
Bathroom humor…

Process needs to wait for Condition 1

Bathroom: critical section
Bedrooms: waiting conditions

At any time exactly one semaphore or process is green
Bathroom humor…

no process waiting for condition that holds

Bathroom: critical section
Bedrooms: waiting conditions

at any time exactly one semaphore or process is green
Bathroom humor…

another process can enter the critical section

Bathroom: critical section
Bedrooms: waiting conditions

at any time exactly one semaphore or process is green

Bathroom: critical section
Bedrooms: waiting conditions

holds baton
does not hold baton
Bathroom humor…

process entered the critical section

Bathroom: critical section
Bedrooms: waiting conditions

at any time exactly one semaphore or process is green
Bathroom humor...

Bathroom: critical section
Bedrooms: waiting conditions

process enables Condition 1 and wants to leave

at any time exactly one semaphore or process is green
Bathroom humor...

process left, Condition 1 holds

Bathroom: critical section
Bedrooms: waiting conditions

holds baton

does not hold baton

at any time exactly one semaphore or process is green
Bathroom humor…

first process (and only first process) can enter critical section again

Bathroom: critical section
Bedrooms: waiting conditions

at any time exactly one semaphore or process is green
Bathroom humor...

first process entered critical section again

Bathroom: critical section
Bedrooms: waiting conditions

at any time exactly one semaphore or process is green
Bathroom humor…

First process leaves without either condition holding

Bathroom: critical section
Bedrooms: waiting conditions

at any time exactly one semaphore or process is green
Bathroom humor…

First process done

Bathroom: critical section
Bedrooms: waiting conditions

at any time exactly one semaphore or process is green
Bathroom humor...

One process want to enter the critical section

Bathroom: critical section
Bedrooms: waiting conditions

at any time exactly one semaphore or process is green
Bathroom humor...

Last process entered critical section

Bathroom: critical section
Bedrooms: waiting conditions

at any time exactly one semaphore or process is green
Bathroom humor...

Process needs to wait for Condition 2

Bathroom: critical section
Bedrooms: waiting conditions

at any time exactly one semaphore or process is green
Bathroom humor…

Process waiting for Condition 2

- Bathroom: critical section
- Bedrooms: waiting conditions

at any time exactly one semaphore or process is green
Reader/writer lock, again

```
39   mutex, r_sema, w_sema = Semaphore(1), Semaphore(0), Semaphore(0);
40   r_entered, r_waiting, w_entered, w_waiting = 0, 0, 0, 0;
```

Figure 15.1: [code/RWsbs.hny] Reader/Writer Lock using Split Binary Semaphores.

Accounting:
- `r_entered`: #readers in the critical section
- `r_waiting`: #readers waiting to enter the critical section
- `w_entered`: #writers in the critical section
- `w_waiting`: #writers waiting to enter the critical section

Invariants:
- if $n$ readers in the critical section, then $nreaders \geq n$
- if $n$ writers in the critical section, then $nwriters \geq n$
- $(nreaders \geq 0 \land nwriters = 0) \lor (nreaders = 0 \land 0 \leq nwriters \leq 1)$
Reader/writer lock, again

```python
def acquire_rlock():
    P(?mutex);
    if w_entered > 0:
        r_waiting += 1;
        V(?mutex); P(?r_sema);
        r_waiting -= 1;
    ;
    r_entered += 1;
    V_one();
    ;
def release_rlock():
    P(?mutex);
    r_entered -= 1;
    V_one();
    ;
```
Reader/writer lock, again

def acquire_rlock():
    P(?mutex);
    if w_entered > 0:
        r_waiting += 1;
        V(?mutex); P(?r_sema);
        r_waiting -= 1;
    ;
    r_entered += 1;
    V_one();
    ;
def release_rlock():
    P(?mutex);
    r_entered -= 1;
    V_one();
    ;
def acquire_wlock():
    P(?mutex);
    if (r_entered + w_entered) > 0:
        w_waiting += 1;
        V(?mutex); P(?w_sema);
        w_waiting -= 1;
    ;
    w_entered += 1;
    V_one();
    ;

def release_wlock():
    P(?mutex);
    w_entered -= 1;
    V_one();
    ;
Reader/writer lock, again

```python
def acquire_wlock():
    P(?mutex);
    if (r_entered + w_entered) > 0:
        w_waiting += 1;
        V(?mutex); P(?w_sema);
        w_waiting -= 1;
;
    w_entered += 1;
    V_one();
;
    def release_wlock():
        P(?mutex);
        w_entered -= 1;
        V_one();
;
```
When leaving critical section:

- if no writers in the Critical Section and there are readers waiting
  then let a reader in
- else if no readers nor writer in the C.S. and there are writers waiting
  then let a writer in
- otherwise
  let any new process in
Reader/writer lock, again

```python
import synch;

def V_one():
    if (w_entered == 0) and (r_waiting > 0): V(?r_sema);
    elif ((r_entered + w_entered) == 0) and (w_waiting > 0): V(?w_sema);
    else: V(?mutex);
```

When leaving critical section:
- if no writers in the Critical Section and there are readers waiting then let a reader in
- else if no readers nor writer in the C.S. and there are writers waiting then let a writer in
- otherwise let any new process in
  - Can the two conditions be reversed?
  - What is the effect of that?
Split Binary Semaphore rules

- $N+1$ binary semaphores
  - 1 "entry" semaphore and $N$ condition semaphores
- Initially only the “entry” semaphore is 1
- Sum of semaphores should always be 0 or 1
  - each process should start with a P operation, alternate V and P operations, and end on a V operation
  - never two Ps or two Vs in a row!!!!
- Keep careful track of state in shared variables
  - including one `#waiting` counter per condition
- Only access variables when sum of semaphores is 0

This “recipe” works for any synchronization problem where the number of conditions is fixed
Making R/W lock starvation-free

• Last implementation suffers from starvation
Making R/W lock starvation-free

• Solution 1: change the waiting and release conditions:
  • when a reader tries to enter the critical section, wait if there is a writer in the critical section OR if there are writers waiting to enter the critical section
  • exiting reader prioritizes releasing a waiting writer
  • exiting writer prioritizes releasing a waiting reader

See Figure 16.1
Making R/W lock starvation-free

- Solution 2: maintain a FCFS queue of all processes trying to enter
  - use a semaphore per process rather than per condition
    - (i.e., each process has its own condition)
  - the queue contains the semaphores that the processes in the queue are waiting for
  - processes at head of queue are awakened when possible (in a baton-passing style)
  - Works with a variable #conditions too!!!

See Figure 16.2
Conditional Critical Sections

We now know two ways to implement them:

<table>
<thead>
<tr>
<th>Busy Waiting</th>
<th>Split Binary Semaphores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wait for condition in loop, acquiring lock before testing condition and releasing it if the condition does not hold</td>
<td>Use a collection of binary semaphores and keep track of state including information about waiting processes</td>
</tr>
<tr>
<td>Easy to understand the code</td>
<td>State tracking is complicated</td>
</tr>
<tr>
<td>Ok for true multi-core, but bad for virtual threads</td>
<td>Good for both multi-core and virtual threading</td>
</tr>
</tbody>
</table>
Language support?

• Can’t the programming language be more helpful here?
  • Helpful syntax
  • Or at least some library support
“Hoare” Monitors

- Tony Hoare 1974
- similar construct given by Per Brinch-Hansen 1973
- Syntactic sugar around split binary semaphores

```
single resource: monitor
begin busy: Boolean;
  nonbusy: condition;
procedure acquire;
  begin if busy then nonbusy.wait;
      busy := true
  end;
procedure release;
  begin busy := false;
      nonbusy.signal
  end;
  busy := false; comment initial value;
end single resource
```
“Hoare” Monitors

- Tony Hoare 1974
- similar construct given by Per Brinch-Hansen 1973
- Syntactic sugar around split binary semaphores

```plaintext
single resource: monitor
begin busy: Boolean;
   nonbusy: condition;
procedure acquire;
   begin if busy then nonbusy.wait;
       busy := true
end;
procedure release;
   begin busy := false;
       nonbusy.signal
end;
busy := false; comment initial value;
end single resource
```

```python
3     def acquire():
4         mon_enter();
5         if busy:
6             wait(?nonbusy);
7             busy = True;
8         mon_exit();
9     ;
10    def release():
11        mon_enter();
12        busy = False;
13        signal(?nonbusy);
14        mon_exit();
15    ;
16    mutex = Semaphore(1);
17    nonbusy = Condition(?mutex);
18    busy = False;
```
import synch;

def mon_enter():
    P(?mutex);
;
def mon_exit():
    V(?mutex);
;
def Condition(mon):
    result = dict{ .lock: mon, .sema: Semaphore(0), .count: 0 }; 
;
def wait(cond):
    cond→count += 1;
    V(cond→lock); P(?cond→sema);
    cond→count -= 1;
;
def signal(cond):
    if cond→count > 0:
        V(?cond→sema); P(cond→lock);
;
;
Mesa Monitors

- Introduced in the Mesa language
  - Xerox PARC, 1980
- Syntactically similar to Hoare monitors
- Semantically closer to busy waiting approach
Hoare vs Mesa Monitors

Hoare monitors
- Baton passing approach
- Signal passes baton

Mesa monitors
- Sleep + try again
- Notify(all) wakes sleepers

Mesa monitors won the test of time…
Mesa Monitors in Harmony

```python
def Condition(lk):
    result = dict{ .lock: lk, .waiters: [] };

def wait(c):
    atomic:
        unlock(c->lock);
        stop c->waiters;


def notify(c):
    atomic:
        let lk, waiters = c->lock, c->waiters:
          if waiters != []:
            lk->suspended += [waiters[0],];
            c->waiters = tail(waiters);


def notifyAll(c):
    atomic:
        let lk, waiters = c->lock, c->waiters:
          lk->suspended += waiters;
          c->waiters = [];
```

mon\_enter: grab lock

mon\_exit: release lock

**Condition**: consists of lock + list of processes waiting

**wait**: unlock + add process context to list of waiters

**notify**: move one waiter to the list of suspended processes associated with the lock

**notifyAll**: move all waiters to the list of suspended processes associated with the lock
R/W lock with Mesa monitors

Invariants:
- if $n$ readers in the critical section, then $n_{\text{readers}} \geq n$
- if $n$ writers in the critical section, then $n_{\text{writers}} \geq n$
- $(n_{\text{readers}} \geq 0 \land n_{\text{writers}} = 0) \lor (n_{\text{readers}} = 0 \land 0 \leq n_{\text{writers}} \leq 1)$

`rwlock` protects the $n_{\text{readers}}/n_{\text{writers}}$ variables, not the critical section!
R/W Lock, reader part

busy waiting

```python
def acquire_rlock():
    lock(?rwlock);
    while nwriters > 0:
        unlock(?rwlock);
        lock(?rwlock);
    nreaders += 1;
    unlock(?rwlock);

def release_rlock():
    lock(?rwlock);
    nreaders -= 1;
    unlock(?rwlock);
```

Mesa monitor

```python
def acquire_rlock():
    lock(?rwlock);
    while nwriters > 0:
        wait(?rcond);
        nreaders += 1;
        unlock(?rwlock);

    def release_rlock():
        lock(?rwlock);
        nreaders -= 1;
        if nreaders == 0:
            notify(?wcond);
        unlock(?rwlock);
```
R/W Lock, reader part

busy waiting

```python
def acquire_rlock():
    lock(?rwlock);
    while nwriters > 0:
        unlock(?rwlock);
        lock(?rwlock);
    nreaders += 1;
    unlock(?rwlock);

def release_rlock():
    lock(?rwlock);
    nreaders -= 1;
    unlock(?rwlock);
```

Mesa monitor

```python
def acquire_rlock():
    lock(?rwlock);
    while nwriters > 0:
        unlock(?rwlock);
        lock(?rwlock);

    nreaders += 1;
    unlock(?rwlock);

def release_rlock():
    lock(?rwlock);
    nreaders -= 1;
    if nreaders == 0:
        notify(?wcond);
    unlock(?rwlock);
```
R/W lock, writer part

busy waiting

Mesa monitor

```python
def acquire_wlock():
    lock(?rwlock);
    while (nreaders + nwriters) > 0:
        unlock(?rwlock);
        lock(?rwlock);
    nwriters = 1;
    unlock(?rwlock);

def release_wlock():
    lock(?rwlock);
    nwriters = 0;
    unlock(?rwlock);
```

```python
def acquire_wlock():
    lock(?rwlock);
    while (nreaders + nwriters) > 0:
        wait(?wcond);
    nwriters = 1;
    unlock(?rwlock);

def release_wlock():
    lock(?rwlock);
    nwriters = 0;
    notifyAll(?rcond);
    notify(?wcond);
    unlock(?rwlock);
```
R/W lock, writer part

```python
def acquire_wlock():
    lock(?rwlock);
    while (nreaders + nwriters) > 0:
        unlock(?rwlock);
        lock(?rwlock);
    ;
    nwriters = 1;
    unlock(?rwlock);
;
def release_wlock():
    lock(?rwlock);
    nwriters = 0;
    unlock(?rwlock);
;
```

Mesa monitor

```python
def acquire_wlock():
    lock(?rwlock);
    while (nreaders + nwriters) > 0:
        wait(?wcond);
    ;
    nwriters = 1;
    unlock(?rwlock);
;
def release_wlock():
    lock(?rwlock);
    nwriters = 0;
    notifyAll(?rcond);
    notify(?wcond);
    unlock(?rwlock);
;
What the recruiter wanted…

```python
import synch;

def T0():
    lock(?mutex);
    while not done:
        wait(?cond);
    ;
    unlock(?mutex);
;
def T1():
    lock(?mutex);
    done = True;
    notify(?cond);
    unlock(?mutex);
;
mutex = Lock();
cond = Condition(?mutex);
done = False;
spawn T0();
spawn T1();```