Practical multicore programming

Getting out of the DRF fragment

Luc Maranget Luc.Maranget@inria.fr

Avoiding locks, why?

Run faster.

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.

Part 1.

A few non-blocking primitives: atomic instructions

Avoiding locks, a simple example

Modern computers provides atomic instructions, such as atomic addition.

One single instructions atomically performs:

1. Read memory location x,
2. then compute the sum $x + v$,
3. 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)

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

type __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.

C11 atomic operations

For instance:

```
#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:

```
#include <stdatomic.h>

static atomic_int x ;

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

Examples

Blocking

```
lock_mutex(mutex) ;
sum++ ;
unlock_mutex(mutex) ;
```

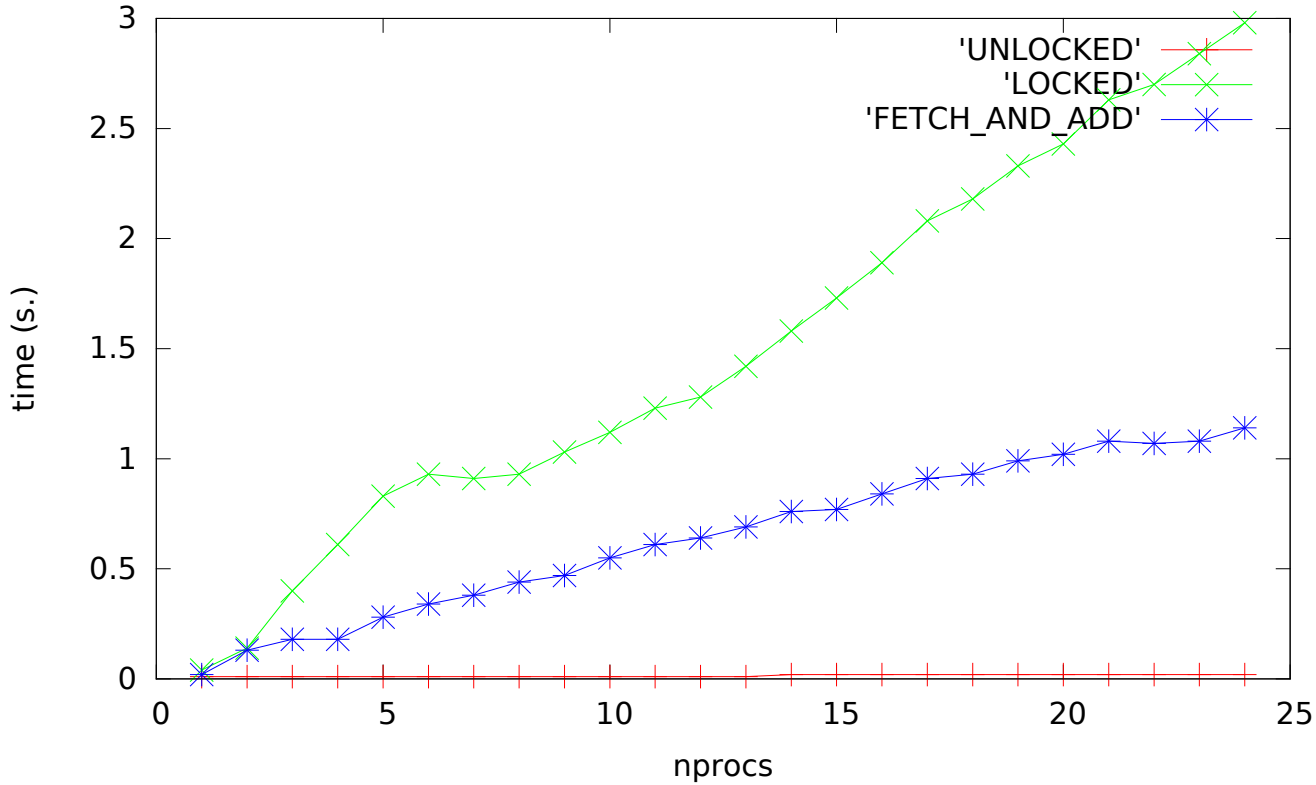
Non-blocking

```
__sync_fetch_and_add(&sum,1) ;
```

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

```
% 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, $\times 2$ machine



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)"

```
#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)`)

```
#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 `xchgl`.

What instructions (Power)?

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

```
#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)

atomic_fetch_add_explicit(&x,1,*mo*);

Where *mo* is memory ordering.

mo = **memory_order_relaxed**

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

mo = `memory_seq_cst`

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

What instructions, (Power, C11)

atomic_fetch_add_explicit(&x,1,*mo*);

Where *mo* is memory ordering.

mo = **memory_order_relaxed**

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

mo = memory_seq_cst

```
        sync
.L2:
        lwarx 10,0,3
        addi 9,10,1
        stwcx. 9,0,3
        bne- 0,.L2
        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 builtins" apparently equate to C11 sequentially consistent atomics;

What instructions, (Power, C11)

atomic_fetch_add_explicit(&x,1,*mo*);

Where *mo* is memory ordering.

mo = **memory_order_relaxed**

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

mo = memory_seq_cst

```
        sync
.L2:
        lwarx 10,0,3
        addi 9,10,1
        stwcx. 9,0,3
        bne- 0,.L2
        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 builtins" 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 T_1 and T_2 which must be performed sequentially (`T1() ; T()`). Assume easy subdivision of T_1 between n threads: $T_1(0), T_1(1), \ldots, T_1(n-1)$

```
// 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:

```
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 ;
}

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

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:

```
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 $n = 4$, $m = 2$?

Exercise I, demo

Consider n threads executing:

```
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")`.

A (almost) serious T_1: compute a vector norm

Sequential version:

```
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[`:

```
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 ;
}
```

Good practice

A context argument to threads, with:

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

```
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 ;
```

Computing norm: concurrent thread

```
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

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

  // Launch threads
  pthread_t th[nprocs] ; ctx_t a[nprocs] ;
  for (int k = 0 ; k < nprocs ; k++) {
    ctx_t *p = &a[k] ; p->id = k ;  p->common = &c ;
    create_thread(&th[k],f,p) ;
  }
  // Join
  for (int k = 0 ; k < nprocs ; k++)join_thread(&th[k]) ;
  printf("%g\n",c.result) ;
  // Clean up
  free(r) ; free(t) ; free_barrier(b) ;
}
```

Computing norm, demo

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

```
# 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:

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

A note. . .

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

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

Answer:

A note...

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

```
double norm_partial(double *t,int id,int nprocs,int sz) {
  double r = 0.0 ;
  for (int k = id ; k < sz ; k += nprocs) {
    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:

```
// Array of size K*procs
// K = sizeof(cache line)/sizeof(double)
double *r = malloc_check(sizeof(r[0])*nprocs % K) ;
...
// Use one element per cache line
q->r[K*p->id] = norm_partial(q->t,p->id,q->nprocs,q->sz) ;
...
```

Useless here:

A note...

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

```
double norm_partial(double *t,int id,int nprocs,int sz) {
  double r = 0.0 ;
  for (int k = id ; k < sz ; k += nprocs) {
    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:

```
// Array of size K*procs
// K = sizeof(cache line)/sizeof(double)
double *r = malloc_check(sizeof(r[0])*nprocs % K) ;
...
// Use one element per cache line
q->r[K*p->id] = norm_partial(q->t,p->id,q->nprocs,q->sz) ;
...
```

Useless here: Few accesses.

Exercise I

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

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

barrier_t *alloc_my_barrier(unsigned int nprocs) {
  barrier_t *p = malloc_check(sizeof(*p)) ;
  p->nprocs = p->count = nprocs ;
  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:

```
int wait_barrier(barrier_t *p) {
```

Exercise I, write `wait`

Here is a simple solution:

```
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.

Exercise I, write `wait`

Here is a simple solution:

```
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!

A real-world application for barriers?

Testing memory models! Consider the litmus test SB:

T_1	T_2
$x \leftarrow 1$	$y \leftarrow 1$
$r_1 \leftarrow y$	$r_2 \leftarrow x$

$$r_1 = 0 \wedge r_2 = 0?$$

Sequentially consistent executions:

$$r_1 = 0, r_2 = 1 \quad r_1 = 1, r_2 = 0 \quad r_1 = 1, r_2 = 1$$

Non-sequentially consistent executions:

$$r_1 = 0, r_2 = 0$$

Why not test?

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

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

```
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 better design

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

```
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:

```
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

```
    :
    int count[2][2] ; // indexed by r1,r2
    count[0][0] = count[0][1] = count[1][0] = count[1][1] = 0 ;
    for (int k = 0 ; k < size ; k++) {
      count[r1[k]][r2[k]]++ ;
    }
    printf("**********\n") ;
    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) ;
      }
    free(x) ; free(y) ; free(r1) ; free(r2) ;
    if (nz == 4) break ; // All four results found.
  }
  free_barrier(b) ;
}
```

Testing SB, demo

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

```
% ./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:

```
% ./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.

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

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

}

free_barrier(barrier_t *p { free(p) ; }
```

A first try, `wait` (exercise II)

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

void wait_barrier(barrier_t *p) {
```

A first try, `wait` (exercise II)

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

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

Problem:

A first try, `wait` (exercise II)

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

void wait_barrier(barrier_t *p) {
  (void)DECR(&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* see a null `count`, then performs `wait` again, and...

A first try, `wait` (exercise II)

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

void wait_barrier(barrier_t *p) {
  (void)DECR(&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* see 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`.

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) {
```

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 = DECR(&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_bad2_barrier 2 1`).
Let us attempt to reinitialise the barrier:

```
void wait_barrier(barrier_t *p) {
  int rem = DECR(&p->count) ;
  if (rem == 0) {

    p->go = 1 ;
  } else {
    while (p->go == 0) ;
  }
}
```

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 = DECR(&p->count) ;
  if (rem == 0) {
    p->count = p->nprocs ; // Reinitialisation
    p->go = 1 ;
  } else {
    while (p->go == 0) ;
  }
}
```

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 = DECR(&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?

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 = DECR(&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 = DECR(&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 = DECR(&p->count) ;
  if (rem == 0) {
    p->count = p->nprocs
    p->go = 0 ;
    return PTHREAD_... ;
  } else {
    while (p->go == 1) ;
    return 0 ;
  }
}
```

A barrier with two steps

```
int wait0(barrier_t *p) {
  int rem = DECR(&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 = DECR(&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 `wait` is not convenient. A better solution is to pass a thread-specific "`sense`" argument.

```
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 ;
}
```

Using the sense reversing barrier

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

```
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 ;
}
```

Using the sense reversing barrier

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

```
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 ;
}
```

Alternative: Replace the `sensep` argument by thread identifier.

- `create_barrier` allocates an array `sense` of size `nprocs`.
- `wait_barrier` takes `id` as argument and `sensep` is `&sense[id]`.

Alternative: Suppress the additional argument with tread-specific data, painful (cf. `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).

```
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 ;
}
```

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:

```
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) ;
  }
}
```

We can check the correctness of the new barrier using a verification tool such as `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

```
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) ;
}
```

Notice: A new field : `p->sense`.

Back to testing SB

```
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 ;
  }
  ...
}
```

```
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?

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?

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:

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

So let us try!

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

```
void *T2(void *_p) {
  ...
  for (int k = size-1 ;
       k >= 0 ; k--) {
    wait(q->b) ;
    q->y[k] = 1 ;
    __sync_synchronize() ;
    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.)

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:

```
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 ;
  }
  ...
}
```

```
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 ;
  }
  ...
```

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.

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.

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.

Fencing barrier code

```
inline static void SYNC(void) { __sync_synchronize(); }

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) ;
  }

}
```

Fencing barrier code playing it safe

```
inline static void SYNC(void) { __sync_synchronize(); }

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

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.

Fencing barrier code for x86

```
inline static void SYNC(void) { __sync_synchronize(); }

void wait_barrier(barrier_t *p) {
  SYNC() ;
  int sense = p->go;

  int rem = DECR(&p->count) ;
  if (rem == 0) {

    p->count = p->nprocs ;

    p->go = !sense ; SYNC() ;
  } else {

    while (p->go == sense) ;
  }

}
```

As we need fences between writes and reads only. Also assuming that **DECR** have "fence semantics".

C11 barrier code, playing it safe

```
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 == 1) {
    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. . .

Part 1-a.

Write your own spin-lock

Test and Set spin-lock (TAS)

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

1. Load value *w* of `*p`.
2. 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:

```
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) { }

inline static void export(void) { }
```

Primitives, Power

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

```
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␣1f\n\t"
   "stwcx.␣%[r1],0,%[m]\n\t"
   "bne␣0b\n\t"
   "1:\n\t"
   : [r2] "=&r" (r2)
   : [m] "r" (p), [r1] "r" (v)
   : "memory") ;
  return r2 ;
}

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).

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).

```
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).

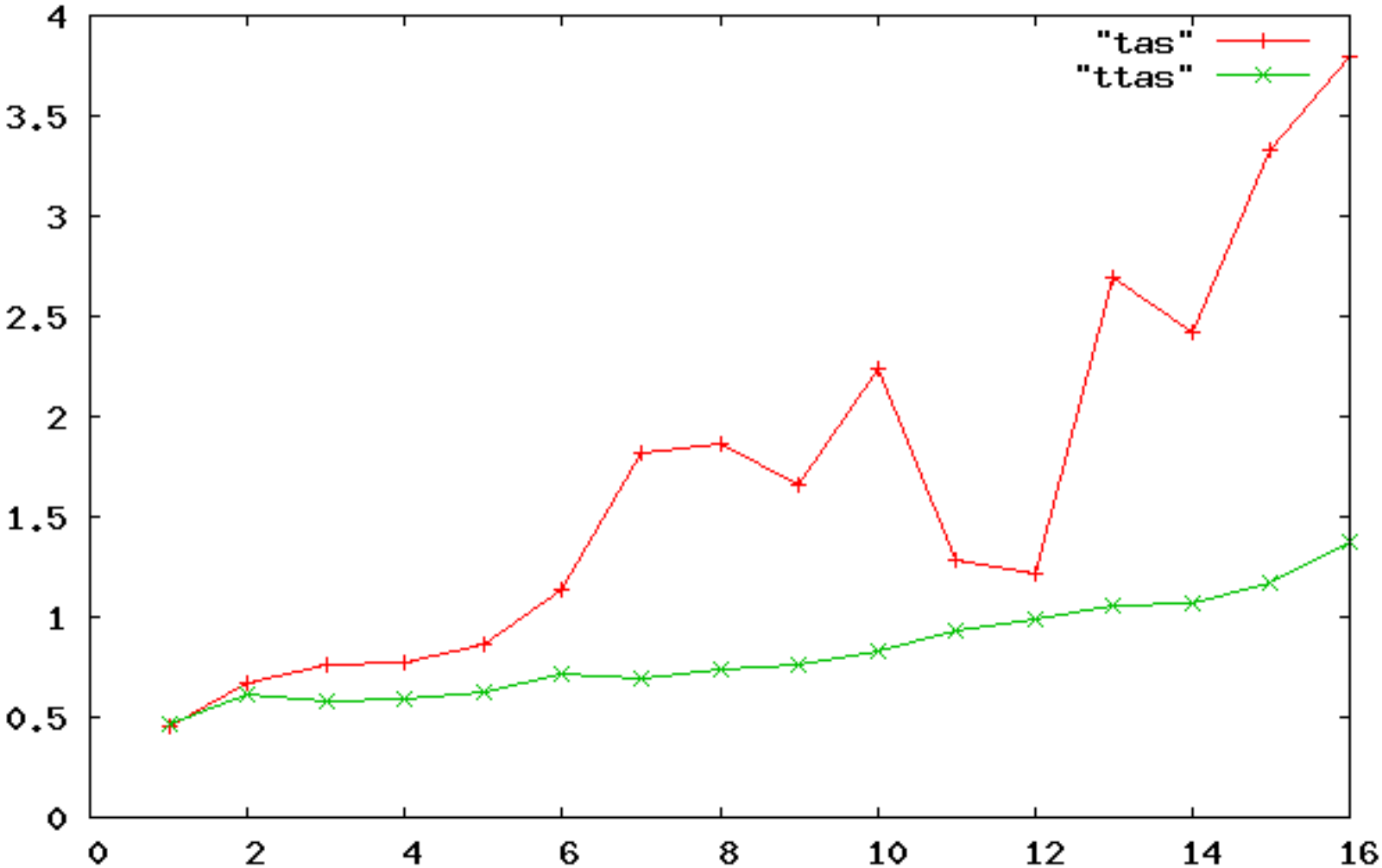
Performance evaluation

We run `nprocs` threads that execute a loop of size `sz/nprocs`.

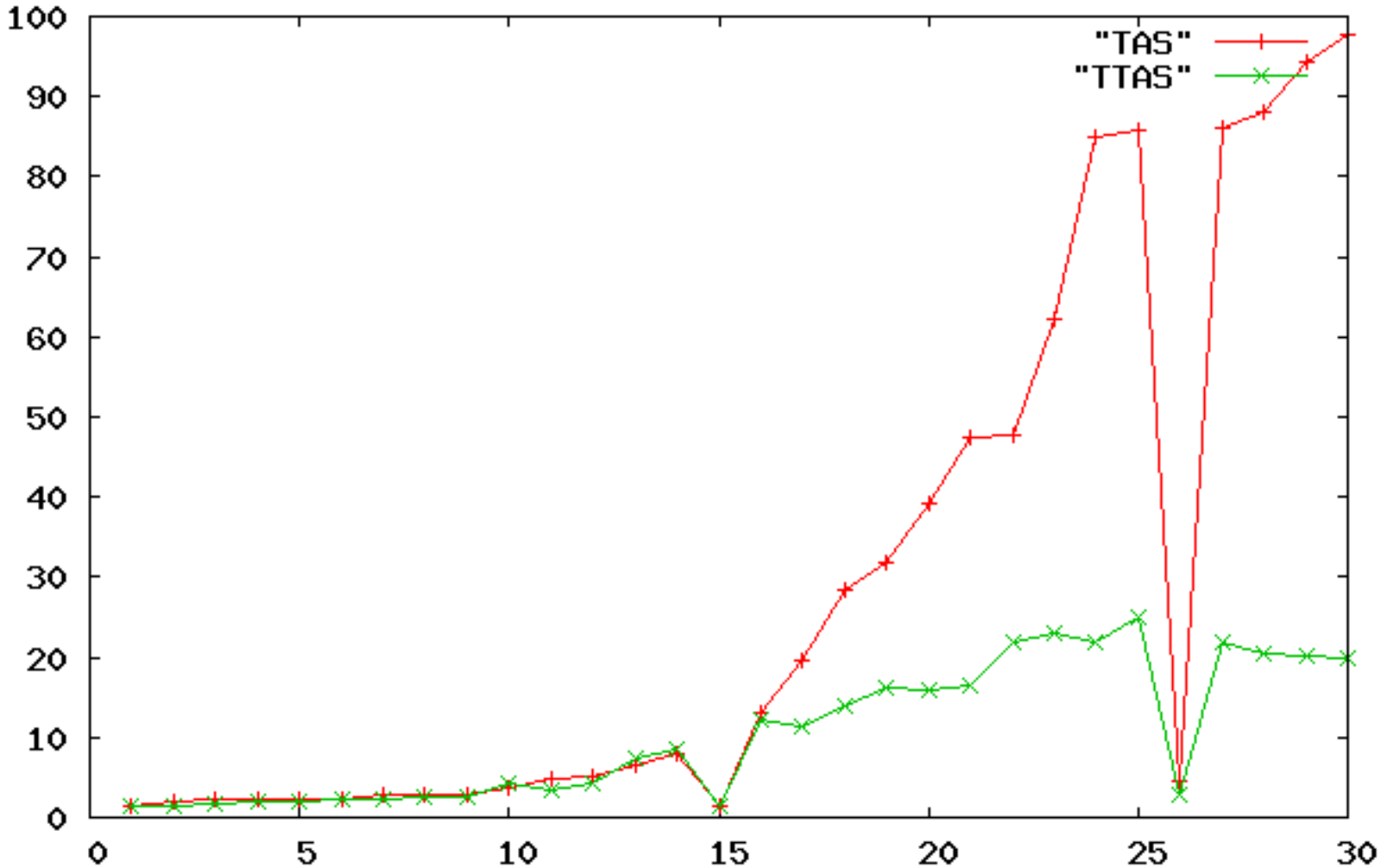
```
void *P1(void *_p) {
  ctx_t *p = _p ;
  common_t *q = p->common ;
  int sz = q->sz/q->nprocs ;

  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 `nprocs` the amount of work performed is roughly the same, up to contention!

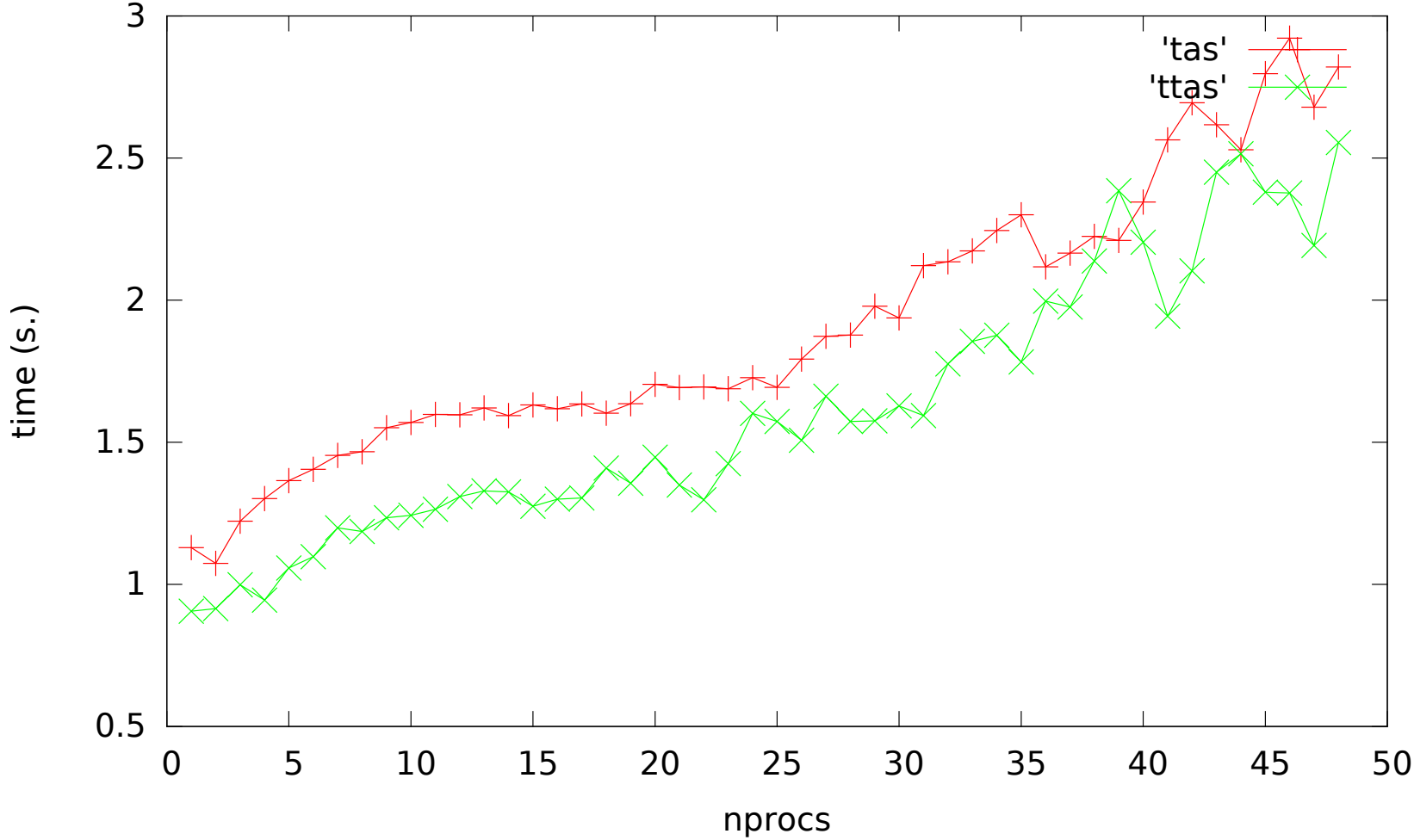
Machine beaune, pentium Xeon, 8 cores



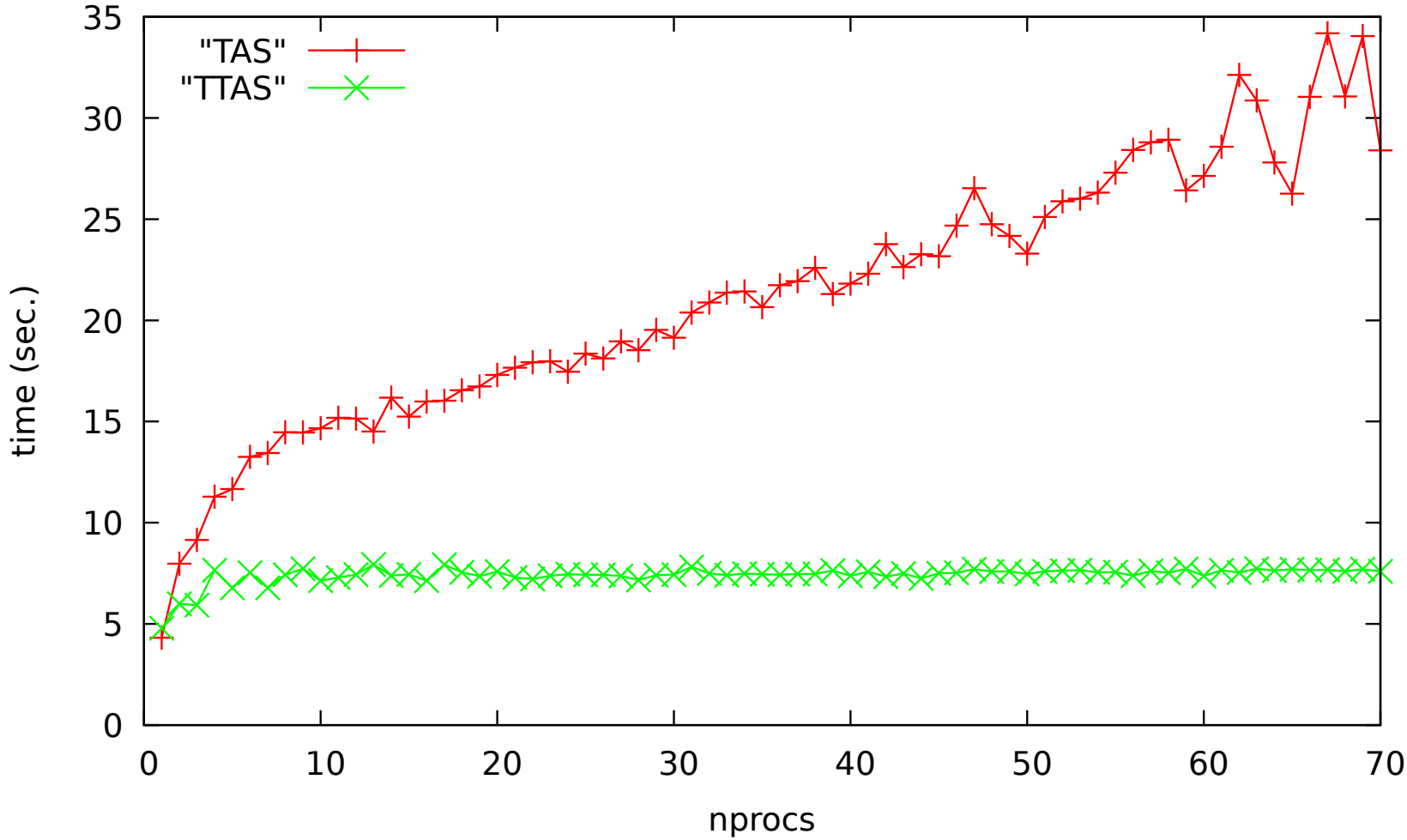
Machine power7, power 7, 8 cores, 4-ways SMT



Modern Xeon, 12 cores, 2-ways hypterthreading



Modern Xeon, 40 cores, 2-ways hypterthreading



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.

```
/* 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

```
void push(stack_t *p, void *v) {
```

Concurrent stack, push

```
void push(stack_t *p, void *v) {
  stack_node_t *q = alloc_stack_node(v) ;  // Fresh node
```

Concurrent stack, push

```
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) ;
}
```

Concurrent stack, pop

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

```
void *pop(stack_t *p) {
```

Concurrent stack, pop

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

```
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 ;
  }
}
```

Look again

The mutex serves one important purpose:

```
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) ;
  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:

Look again

The mutex serves one important purpose:

```
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) ;
  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.

(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:

```
#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.

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

Lock-free push

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

```
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.

```
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 ;
  }
}
```

Lock-free pop

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

```
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 func
        tions.  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 allo
        cation  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 resister `eax`.

Implementing CAS on Power

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

```
  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
```

Contention

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. Fo 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 $d_0 \times 2^{\text{nfail}-1}$.

```
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:

```
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->top` value is the same at L2 and L1), then no other thread has touched the stack between L1 and L2. Is it true?

The ABA problem

Schematic pop:

```
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->top` value is the same at L2 and L1), then no other thread has touched the stack between L1 and L2. 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 pushes 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:

```
typdef 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

```
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:

Push with tagged pointers

```
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

From: "Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects" Maged M. Michael, IEE Transactions on Parallel and Distributed Systems, Vol 15, No 6, June 2004. `http://researchweb.watson.ibm.com/people/m/michael/ieeetpds-2004.pdf`.

1. Some pointers are registered as "hazardous". In our case, stack nodes pointers as soon as pop holds a copy.
2. Freeing of stack nodes is delayed, until they are no longer hazardous

Schematic pop (with extra `id` argument)

```
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:

1. Register the retired node in a thread-local data structure `retired[id]`.

More on "Free non-hazardous stack nodes"

The function `retire_stack_node(p,id,old)` will:

1. Register the retired node in a thread-local data structure `retired[id]`.
2. If size of `retired[id]` is lower than T, exit.

More on "Free non-hazardous stack nodes"

The function `retire_stack_node(p,id,old)` will:

1. Register the retired node in a thread-local data structure `retired[id]`.
2. If size of `retired[id]` is lower than T, exit.
3. Otherwise, collect all current hazard pointers (all `p->hazard[id]`) in H (size at most N).

More on "Free non-hazardous stack nodes"

The function `retire_stack_node(p,id,old)` will:

1. Register the retired node in a thread-local data structure `retired[id]`.
2. If size of `retired[id]` is lower than T, exit.
3. Otherwise, collect all current hazard pointers (all `p->hazard[id]`) in H (size at most N).
4. Then, scan `retired[id]`, for each element, lookup in H
   - If in H, retain.
   - Otherwise, do free.

More on "Free non-hazardous stack nodes"

The function `retire_stack_node(p,id,old)` will:

1. Register the retired node in a thread-local data structure `retired[id]`.
2. If size of `retired[id]` is lower than T, exit.
3. Otherwise, collect all current hazard pointers (all `p->hazard[id]`) in H (size at most N).
4. 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).

More on "Free non-hazardous stack nodes"

The function `retire_stack_node(p,id,old)` will:

1. Register the retired node in a thread-local data structure `retired[id]`.
2. If size of `retired[id]` is lower than T, exit.
3. Otherwise, collect all current hazard pointers (all `p->hazard[id]`) in H (size at most N).
4. 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 ?, $2N$), scanning is performed at a rate of every $\Omega(N)$ pops. And we reach constant (or $\log(N)$) amortized cost.

Demo(s)

`./run_...out`N 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 corrupt
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 ; }
  }
}
```

Part 3.

Fine grain locking.

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.

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 POSTINCR(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 = POSTINCR(&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 POSTDECR(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 = POSTDECR(&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[..]`.

```
...
while
  (f->nitems == f->sz) {
...
f->t[f->lst] = z ;
...
POSTINCR(&f->nitems) == 0 ;
...
```

```
...
while
  (f->nitems == 0) {
...
r = f->t[f->lst] ;
...
POSTDECR(&f->nitems) == f->sz ;
...
```

Schematically

Inserting fences.

R `nitems`

W `t[lst]`

R/W `nitems`

R `nitems`

R `t[fst]`

R/W `nitems`

Schematically

Inserting fences, playing it safe:

SYNC	SYNC
R `nitems`	R `nitems`
SYNC	SYNC
W `t[lst]`	R `t[fst]`
R/W `nitems`	R/W `nitems`
SYNC	SYNC

The initial fences are for external data races.

No fence before atomic R/W, as we know there is already one (for x86, and Power, having checked assembler code).

Schematically

Inserting fences, for x86:

SYNC	SYNC
R `nitems`	R `nitems`
W `t[lst]`	R `t[fst]`
R/W `nitems`	R/W `nitems`

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

No fence after W `t[lst]`, as the following atomic R/W acts as a fence

Some references

On synchronisation barrier:

"The Art of Multiprocessor Programming" (Chap. 17) Maurice Herlihy and Nir Shavit. Morgan Kaufmann.

"Vérification de programmes C concurrents avec Cubicle : Enfoncer les barrières", Sylvain Conchon, Luc Maranget, Alain Mebsout et David Declerck. JFLA'14.

On hazard pointers:

"Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects", Maged M. Michael. IEEE transactions on parallel and distributed systems, vol. 15, no. 6, June 2004.
`http://researchweb.watson.ibm.com/people/m/michael/ieeetpds-2004.pdf`.