Practical multicore programming

Getting out of the DRF fragment

Avoiding locks, why?

Run faster.
- Run less code for synchronisation primitives (cf. suspension code).
- One very annoying problem.
  - If a thread is scheduled out inside a critical section, then all other threads that attempt entering the critical section are blocked.
- Or, priority inversion:
  - If a low-priority thread is inside a critical section, then all other threads are waiting for it to release the lock, including higher priority ones.

In other words some delay by some thread (in critical section) impacts all threads.

Non-blocking algorithms (primitives) avoid this problem.

Avoiding locks, a simple example

Modern computers provides atomic instructions, such as atomic addition.

One single instructions atomically performs:
- Read memory location $x$,
- then compute the sum $x + v$,
- then store the sum into location $x$.

Atomicity: No other thread can store into $x$ between steps 1. and 3.

To code in a portable manner, gcc provides "atomic_builtins" (C11 does provide similar primitives)

```c
//_sync_fetch_and_add
int __sync_fetch_and_add(type *p, type v)
{ type tmp = *p ; *p = tmp + v ; return tmp ; }
```

```c
//_sync_add_and_fetch
int __sync_add_and_fetch(type *p, type v)
{ type tmp = *p + v ; *p = tmp ; return tmp ; }
```

Notice that the value in location *p before or after the operation is returned.

Part 1.

A few non-blocking primitives:
atomic instructions
C11 atomic operations

For instance:

```c
#include <stdatomic.h>

static atomic_int x;

static int fetch_add() { return atomic_fetch_add(&x, 1); }
static int add_fetch() { assert 0; } // Does not exists

Notice: The C11 primitives also exist as “explicit” versions that takes an extra-argument related to memory model. The default model being SC. Then, for instance, for weaker guarantees:

```c
#include <stdatomic.h>

static atomic_int x;

static int fetch_add() { return atomic_fetch_add_explicit(&x, 1, memory_order_relaxed); }
```

Examples

<table>
<thead>
<tr>
<th>Blocking</th>
<th>Non-blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock_mutex(mutex);</td>
<td>__sync_fetch_and_add(&amp;sum, 1);</td>
</tr>
<tr>
<td>sum++;</td>
<td>unlock_mutex(mutex);</td>
</tr>
</tbody>
</table>

Non-blocking will run faster: (1) it runs (much) less code; (2) failure to proceed occurs less often. Demo in tst/:

```c
% safe ./sum.out -j2 -u 10000000
10244551 ...
% safe ./sum.out -j2 -l 10000000
20000000
status: 0
real: 2.44
user: 2.81
sys: 2.04
% safe ./sum.out -j2 -f 10000000
20000000
status: 0
real: 0.64
user: 1.27
sys: 0.00
```

Performance on a 12 cores, ×2 machine

![Graph showing performance comparison between UNLOCKED, LOCKED, and FETCH_AND_ADD]

Only LOCKED and FETCH_AND_ADD are correct, the second being faster.

What instructions (x86)?

Intel processors provide “locked” instructions, that execute a read and a write atomically (ie, no concurrent write to sum)

```c
#tmp <- *sum, tmp <- tmp+1, *sum <- tmp, all atomic.
lock addl $1, sum
```

The addl instruction is addition.

If we need the result (e.g. `int r = atomic_fetch_and_add(&sum, 1)`)

```c
#rdi holds the address of sum
movl $1, %eax # eax <- 1
#tmp <- eax, eax <- (rdi), (rdi) <- eax+tmp, all atomic.
lock xaddl %eax, (%rdi)
#eax holds the old value of (rdi)
```

The xaddl is “exchange and add” it performs the complex exchange operation depicted above. There exists a simple (locked by default) exchange instruction xchg1.
What instructions (Power)?

To execute a read and a write atomically, Power provides “load reserve” / “store conditional” instructions:

```assembly
#r3 holds the address of sum
sync ??
.L2:
lwarx 10,0,3 #r10 <- (r3) reserve r3
addi 9,10,1 #r9 <- r10+1
stwcx. 9,0,3 #(r3) <- r9, if reservation still valid
bne- 0,.L2 #jump if reservation invalidated
isync ??

#r10 holds the old value of (r3)
```

The instruction pair “load reserve” / “store conditional” performs a read/write atomic operation (i.e. no other thread can write to the reserved location in-between).

What instructions, (Power, C11)

```assembly
atomic_fetch_add_explicit(&x,1,mo);
```

Where `mo` is memory ordering.

```assembly
mo = memory_order_relaxed
```

```assembly
sync
```

```assembly
.L2:
lwarx 10,0,3
addi 9,10,1
stwcx. 9,0,3
bne- 0,.L2
```

```assembly
isync
```

The memory ordering specification provides some memory model guarantees. In final code, those correspond to adding explicit “fence” instructions, such as `sync` or `isync`.

Also observe: (1) the gcc old “atomic builtns” apparently equate to C11 sequentially consistent atomics; (2) for X86, generated code is as before.

A new primitive: the (synchronisation) barrier

- **init(b,n)**: Initialize the barrier for `n` participant.
- **wait(b)**: Wait on the barrier:
  - If `n-1` threads are waiting free everybody,
  - Otherwise wait (poll or suspend)

**Typical usage**: Provide synchronisation points for all threads. Consider two tasks `T1` and `T2` which must be performed sequentially (`T1(); T2();`). Assume easy subdivision of `T1` between `n` threads: `T1(0), T1(1), ..., T1(n-1)`

```assembly
// Perform the share of tasks 1 allocated to id
T1_share(id) ;
wait_barrier(b) ;
// Task 1 is now completed
if (id = 0) T2() ;
```

Used internally by frameworks that parallelize loops (e.g. openMP).

POSIX threads barriers

Our “basic” wrappers:

```c
pthread_barrier_t *alloc_barrier(unsigned count) {
    pthread_barrier_t *p = malloc_check(sizeof(*p)) ;
    int err ;
    if ((err = pthread_barrier_init(p,NULL,count)))
        errexit("pthread_barrier_init",err) ;
    return p ;
}
```

```c
void free_barrier(pthread_barrier_t *p) { ... }
```

```c
int wait_barrier(pthread_barrier_t *p) {
    int ret = pthread_barrier_wait(p) ;
    /* One thread will return PTHREAD_BARRIER_SERIAL_THREAD
    Others return 0.
    Other values signal errors. */
    if (ret != 0 && ret != PTHREAD_BARRIER_SERIAL_THREAD)
        errexit("pthread_barrier_wait",ret) ;
    return ret ;
}
```
Exercise I, demo

Consider \( n \) threads executing:

```c
void *runner(void *p) {
    ctx_t *p = p; common_t *q = p->common;
    for (int k = 0; k < q->m; k++) {
        wait_barrier(q->b);
        printf("<%i">,p->id); fflush(stdout);
        (void)wait_barrier(q->b);
        if (p->id == 0) printf("\n") ;
    }
    return NULL ;
}
```

What will this code do with for instance \( n = 4, m = 2 \)?

```
% tst/tst_my_barrier.out 4 2
<1><0><2><3>
<0><2><1><3>
```

Notice: \( T_1 \) is `printf("<\%i"",p->id)`. \( T_2 \) is `printf("\n")`.

---

Good practice

A context argument to threads, with:

- A thread-specific part.
- A part common to all threads.

```c
typedef struct {
    int nprocs, sz ; // Problem parameters
    pthread_barrier_t *b ; // Barrier
    double *r, *t ; // Arrays
    double result ; // Result will go here
} common_t ;

typedef struct {
    int id ; // Thread specific: identifier
    common_t *common;
} ctx_t ;
```

---

A (almost) serious \( T_1 \): compute a vector norm

Sequential version:

```c
double norm(double *t, int sz) {
    double r = 0.0 ;
    for (int k = 0; k < sz; k++) {
        double x = t[k] ;
        r += x * x ;
    }
    return r ;
}
```

Partition the vector into \( nprocs \) chunks \([kmin... kmax[) :

```c
double norm_partial(double *t, int kmin, int kmax) {
    double r = 0.0 ;
    for (int k = kmin ; k < kmax ; k++) {
        double x = t[k] ;
        r += x * x ;
    }
    return r ;
}
```

Computing norm: concurrent thread

```c
void *f(void *p) {
    ctx_t *p = p; common_t *q = p->common;
    // Task 1
    int chunk = q->sz / q->nprocs ;
    int kmin = p->id * chunk ;
    int kmax = kmin+chunk ;
    if (p->id == q->nprocs-1) kmax = q->sz ;
    q->r[p->id] = norm_partial(q->t,kmin,kmax) ;
    (void)wait_barrier(q->b);
    // Task 2
    if (p->id == 0) {
        q->result = 0.0 ;
        for (int k = 0; k < q->nprocs ; k++) q->result += q->r[k] ;
    }
    return NULL ;
}
```
Computing norm, thread control

```c
void run(int sz, int np) {
    // Initialisation
    pthread_barrier_t *b = alloc_barrier(np) ;
    double *t = malloc_check(sizeof(t[0])*sz) ;
    double *r = malloc_check(sizeof(r[0])*np) ;
    for (int k = 0 ; k < sz ; k++) t[k] = k ;
    common_t c ;
    c.nprocs = np, c.sz = sz ;
    c.b = b ; c.t = t ; c.r = r ;

    // Launch threads
    pthread_t th[np] ; ctx_t a[np] ;
    for (int k = 0 ; k < np ; k++) {
        ctx_t *p = &a[k] ; p->id = k ; p->common = &c ;
        create_thread(&th[k],f,p) ;
    }

    // Join
    for (int k = 0 ; k < np ; k++) join_thread(&th[k]) ;
    printf("%g\n",c.result) ;

    // Clean up
    free(r) ; free(t) ; free_barrier(b) ;
}
```

A note...

Why not use the simpler partition of array t, with stride np:

```c
double norm_partial(double *t,int id,int np,int sz) {
    double r = 0.0 ;
    for (int k = id ; k < sz ; k += np) {
        double x = t[k] ; r += x * x ;
    }
    return r ;
}
```

Answer: This solution would exhibit worse caching behaviour (array t).

Similarly, one could even think of avoiding "false sharing in the r array:

```c
// Array of size K*np
// K = sizeof(cache line)/sizeof(double)
double *r = malloc_check(sizeof(r[0])*np)*np ;

// Use one element per cache line
q->r[K*p->id] = norm_partial(q->t,p->id,q->np,q->sz) ;
```

Useless here: Few accesses.

Computing norm, demo

Actual code tst/norm.c, enriched with (wall-clock) timings (in sec.):

```c
# 2 cores x 2 ways HT
%pl ./norm.out 400000000 2
Parallel (nprocs=2) 2.13333e+25 [t=0.293575]  
Sequential 2.13333e+25 [t=0.453232] 
%pl ./norm.out 400000000 4
Parallel (nprocs=4) 2.13333e+25 [t=0.295624]  
Sequential 2.13333e+25 [t=0.449971]
```

On a (10-years old) 8-core Power 7 machine:

```c
./norm.out 400000000 8
Parallel (nprocs=8) 2.13333e+25 [dt=0.268532]
Sequential 2.13333e+25 [dt=1.2167]
```

Exercise I

Write your own barrier with one locks and one condition variable.

```c
typedef struct {
    unsigned int np ;
    volatile unsigned int count ;
    pthread_mutex_t *mutex ;
    pthread_cond_t *cond ;
} barrier_t ;

barrier_t *alloc_my_barrier(unsigned int np) {
    barrier_t *p = malloc_check(sizeof(*p)) ;
    p->np = p->count = np ;
    p->mutex = alloc_mutex() ;
    p->cond = alloc_cond() ;
    return p ;
}

void free_my_barrier(barrier_t *p) { ... }
```
Exercise I, write wait

Here is a simple solution:

```c
int wait_barrier(barrier_t *p) {
    int ret;
    lock_mutex(p->mutex);
    p->count--;
    if (p->count > 0) {
        wait_cond(p->cond,p->mutex); ret = 0;
    } else {
        broadcast_cond(p->cond);
        p->count = p->nprocs; ret = PTHREAD_BARRIER_SERIAL_THREAD;
    }
    unlock_mutex(p->mutex);
    return ret;
}
```

Rather straightforward:

- The first nprocs-1 threads suspend on condition.
- The last thread awake them.

Informally, the barrier is correct even if used several times. But incorrect if suspended threads wake up spuriously. Complete solution later!

**Why not test?**

```c
T1(void *p) {                      T2(void *p) {
    ctx_t *p = _p;                 ctx_t *p = _p;
    common_t *q = p->common;       common_t *q = p->common;
    q->x = 1;                      q->y = 1;
    int r1 = q->y;                 int r2 = q->x;
    return alloc_boxed_int(r1);    return alloc_boxed_int(r2);
}                                  }
```

```c
void run(void) {
    for ( ; ; ) {
        // Prepare arguments, launch threads
        boxed_int_t *p1 = join(&th1), *p2 = join(&th2);
        int r1 = p1->val, r2 = p2->val;
        free_boxed_int(p1); free_boxed_int(p2);
        if (r1 == 0 && r2 == 0) return; // Found!
    }
}
```

It will not work: too expensive, too little simultaneous execution.

**A real-world application for barriers?**

Testing memory models! Consider the litmus test SB:

```
<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x ← 1</td>
<td>y ← 1</td>
</tr>
<tr>
<td>r1 ← y</td>
<td>r2 ← x</td>
</tr>
</tbody>
</table>
```

Non-sequentially consistent executions:

```
r1 = 0, r2 = 0
```

**A better design**

Amortise thread creation, using a loop; synchronise loop iterations with a barrier:

```c
void *P1(void *p) {
    ctx_t *p = _p;         common_t *q = p->common;
    int *t = calloc_check(size, sizeof(t[0]));
    for (int k = q->size-1; k >= 0; k--) {
        wait_barrier(q->b);
        q->x[k] = 1;
        int r1 = q->y[k];
        t[k] = r1;
    }
    return t;
}
```

Also observe that memory locations x, y now are array cells x[k], y[k].
Testing SB, thread control

The usual stuff: prepare arguments, launch threads, join:

```c
void run(int size) {
  barrier_t *b = alloc__barrier(2) ;
  for (; ;) {
    int *x = calloc(size,sizeof(x[0])) ;
    int *y = calloc(size,sizeof(y[0])) ;
    common_t c ;
    c.size = size ; c.b = b ; c.x = x ; c.y = y ;
    pthread_t th1,th2 ;
    ctx_t a1,a2 ;
    a1.id = 1 ; a1.common = &c ; create_thread(&th1,P1,&a1) ;
    a2.id = 2 ; a2.common = &c ; create_thread(&th2,P2,&a2) ;
    int *r1 = join_thread(&th1) ; int *r2 = join_thread(&th2) ;
    ...
  }
}
```

Testing SB, result analysis

```c
int count[2][2] ; // indexed by r1,r2
for (int k = 0 ; k < size ; k++) {
  count[r1[k]][r2[k]]++ ;
}
```

```c
define count[0][0] = count[0][1] = count[1][0] = count[1][1] = 0 ;
```

```c
int nz = 0 ;
for (int k1 = 0 ; k1 < 2 ; k1++)
  for (int k2 = 0 ; k2 < 2 ; k2++) {
    int c = count[k1][k2] ;
    if (c > 0) nz++ ;
    printf("%-5i> r1=%i r2=%i\n" ,c,k1,k2) ;
  }
```

```c
free(x) ; free(y) ; free(r1) ; free(r2) ;
if (nz == 4) break ; // All four results found.
```

```c
free_barrier(b) ;
```

Testing SB, demo

Run tst/sb.out, first with small size parameter:

```bash
% ./sb.out 1000
**********
0 > r1=0 r2=0
689 > r1=0 r2=1
311 > r1=1 r2=0
0 > r1=1 r2=1
...
```

No simultaneous execution (i.e. r1=1, r2=1).
Can have some for size=100000:

```bash
% ./sb.out 100000
**********
0 > r1=0 r2=0
63841> r1=0 r2=1
36156> r1=1 r2=0
3 > r1=1 r2=1
...
```

Quite slow (a lot of system time due to suspensions).

Let us write our fast barrier, first try

A very simple structure, a counter.

```c
typedef struct {
  unsigned int nprocs ;
  volatile unsigned int count ;
} barrier_t ;
```

```c
barrier_t alloc_barrier(int nprocs) {
  barrier_t *p = malloc_check(sizeof(*p)) ;
  p->nprocs = p->count = nprocs ;
  return p ;
}
```

```c
free_barrier(barrier_t *p { free(p) ; }
```
A first try, wait (exercise II)

// We use this atomic builtin
#define DEC(x) __sync_add_and_fetch(x,-1)

void wait_barrier(barrier_t *p) {
    (void)DEC(&p->count);
    while (p->count > 0);
}

Problem: Can be used once only (demo: tst/tst_bad_barrier.out).
Even worse, the first invocation may fail:
1. Thread A decrements count.
2. Thread B decrements count.
3. Thread A sees a null count, then performs wait again, and...
   Thread A decrements count (Note: type is unsigned).
4. Thread B pools for ever, since count holds 0xffffffff.

Third try, reinitialising the barrier

Second try works once. (demo: tst/tst_bad2_barrier 2 1).
Let us attempt to reinitialise the barrier:

void wait_barrier(barrier_t *p) {
    int rem = DEC(&p->count);
    if (rem == 0) {
        p->count = p->nprocs; // Reinitialisation
        p->go = 1;
    } else {
        while (p->go == 0);
    }
}

But where can "go" be re-initialised?

A simple solution is not to reinitialise "go", but to alternate between two
version of wait that swap 0 and 1.

A barrier with two steps

int wait0(barrier_t *p) {
    int rem = DEC(&p->count);
    if (rem == 0) {
        p->count = p->nprocs
        p->go = 1;
        return PTHREAD...
    } else {
        while (p->go == 0);
        return 0;
    }
}

int wait1(barrier_t *p) {
    int rem = DEC(&p->count);
    if (rem == 0) {
        p->count = p->nprocs
        p->go = 0;
        return PTHREAD...
    } else {
        while (p->go == 1);
        return 0;
    }
}

int wait(barrier_t *p) { (void)wait0(p); return wait1(p); }

Second try, let us focus on the first synchronisation

We add a field, p->go, to poll on:

typedef struct {
...
    volatile int go ;
} barrier_t ;

barrier_t *alloc_my_barrier(unsigned int nprocs) {
    ...
    p->go = 0;
    ...
    return p;
}

Idea: The last thread to enter the barrier is responsible for: releasing the
other threads (exercise III).

void wait_barrier(barrier_t *p) {
    int rem = DEC(&p->count);
    if (rem == 0) p->go = 1; // Last thread
    else while (!p->go); // Other threads
}

Third try, reinitialising the barrier

Second try works once. (demo: tst/tst_bad2Barrier 2 1).
Let us attempt to reinitialise the barrier:

void wait_barrier(barrier_t *p) {
    int rem = DEC(&p->count);
    if (rem == 0) {
        p->count = p->nprocs;
        p->go = 1;
    } else {
        while (p->go == 0);
    }
}

But where can "go" be re-initialised?

A simple solution is not to reinitialise "go", but to alternate between two
version of wait that swap 0 and 1.

A barrier with two steps

int wait0(barrier_t *p) {
    int rem = DEC(&p->count);
    if (rem == 0) {
        p->count = p->nprocs
        p->go = 1;
        return PTHREAD...
    } else {
        while (p->go == 0);
        return 0;
    }
}

int wait1(barrier_t *p) {
    int rem = DEC(&p->count);
    if (rem == 0) {
        p->count = p->nprocs
        p->go = 0;
        return PTHREAD...
    } else {
        while (p->go == 1);
        return 0;
    }
}

int wait(barrier_t *p) { (void)wait0(p); return wait1(p); }
Fourth try, sense reversing barrier

Alternating the two versions of \texttt{wait} is not convenient. A better solution
is to pass a thread-specific "\texttt{sense}" argument.

\begin{verbatim}
void wait_barrier(barrier_t *p, int *sensep) {
    int sense = *sensep ;
    int rem = DECR(&p->count) ;
    if (rem == 0) {
        p->count = p->nprocs ;
        p->go = !sense ;
    } else {
        while (p->go == sense) ;
    }
    *sensep = !sense ;
}
\end{verbatim}

Using the sense reversing barrier

\textbf{Notice:} Thread specific means that each thread has his own memory cell
to hold the sense.

\begin{verbatim}
void *runner(void *p) {
    ctx_t *p = _p ; common_t *q = p->common ;
    int sense = 0 ; // My own sense variable, notice initialisation.
    for (int k = 0 ; k < q->m ; k++) {
        wait_barrier(q->b,&sense) ;
        printf("%i",p->id) ;
        wait_barrier(q->b,&sense) ;
        if (p->id == 0) printf("\n") ;
    }
    return NULL ;
}
\end{verbatim}

Alternative:
Replace the \texttt{sensep} argument by thread identifier.

- \texttt{create_barrier} allocates an array \texttt{sense} of size \texttt{nprocs}.
- \texttt{wait_barrier} takes \texttt{id} as argument and \texttt{sensep} is \&sens[e][i].

Alternative:
Suppress the additional argument with thread-specific data, painful (cf. \texttt{tst/tst_sense2_barrier.c}).

Getting rid the “sense” argument

The sense reversing barrier comes from Herlihy and Shavit's textbook
“The Art of Multiprocessor Programming” (and thus we trust this code).
It is tempting to simplify its code as follows:

\begin{verbatim}
void wait_barrier(barrier_t *p) {
    int sense = p->go ;
    int rem = DECR(&p->count) ;
    if (rem == 0) {
        p->count = p->nprocs ;
        p->go = !sense ;
    } else {
        while (p->go == sense) ;
    }
}
\end{verbatim}

We can check the correctness of the new barrier using a verification tool
such as \texttt{cubicle}.

Voir “Vérification de programmes C concurrents avec Cubicle : Enfoncer
les barrières”, Sylvain Conchon, Luc Maranget, Alain Mebsout, David
Declerck. JFLA’14.

BTW: A DRF barrier that handles spurious wakeups

\begin{verbatim}
void wait_barrier(barrier_t *p) {
    lock_mutex(p->mutex) ;
    int sense = p->sense ;
    --p->count ;
    if (p->count > 0) {
        /* Not last */
        do {
            wait_cond(p->cond,p->mutex) ;
        } while (p->sense == sense) ;
    } else {
        /* I'am last */
        p->count = p->nprocs ; /* Re-triggers barrier */
        p->sense = !sense ; /* Free waiting threads */
        broadcast_cond(p->cond) ; /* Free waiting threads */
    }
    unlock_mutex(p->mutex) ;
}
\end{verbatim}

\textbf{Notice:} A new field : p->sense.
Back to testing SB

```c
void *T1(void *p) {
    ... for (int k = size-1 ; k >= 0 ; k--) {
        wait(q->b);
        q->x[k] = 1;
        int r1 = q->y[k];
        t[k] = r1;
    }
    ...
}
```

```c
void *T2(void *p) {
    ... for (int k = size-1 ; k >= 0 ; k--) {
        wait(q->b);
        q->y[k] = 1;
        int r2 = q->x[k];
        t[k] = r2;
    }
    ...
}
```

**Demo:** Try `tst/mysb.out`

```
% ./mysb.out 10
...
**********
4 > r1=0 r2=0
4 > r1=0 r2=1
1 > r1=1 r2=0
1 > r1=1 r2=1
```

Hence, much better synchronisation.

---

**Weak memory models?**

For the testing application, there are no races anyway, except for the ones we test! (and the ones on barrier subcomponents).

**But:** Is the barrier itself correct? We assume the barrier to be correct in SC (I took it from Herlihy and Shavit’s book). How can we rule out non-SC executions?

**Memory barriers (fences):** We assume: inserting the “stronger fence” between every two pairs of shared memory accesses (with different locations) by the same processor rules out non-SC execution.

**Being portable:** GCC documentation:

```c
__sync_synchronize (...)  
This builtin issues a full memory barrier.
```

**Really inserting fences**

The gcc compiler on my old machine emitted nothing for `__sync_synchronize` (as could be checked with gcc -S).

Let us be sure about this, using the asm construct:

```c
void *T1(void *p) {
    ... for (int k = size-1 ; k >= 0 ; k--) {
        wait(q->b,&sense);
        q->x[k] = 1;
        asm __volatile__ ("mfence:::memory") ;
        int r1 = q->y[k];
        t[k] = r1;
    }
    ...
}
```

```c
void *T2(void *p) {
    ... for (int k = size-1 ; k >= 0 ; k--) {
        wait(q->b,&sense);
        q->y[k] = 1;
        asm __volatile__ ("mfence:::memory") ;
        int r2 = q->x[k];
        t[k] = r2;
    }
    ...
}
```

**Demo:** Edit `tst/sb.c` and test again.

```
% ./mysb.out 1000000
1 > r1=0 r2=0
500489 > r1=0 r2=1
499175 > r1=1 r2=0
335 > r1=1 r2=1
```

It did not work on my old computer! (Works now.)

Then `sbfence.out` may run forever.

---

**Really inserting fences**

Then `sbfence.out` may run forever.
The rules of fencing

To rule-out all non-SC behaviours:
- Execute full fence between every two pairs of shared accesses to different locations.

For x86,
- Atomic operations have fence semantics.
- Only write to read pairs need to be fenced.

To optimise fence insertion, i.e. using less fences and weaker fences, more information on the memory model or application is required.

C11 barrier code, playing it safe

```c
typedef struct {
    unsigned int nprocs;
    atomic_uint count;
    atomic_int sense;
} barrier_t;
...
#define MO memory_order_seq_cst
void wait_barrier(barrier_t *p) {
    int sense = atomic_load_explicit(&p->sense,MO);
    int rem = atomic_fetch_add_explicit(&p->count,-1,MO);
    if (rem == 0) {
        atomic_store_explicit(&p->count,p->nprocs,MO);
        atomic_store_explicit(&p->sense,1-sense,MO);
    } else {
        while (atomic_load_explicit(&p->sense,MO) == sense);
    }
}
```

Using SC as the memory model looks safe...

Fencing barrier code playing it safe

```c
inline static void SYN_C(void) { __sync_synchronize(); }

void wait_barrier(barrier_t *p) {
    SYN_C();
    int sense = p->go;
    SYN_C();
    int rem = DECR(&p->count);
    if (rem == 0) {
        SYN_C();
        p->count = p->nprocs;
        SYN_C();
        p->go = !sense;
    } else {
        SYN_C();
        while (p->go == sense);
    }
    SYN_C();
}
```

Inserting fences everywhere, playing it really safe, as we ignore the possible fences around DECR. However notice, no fence in while loop, because accesses are to the same location.

Part 1-a.

Write your own spin-lock
Test and Set spin-lock (TAS)

The get_and_set(type *p, type v) atomic primitive:

- Load value w of *p.
- Store value v into *p.

How do we write a simple spin-lock?

inline static void lock(volatile int *p) {
    while (get_and_set(p, 1) != 0) ;
    import() ;
}

inline static void unlock(volatile int *p) {
    export() ;
    *p = 0 ;
}

Notice: Import and export barrier are fences sufficient for memory accesses to be performed inside critical section.

Primitives, x86

Get-and-set is implemented with the atomic exchange instruction:

```c
inline static int get_and_set(volatile int *p, int v) {
    asm __volatile__ ("xchgl \%[r], (%[m])"
        : [r] "=r" (v)
        : [m] "r" (p), "[r]" (v)
        : "memory") ;
    return v ;
}

inline static void import( void ) {
    asm __volatile__ ("isync" :: : "memory") ;
}

inline static void export( void ) {
    asm __volatile__ ("lwsync" :: : "memory") ;
}
```

In case of contention...

As usual intensive usage of arbitrating primitives degrades the performance of the whole memory system.

Write a new version of lock that avoids the atomic get_and_set when the lock is not free — a TTAS lock (test-and-test-and-set).

```c
inline static void lock(volatile int *p) {
    while (get_and_set(p, 1) != 0) {
        while (*p != 0) ;
    }
    import() ;
}

inline static void unlock(volatile int *p) {
    export() ;
    *p = 0 ;
}
```

This lock is called TTAS (test-and-test-and-set).

Primitives, Power

Get-and-set is implemented with a load-reserve/store conditional pair.

```c
inline static int get_and_set(volatile int *p, int v) {
    int r2 ;
    asm __volatile__ ("0:\n\t" "lwarx %[r2], 0,%[m]\n\t" "cmpwi %[r2], 0\n\t" "bne_if\n\t" "stwcx. %[r1], 0,%[m]\n\t" "bne_0b\n\t" : [r2] "=r" (r2)
        : [m] "p" (p), [r1] "p" (v)
        : "memory") ;
    return r2 ;
}

inline static void import( void ) {
    asm __volatile__ ("isync" :: : "memory") ;
}

inline static void export( void ) {
    asm __volatile__ ("lwsync" :: : "memory") ;
}
```
Performance evaluation

We run \( n\text{procs} \) threads that execute a loop of size \( \frac{sz}{n\text{procs}} \).

```c
void *P1(void *p) {
    ctx_t *q = _p;
    common_t *q = p->common;
    int sz = q->sz/q->nprocs;

    while (!q->start); // Simple synchro with siblings.

    for (int k = sz ; k > 0 ; k--) {
        lock(&q->spin);
        // Waste time ...  
        q->x++;
        unlock(&q->spin);
    }
    return(NULL);
}
```

Hence for all \( n\text{procs} \) the amount of work performed is roughly the same, up to contention!

Machine beaune, pentium Xeon, 8 cores

![Graph showing performance comparison for different architectures]

Modern Xeon, 12 cores, 2-ways hypterthreading

![Graph showing performance comparison for different architectures]
Modern Xeon, 40 cores, 2-ways hyperthreading

Part 2.

A non-blocking data structure: Treiber’s stack

Non-blocking data structure

Target:

- Pass items between threads, as the fifo of last class.
- But no thread can delay others.
- Which prevents us from using locks.

The easiest to code such structure is a list-based stack.

Warm up

Let us code a locked list-based stack, i.e. nothing more than an ordinary stack with push and pop protected by a mutex.

```c
/* List cells */
typedef struct stack_node_t {
    void *val ;
    struct stack_node_t *next ;
} stack_node_t ;

stack_node_t *alloc_stack_node(void *val) ; // next field <- NULL
void free_stack_node(stack_node_t *p) ;

/* Stack proper */
typedef struct {
    stack_node_t *top ;
    pthread_mutex_t *mutex ;
} stack_t ;

stack_t *alloc_stack(void) ;
void free_stack(stack_t *p) ;

Exercise IV: write push and pop.
```
Concurrent stack, push

```c
void push(stack_t *p, void *v) {
    stack_node_t *q = alloc_stack_node(v);  // Fresh node
    lock_mutex(p->mutex);
    stack_node_t *old = p->top;           // Old top node
    q->next = old;
    p->top = q;                           // Do push
    unlock_mutex(p->mutex);
}
```

Look again

The mutex serves one important purpose:

```c
void push(stack_t *p, void *v) {
    ... // q is the node to insert in front.
    stack_node_t *old = p->top;      // Old top node
    q->next = old;
    assert(p->top == old);           // Old top node
    p->top = q;                      // Do push
    ...
}
```

If we get rid of mutex how can we be sure that p->top still holds old when writing q to p->top?

**Answer:** There is a primitive (machine instruction) to help us: atomic compare and swap.

Concurrent stack, pop

If we pop and empty stack, return the distinguished value **NULL**.

```c
void *pop(stack_t *p) {
    lock_mutex(p->mutex);
    stack_node_t *old = p->top; // Old top node.
    if (old == NULL) {
        unlock_mutex(p->mutex);
        return NULL;
    } else {
        p->top = old->next;       // Do pop
        unlock_mutex(p->mutex);
        void *r = old->val;       // Clean up and return
        free_stack_node(old);
        return r;
    }
}
```

(Atomic) compare and swap (CAS)

The construct **CAS**(type *p, type old, type new), where type is some basic type (integer, pointer...),

- Compare *p and the value old, then:
  - If equal, store new into *p.
  - If not equal, do nothing.
- Return result of comparison (or value read, it depends).

All those steps are performed atomically, in particular no other thread can write to *p between read and write.

**gcc builtins:** There are two depending on whether the comparison (a boolean, a civilised int) or the value read is returned.

- bool __sync_bool_compare_and_swap (type *p,...)
- type __sync_val_compare_and_swap (type *p,...)

We shall use:

```c
#define CAS(p,old,new) __sync_bool_compare_and_swap (p,old,new)
```
Lock-free push

We shall thus insert the push attempt into a loop, retrying until update succeeds.

```c
void push(stack_t *p, void *v) {
    stack_node_t *q = alloc_stack_node(v), *old;
    do {
        old = p->top;
        q->next = old;
    } while (!CAS(&p->top, old, q)); // Do push
}
```

Fencing left as an exercise.

Lock-free pop

Again, we attempt updating p->top until success.

```c
void *pop(stack_t *p) {
    for (; ; ) {
        stack_node_t *old = p->top;
        // Old top node.
        if (old == NULL) {
            return NULL;
        } else {
            stack_node_t *next = old->next;
            if (CAS(&p->top, old, next)) { // Do pop
                void *r = old->val;
                // Clean up and return
                free_stack_node(old);
                return r;
            }
        }
    }
}
```

Fencing left as an exercise

Oh well, is this lock-free?

```
% man malloc
...
```

To avoid corruption in multithreaded applications, mutexes are used internally to protect the memory-management data structures employed by these functions. In a multithreaded application in which threads simultaneously allocate and free memory, there could be contention for these mutexes. To scalably handle memory allocation in multithreaded applications, glibc creates additional memory allocation arenas if mutex contention is detected. Each arena is a large region of memory that is internally allocated by the system (using brk(2) or mmap(2)), and managed with its own mutexes.

Looks ok: mutexes are still present, but at least contentions is claimed to be avoided.

Implementing CAS on x86

Let us see __sync_bool_compare_and_swap(&x,100,200).

Intel processors feature a specific (lock) compare and exchange instruction.

```
movl $100, %eax  # eax <- 100
movl $200, %edx  # ebx <- 200
lock cmpxchgl %edx, (%ecx) # CAS((ecx),eax,edx)
#eax contains the value read.
#Flag ZF records the result of comparing eax and (ecx)
```

Notice that cmpxchgl implicitly uses the register eax.
Implementing CAS on Power

Let us see \_\_sync\_bool\_compare\_and\_swap(&x, 100, 200).

```plaintext
li 0,200    # r0 <- 200
sync        # full fence
.L3:
lwarx 3,0,9 # r3 <- (r9), reserve r9
cmpwi 0,3,100 # compare r3 and 100
bne- 0,.L4  # if <> jump
stwcx. 0,0,9 # (r9) <- r0 (ie 200)
bne- 0,.L3  # jump if reservation invalidated
isync      # not a full fence, enough?
.L4:
#Compute the boolean value of r3 == 100...
xori 3,3,100
cntlzw 3,3
srwi 3,3,5
addi 1,1,32
```

Contestion

If the stack is accessed repeatedly and concurrently by many threads, performance will degrade severely. Due to repeated, system-wide synchronisation induced by CAS.

Pthread mutex solves the issue by suspending. For instance, threads that fail to acquire a lock suspend (after a while).

For us, a simple solution: suspend for a while, ideally:

- The initial delay is random, so as to delay contending threads differently (cf. ethernet retry delay).
- Delay increases as contention does.

Example of “exponential backoff” with random initial delay

One estimates contention by counting failed CAS attempts. i.e. delay is \( d_0 \times 2^{\text{fail}-1} \).

```plaintext
void push(stack_t *p, void *v) {
    int d = -1;
    stack_node_t *q = alloc_stack_node(v);
    for (; ; ) {
        stack_node_t *old = p->top;
        q->next = old;
        if (CAS(&p->top,old,q)) return;
        // CAS failed, sleep for some time d
        if (d < 0) delay = rand(); // Hum, rand() locked?
        else if (delay < MAX_DELAY) d *= 2;
        delay(d);
    }
}
```

Delay is bounded...

The ABA problem

Schematic pop:

```plaintext
void *pop(stack_t *p) {
    ...
    stack_node_t *old = p->top;  // L1
    stack_node_t *next = old->next;
    if (CAS(&p->top,old,next)) ... // L2
    ...
}
```

We assume that when CAS succeeds (because \( p->\text{top} \) value is the same at L2 and L1), then no other thread has touched the stack between L2 and L1. Is it true?

No: because nodes are recycled. Let be a stack that holds A and B.

- Thread 1 attempt a pop, but is delayed just before the CAS.
- Thread 2 pops A and B freeing the nodes \( n_A \) and \( n_B \).
- Then thread 2 pushed a value C, reusing \( n_A \).
- Then 1 starts again returning A instead of C and leaving an inconsistent stack (as \( n_B \) is re-introduced).
Also notice that ABA does not disturb push!

Schematic push:

```
stack_node_t *q = alloc_stack_node(v) ;
stack_node_t *old = p->top ; // L1
q->next = old ;
if (CAS(&p->top,old,q)) ... // L2
```

Even if the node pointed to by `old` returns in front of list after being recycled, there is no problem!

As `old` still points to a valid (current) stack, even if not the same as in L1.

### Solving ABA

A garbage collector solves ABA: \( n_A \) cannot be freed and recycled as long as a thread holds a pointer to it, which thread 1 precisely does.

Using load reserve/store conditional also solves the issue, because if the store conditional succeeds we can be sure that no thread has written to \( p->top \):

```
stack_node_t *old = LR(p->top) ; // L1
stack_node_t *next = old->next ;
if (SC(&p->top,next)) ... // L2
```

This solution is non-portable and there are restrictions (no other LR between LR and SC).

Another solution is for \( p->top \) to hold both a pointer to the front node and a counter, which is incremented whenever \( p->top \) changes.

### Tagging pointers

Assume a `tagged_t` type, with the following operations:

- `tagged_t pack(stack_node_t *p, unsigned tag)` build tagged pointer.
- `stack_node_t *getptr(tagged_t p)` extract pointer.
- `unsigned gettag(tagged_t p)` extract tag.

And of course we declare:

```
typedef struct {
  tagged_t top ;
} stack_t ;
```

It remains to rewrite `pop` and `push`, managing the tags.

### Pop with tagged pointers

```
void *pop(stack_t *p) {
  for ( ; ; ) {
    tagged_t old = p->top ; // Old top node.
    stack_node_t *realold = getptr(old) ;
    unsigned tag = gettag(old) ;
    if (realold == NULL) {
      return NULL ;
    } else {
      stack_node_t *realnext = realold->next ;
      tagged_t next = pack(realnext,tag+1) ;
      if (CAS(&p->top,old,next)) { // Do pop
        void *r = realold->val ; // Clean up and return
        free_stack_node(realold) ;
        return r ;
      }
    }
  }
}
```
Push with tagged pointers

```c
void push(stack_t *p, void *v) {
    stack_node_t *q = alloc_stack_node(v);
    tagged_t old, q_packed;
    do {
        old = p->top;
        stack_node_t *realold = getptr(old);
        unsigned tag = gettag(old);
        q->next = realold;
        q_packed = pack(q, tag);
    } while (!CAS(&p->top, old, q_packed)); // Do push
}
```

Observe that the tag is not changed: ABA scenarios involve push and pop, it suffices to change the tag for either one or the other.

Implementation of tagged pointers

To alleviate the effect of wraparound, we need tag to occupy as many bits as possible.

But wait, we need CAS on tagged pointers. This severely limits their size.

In effect, we use machine level atomic compare and exchange instructions, which typically operate on the natural words (32 or 64 bits), which happen to be of the same size as pointers.

Some ideas:
- Steal bits from pointers, which do not use all of them.
  - We have some knowledge of maximal pointer value (rare), and we use higher order bits.
  - We have 32-bits pointers and a 64 bits compare and swap (rare).
- Use a multi-word non-blocking CAS, see relevant and recent literature.

Another solution: hazard pointers


Some pointers are registered as “hazardous”. In our case, stack nodes pointers as soon as pop holds a copy.

Freeing of stack nodes is delayed, until they are no longer hazardous

Schematic pop (with extra id argument)
```c
void *pop(stack_t *p, int id) {
    stack_node_t *old = p->top;
    p->hazard[id] = old; // Register old as ‘hazardous’
    if (p->top != old) continue; // And re-validate it
    stack_node_t *next = old->next;
    if (CAS(&p->top, old, next)) {
        ...
        // Replaces free_stack_node(old)
        retire_stack_node(p, id, old);
        ...
    }
}
```

More on “Free non-hazardous stack nodes”

The function retire_stack_node(p, id, old) will:
- Register the retired node in a thread-local data structure retired[id].
- If size of retired[id] is lower than $T$, exit.
- Otherwise, collect all current hazard pointers (all p->hazard[id]) in $H$ (size at most $N$).
- Then, scan retired[id], for each element, lookup in $H$
  - If in $H$, retain.
  - Otherwise, do free.

Aiming at amortized constant cost for $N$ threads

Cost is: $\Omega(N) \times I + \Omega(T) \times L$, where $I$ and $L$ are insertion/lookup costs. $I$ and $L$ can be constant (!) (hashtable) or log($N$) (sorted array).

If $T$ is $N + \Omega(N)$ (e.g. $N^2$, $2N$), scanning is performed at a rate of every $\Omega(N)$ pops. And we reach constant (or log($N$)) amortized cost.
Demo(s)

./run_...outN P: have P "poppers" and 2P "pushers". A pusher pushes the integers from 1 to \( N \), while poppers compete to pop the stack.

% ./run_treiber.out 1000 100 #no ABA provision
*** glibc detected *** ./run_treiber.out: double free or corruption (out): 0x00007f567802b4f0
Segmentation fault (core dumped)
% ./run_treiber_hazard.out
% ./run_treiber_hazard.out
% ./run_treiber_hazard.out
% ./run_treiber_hazard.out
...

BTW: Identifying "hazards" may be non-obvious for data structures more sophisticated that a stack.

Multiword CAS

By the mean of one indirection + CAS on pointer.

typedef struct { int data[N]; } multi_t ;
...

int multi_cas(muti_t **p, multi_t *old, multi_t *new) {
    return CAS(p,old,new) ;
}

But beware of ABA! \( \rightarrow \) allocation through hazard pointers.

void muti_inc(multi_t **p, int id) {
    for (; ; ) {
        multi_t *old = *p ;
        hazard[id] = old ;
        if (*p != old) continue ;
        multi_t *new = … // Allocate old + 1
        int r = multi_cas(p,old,new) ;
        if (r) { retire(old,id) ; return ; }
    }
}

Fine grain locking

A simple idea: partition data structure and use a mutex per subpart.

Examples
- Lists: one mutex per cell.
- Hash-tables, one lock per group of buckets.
- …

Coding is often far from being obvious.

We consider a simple example: a FIFO with two locks.

Part 3.

Fine grain locking.
The sequential FIFO again

typedef enum {OK,NO} ret_val; // Return value for put below

int put(fifo_t *f, subtask_t *z) {
  if (f->nitems == f->sz) return NO;
  f->t[f->lst] = z;
  f->lst++; f->lst %= f->sz; f->nitems++;
  return OK;
}

subtask_t *get(fifo_t *f) {
  subtask_t *r;
  if (f->nitems == 0) return NULL; // special value
  r = f->t[f->fst];
  f->fst++; f->fst %= f->sz; f->nitems--;
}

We observe that put and get have their own indices fst and lst to array t.

So why not use one lock for put and one lock for get.

Two-locks FIFO, put

As put and get both access nitems, we use atomic increment.

// Atomic *p++
#define POSTINC(p) __sync_fetch_and_add(p,1)

void put(fifo_t *f, void *z) {
  lock_mutex(f->lst_mutex);
  while (f->nitems == f->sz) {
    wait_cond(f->is_full,f->lst_mutex);
  }
  f->t[f->lst] = z;
  f->lst++; f->lst %= f->sz;
  int was_empty = POSTINC(&f->nitems) == 0;
  unlock_mutex(f->lst_mutex);
  if (was_empty) {
    lock_mutex(f->fst_mutex);
    broadcast_cond(f->is_empty);
    unlock_mutex(f->fst_mutex);
  }
}

Two-locks FIFO, get

Symmetric, decrementing nitems.

// Atomic *p--
#define POSTDEC(p) __sync_fetch_and_add(p,-1)

void *get(fifo_t *f) {
  void *r;
  lock_mutex(f->fst_mutex);
  while (f->nitems == 0) {
    wait_cond(f->is_empty,f->fst_mutex);
  }
  r = f->t[f->fst];
  f->fst++; f->fst %= f->sz;
  int was_full = POSTDEC(&f->nitems) == f->sz;
  unlock_mutex(f->fst_mutex);
  if (was_full) {
    lock_mutex(f->lst_mutex);
    broadcast_cond(f->is_full);
    unlock_mutex(f->lst_mutex);
  }
  return r;
}

Weak memory model?

There are data races, on nitems and t[..].
Schematically

Inserting fences, playing it safe, for x86:

```
SYNC       SYNC
R nitems   R nitems
SYNC       SYNC
W t[1st]   R t[fst]
R/W nitems R/W nitems
SYNC       SYNC
```

The initial fences are for external data races. (useless?).

No fence before atomic R/W, as we know there is already one (for x86, and Power, having checked assembler code). No fence after W t[1st], as the following atomic R/W acts as a fence.

Some references

On synchronisation barrier:


On hazard pointers: