Limitations of Wait- and Lock-Free Algorithms

Wait-/Lock-Free algorithms are severely limited in terms of their computation:
- restricted to the semantics of a single atomic operation
- set of atomic operations is architecture specific, but often includes
  - exchange of a memory cell with a register
  - compare-and-swap of a register with a memory cell
  - fetch-and-add on integers in memory
  - modify-and-test on bits in memory
- provided instructions usually allow only one memory operand

---

**binary semaphores**: a flag that can be acquired (set) if free (unset) and released
**counting semaphores**: an integer that can be decreased if non-zero and increased

---

**mutex**: ensures mutual exclusion using a binary semaphore
**monitor**: ensures mutual exclusion using a binary semaphore, allows other threads to block until the next release of the resource

We will collectively refer to these data structures as **locks**.
Locks

A lock is a data structure that

- protects a **critical section**: a piece of code that may produce incorrect results when executed concurrently from several threads
- it ensures **mutual exclusion**: no two threads execute at once
- block other threads as soon as one thread executes the critical section
- can be **acquired** and **released**
- may cause **deadlock** the program

Semaphores and Mutexes

A (counting) semaphore is an integer \( s \) with the following operations:

```c
void wait() {
    bool avail;
    do {
        atomic { avail = s>0; 
        if (avail) s--;
        }
    } while (!avail);
}

void signal() {
    atomic { s = s + 1; }
}
```

A counting semaphore can track how many resources are still available.

- a thread requiring a resource executes `wait()`
- if a resource is still available, `wait()` returns
- once a thread finishes using a resource, it calls `signal()`

Special case: initializing with \( s = 1 \) gives a **binary** semaphore:

- can be used to block and unblock a thread
- can be used to protect a single resource
- in this case the data structure is also called **mutex**
Implementation of Semaphores

A *semaphore* does not have to wait busily:

```c
void wait() {
    bool avail;
    do {
        atomic {
            avail = s > 0;
            if (avail) s--;
        }
        if (!avail) de_schedule(&s);
    } while (!avail);
}
```

```c
void signal() {
    atomic {
        s = s + 1;
    }
}
```

Busy waiting is avoided by placing waiting threads into queue:

- a thread failing to decrease *s* executes `de_schedule()`
- `de_schedule()` enters the operating system and inserts the current thread into a queue of threads that will be woken up when *s* becomes non-zero, usually by *monitoring writes to* &s
- once a thread calls `signal()`, the first thread *t* waiting on &s is extracted
- the operating system lets *t* return from its call to `de_schedule()`

Practical Implementation of Semaphores

Certain optimisations are possible:

```c
void wait() {
    bool avail;
    do {
        atomic {
            avail = s > 0;
            if (avail) s--;
        }
        if (!avail) de_schedule(&s);
    } while (!avail);
}
```

In general, the implementation is more complicated

- `wait()` may busy wait for a few iterations
  - saves de-scheduling if the lock is released frequently
  - better throughput for semaphores that are held for a short time
- `signal()` might have to inform the OS that *s* has been written

```c
void signal() {
    atomic {
        s = s + 1;
    }
}
```

In general, the implementation is more complicated

- `wait()` may busy wait for a few iterations
  - saves de-scheduling if the lock is released frequently
  - better throughput for semaphores that are held for a short time
- `signal()` might have to inform the OS that *s* has been written

-- using a semaphore with a single thread reduces to `if (s) s--; s++;

```c
atomic {
    s = s + 1;
    if (avail) s--;
}
if (!avail) de_schedule(&s);
while (!avail);
```
Making a Queue Thread-Safe

Consider a double ended queue:

```
double-ended queue: adding an element
void PushLeft(DQueue* q, int val) {
    QNode *qn = malloc(sizeof(QNode));
    qn->val = val;
    // prepend node qn
    QNode* leftSentinel = q->left;
    QNode* oldLeftNode = leftSentinel->right;
    qn->left = leftSentinel;
    qn->right = oldLeftNode;
    leftSentinel->right = qn;
    oldLeftNode->left = qn;
}
```

Mutexes

One common use of semaphores is to guarantee mutual exclusion.
- in this case, a binary semaphore is also called a mutex
- add a lock to the double-ended queue data structure
- decide what needs protection and what not

```
double-ended queue: thread-safe version
void PushLeft(DQueue* q, int val) {
    QNode *qn = (QNode*) malloc(sizeof(QNode));
    qn->val = val;
    QNode* leftSentinel = q->left;
    wait(q->s); // wait to enter the critical section
    QNode* oldLeftNode = leftSentinel->right;
    qn->left = leftSentinel;
    qn->right = oldLeftNode;
    leftSentinel->right = qn;
    oldLeftNode->left = qn;
    signal(q->s); // signal that we're done
}
```

Mutexes

One common use of semaphores is to guarantee mutual exclusion.
- in this case, a binary semaphore is also called a mutex
- add a lock to the double-ended queue data structure
- decide what needs protection and what not

```
Mutexes
One common use of semaphores is to guarantee mutual exclusion.
- in this case, a binary semaphore is also called a mutex
- add a lock to the double-ended queue data structure
- decide what needs protection and what not

Implementing the Removal

By using the same lock q->s, we can write a thread-safe PopRight:

```
double-ended queue: removal
int PopRight(DQueue* q) {
    QNode* oldRightNode;
    QNode* leftSentinel = q->left;
    QNode* rightSentinel = q->right;
    wait(q->s); // wait to enter the critical section
    oldRightNode = rightSentinel->left;
    if (oldRightNode==leftSentinel) { signal(q->s); return -1; }
    QNode* newRightNode = oldRightNode->left;
    newRightNode->right = rightSentinel;
    rightSentinel->left = newRightNode;
    signal(q->s); // signal that we're done
    int val = oldRightNode->val;
    free(oldRightNode);
    return val;
}
```
Implementing the Removal

By using the same lock q->s, we can write a thread-safe PopRight:

double-ended queue: removal

```c
int PopRight(DQueue* q) {
    QNode* oldRightNode;
    QNode* leftSentinel = q->left;
    QNode* rightSentinel = q->right;
    wait(q->s); // wait to enter the critical section
    oldRightNode = rightSentinel->left;
    if (oldRightNode==leftSentinel) { signal(q->s); return -1; }
    QNode* newRightNode = oldRightNode->left;
    newRightNode->right = rightSentinel;
    rightSentinel->left = newRightNode;
    signal(q->s); // signal that we're done
    int val = oldRightNode->val;
    free(oldRightNode);
    return val;
}
```

- error case complicates code --- semaphores are easy to get wrong
- abstract common concept: take lock on entry, release on exit

Monitors: An Automatic, Re-entrant Mutex

Often, a data structure can be made thread-safe by

- acquiring a lock upon entering a function of the data structure
- releasing the lock upon exit from this function

Locking each procedure body that accesses a data structure:

1. is a re-occurring pattern, should be generalized
2. becomes problematic in recursive calls
3. if a thread t waits for a data structure to be filled:
   - t will call e.g. PopRight and obtain -1
   - t then has to call again, until an element is available
   - \( \triangledown \) t is busy waiting and produces contention on the lock

Implementation of a Basic Monitor

A monitor contains a mutex s and the thread currently occupying it:

```c
typedef struct monitor mon_t;
struct monitor { int tid; int count; }; void monitor_init(mon_t* m) { memset(m, 0, sizeof(mon_t)); }
```

Define `monitor_enter` and `monitor_leave`:

- ensure mutual exclusion of accesses to `mon.t`
- track how many times we called a monitored procedure recursively

```c
void monitor_enter(mon_t *m) {
    atomic {
        m->count--;
        if (m->count==0) {
            m->tid=0;
            // wake up threads
        }
    }
}
```

```c
void monitor_leave(mon_t *m) {
    atomic {
        bool mine = false;
        while (!mine) {
            atomic {
                m->count--;
                if (m->count==0) {
                    m->tid=0;
                    // wake up threads
                }
            }
        }
    }
}
```

```c
if (!mine) de_schedule(&m->tid);
```
Rewriting the Queue using Monitors

Instead of the mutex, we can now use monitors to protect the queue:

```c
double-ended queue: monitor version

void PushLeft(DQueue* q, int val) {
    monitor_enter(q->m);
    ...
    monitor_leave(q->m);
}
void ForAll(DQueue* q, void* data, void (*callback)(void*, int)) {
    monitor_enter(q->m);
    for (DNode* qn = q->left->right; qn != q->right; qn = qn->right)
        (*callback)(data, qn->val);
    monitor_leave(q->m);
}
```

Recursive calls possible:
- the function passed to ForAll can invoke PushLeft
- example: ForAll(q, q, &PushLeft) duplicates entries
- using monitor instead of mutex ensures that recursive call does not block

Condition Variables

- Monitors simplify the construction of thread-safe resources.
- Still: Efficiency problem when using resource to synchronize:
  - if a thread \( t \) waits for a data structure to be filled:
    - \( t \) will call e.g. PopRight and obtain \(-1\)
    - \( t \) then has to call again until an element is available
    - \( \Delta \) \( t \) is busy waiting and produces contention on the lock

  Idea: create a condition variable on which to block while waiting:

  ```c
  struct monitor {
    int tid; int count; int cond;
  }
  ```

Condition Variables

- Monitors simplify the construction of thread-safe resources.
- Still: Efficiency problem when using resource to synchronize:
  - if a thread \( t \) waits for a data structure to be filled:
    - \( t \) will call e.g. PopRight and obtain \(-1\)
    - \( t \) then has to call again until an element is available
    - \( \Delta \) \( t \) is busy waiting and produces contention on the lock

  Idea: create a condition variable on which to block while waiting:

  ```c
  struct monitor {
    int tid; int count; int cond;
  }
  ```

Define these two functions:

1. `wait` for the condition to become true
   - called while being inside the monitor
   - temporarily releases the monitor and blocks
   - when `signalled`, re-acquires the monitor and returns

2. `signal` waiting threads that they may be able to proceed
   - one/all waiting threads that called `wait` will be woken up, two possibilities:
     - the `signalling` thread suspends and continues once the `signalled` thread has released the monitor
     - the `signalling` thread continues, any `signalled` thread enters when the monitor becomes available
Signal-And-Urgent-Wait Semantics

Requires one queues for each condition $c$ and a suspended queue $s$:

- a thread who tries to enter a monitor is added to queue $e$ if the monitor is occupied
- a call to `wait` on condition $a$ adds thread to the queue $a.q$
- a call to `signal` for $a$ adds thread to queue $s$ (suspended)
- one thread form the $a$ queue is woken up
- `signal` on $a$ is a no-op if $a.q$ is empty
- if a thread leaves, it wakes up one thread waiting on $s$
- if $s$ is empty, it wakes up one thread from $e$

Signal-And-Continue Semantics

Here, the `signal` function is usually called `notify`.

- a call to `wait` on condition $a$ adds thread to the queue $a.q$
- a call to `notify` for $a$ adds one thread from $a.q$ to $e$ (unless $a.q$ is empty)
- if a thread leaves, it wakes up one thread waiting on $e$

Implementing Condition Variables

We implement the simpler `signal-and-continue` semantics:

```c
void cond_wait(mon_t *m) {
    assert(m->tid==thread_id());
    int old_count = m->count;
    m->tid = 0;
    wait(m->cond);
    bool next_to_enter;
    do {
        atomic {
            next_to_enter = m->tid==0;
            if (next_to_enter) {
                m->tid = thread_id();
                m->count = old_count;
            }
        } while (!next_to_enter);
    }
}

void cond_notify(mon_t *m) {
    signal(m->cond);
}
```

- a notified thread is simply woken up and competes for the monitor
A Note on Notify

With `signal-and-continue` semantics, two notify functions exist:

1. `notify`: wakes up exactly one thread waiting on condition variable
2. `notifyAll`: wakes up all threads waiting on a condition variable

⚠️ an implementation often becomes easier if `notify` means `notify some`

~~ programmer should assume that thread is not the only one woken up

What about the priority of notified threads?

- a notified thread is likely to block immediately on `qm->tid`
- notified threads compete for the monitor with other threads
- if OS implements FIFO order: notified threads will run *after* threads that tried to enter since `wait` was called
- giving priority to waiting threads requires more `complex implementation`

Implementing PopRight with Monitors

We use the monitor `q->m` and the condition variable `q->c`. PopRight:

```c
double-ended queue: removal

int PopRight(DQueue* q, int val) {
    QNode* oldRightNode;
    monitor_enter(q->m); // wait to enter the critical section
    L: QNode* rightSentinel = q->right;
    oldRightNode = rightSentinel->left;
    if (oldRightNode == leftSentinel) {
        cond_wait(q->c); goto L; } // signal that we're done
    QNode* newRightNode = oldRightNode->left;
    newRightNode->right = rightSentinel;
    rightSentinel->left = newRightNode;
    monitor_leave(q->m);
    int val = oldRightNode->val;
    free(oldRightNode);
    return val;
}
```
Monitor versus Semaphores

A monitor can be implemented using semaphores:
- protect each queue with a mutex
- use a binary semaphore to block threads that are waiting

Monitors with a Single Condition Variable

Monitors with a single condition variable are built into Java and C#:

```java
class C {
    public synchronized void f() {
        // body of f
    }
}
```

is equivalent to

```c
class C {
    public void f()
    {
        monitor_enter();
        // body of f
        monitor_leave();
    }
}
```

with Object containing:
- private int mon_var;
- private int mon_count;
- private int cond_var;
- protected void monitor_enter();
- protected void monitor_leave();

Deadlocks with Monitors

Definition (Deadlock)

A deadlock is a situation in which two processes are waiting for the respective other to finish, and thus neither ever does.

(The definition generalizes to a set of actions with a cyclic dependency.)