

# PARALLEL COMPUTING

## Shared Memory



Armin Biere  
2019/04/02



JOHANNES KEPLER  
UNIVERSITY LINZ

# Why Shared Memory?

- wide-spread availability of multi-core

- in servers for more than 20 years
- desktop for more than 10 years
- GPU computing for more than 10 years
- smart phones for more than 5 years

- power limits in CMOS technology

- around 2005 frequency scaling stopped
- Moore's law still continued to hold
- more cores instead of higher frequency

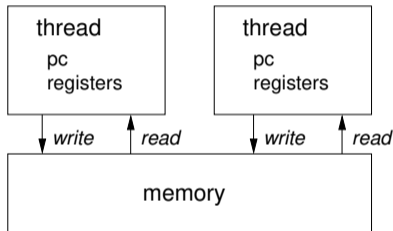
- threads

- "known" programming model
- similar to sequential model
- but with globally shared memory
- and multiple processing units

- processes

- classical but more complicated
- fork / join paradigm
- communication over files / pipes
- `mmap (... , MAP_SHARED, ...)`

# Shared Memory Programming Model



- programs / processes / threads
  - local architectural (CPU) state
  - including registers / program counter
  - shared heap for threads
  - shared memory for processes
- communicate over **global** memory
  - think globally shared variables
- *read* and *write* atomic
  - only for machine word values (and pointers)
  - need other synchronization mechanisms
- solution for mutual exclusion needed

## Data Race

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

pthread_t t0, t1;
int x;

void *
incx (void * dummy)
{
    x++;
    return 0;
}
```

```
int
main (void)
{
    pthread_create (&t0, 0, incx, 0);
    pthread_create (&t1, 0, incx, 0);
    pthread_join (t0, 0);
    pthread_join (t1, 0);
    printf ("%d\n", x);
    return 0;
}
```

## Data Race

- this code already gives some ideas about pthreads
- increment function `incx` just increments the global variable `x` (without locking)
- the `main` function creates two threads running `incx`
- then waits for them to finish (joins with first thread `t0` first, then with second `t1`)
- if first thread finishes executing `incx` before second starts then there is no problem
- incrementing twice should always yield 2 as output
- but there is a potential data race
  1. first thread `t0` reads value 0 of `x` into local register `r0`
  2. also increments its local copy in `r0` to value 1
  3. second thread `t1` reads old value 0 of `x` into its local register `r1`
  4. also increments its local copy in `r1` to value 1
  5. now first thread `t0` writes back `r0` to the global variable `x` with value 1
  6. finally second thread `t1` writes back `r0` to the global variable `x` with value 1
- testing with massif load (schedule steering)  
`valgrind --tool=helgrind` or `gcc -fsanitize=thread`

## Avoiding Data Races Through Locking / Mutual Exclusion

```
void *  
incx (void * dummy)  
{  
    lock ();  
    int tmp = x;  
    tmp++;  
    x = tmp;  
    unlock ();  
    return 0;  
}
```

How to implement locking?

- will first look at software only solutions
- hardware solutions much more efficient

## Eraser / Lock-Set Algorithm

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, Thomas E. Anderson:  
Eraser: A Dynamic Data Race Detector for Multithreaded Programs. ACM Trans. Comput. Syst. 15(4):  
391-411 (1997)

- check for “locking discipline”
  - shared access protected by at least one lock
  - collect lock sets at read and write events
  - check that intersection of lock sets non-empty
- if a lock-set becomes empty
  - produce improper locking warning (potential data race)
  - even though the actual race might not have occurred
- initialization is tricky (phases)
  - spurious warnings
  - only some can be suppressed automatically
- for instance implemented in `helgrind`
- major problem is that it needs “sandboxing” (interpreting memory accesses)

## Mutual Exclusion with Deadlock

```
#include ...

pthread_t t0, t1;
int x;

int id[] = { 0, 1 };
int flag[] = { 0, 0 };

void lock (int * p) {
    int me = *p,
        other = !me;
    flag[me] = 1;
    while (flag[other])
        ;
}

void unlock (int * p) {
    int me = *p;
    flag[me] = 0;
}

void *
incx (void * p)
{
    lock (p);
    x++;
    unlock (p);
    return 0;
}

int
main (void)
{
    pthread_create (&t0, 0, incx, &id[0]);
    pthread_create (&t1, 0, incx, &id[1]);
    pthread_join (t0, 0);
    pthread_join (t1, 0);
    printf ("%d\n", x);
    return 0;
}
```



# Deadlock

## ■ data race

- uncoordinated access to memory
- interleaved partial views
- inconsistent global state (incorrect)
- “always consistent” = **safety** property
- avoided by locking
- which in turn might slow-down application

## ■ deadlock

- two threads wait for each other
- each one needs the other to “release its lock” to move on
- “no deadlock” = **liveness** property
- does not necessarily need sandboxing
- might be easier to debug
- might actually not be that bad (“have you tried turning it off and on again?”)
- more fine-grained versions later

## ■ debugging dead-lock

- tools allow to find locking cycles
- run your own cycle checker after wrapping lock / unlock
- attach debugger to deadlocked program

# Mutual Exclusion with Deadlock

```
#include ...

pthread_t t0, t1;
int x;

int id[] = { 0, 1 };
int victim = 0;

void lock (int * p) {
    int me = *p;
    victim = me;
    while (victim == me)
        ;
}

void unlock (int * p) {
}
```

## ■ previous version

- flag to go first
- hope nobody else has the same idea at the same time
- but check that and if this is not the case proceed
- deadlock under contention

## ■ this version

- even more passive / helpful
- always let the other go first
- tell everybody that you are waiting
- wait until somebody else waits too
- almost always deadlocks (without contention)

## ■ the Peterson algorithm combines both ideas

# Peterson Algorithm

```
void lock (int * p) {
    int me = *p;
    int other = !me;
    flag[me] = 1;
    victim = me;
    // __sync_synchronize ();
    while (flag[other] && victim == me)
        ;
}

void unlock (int * p) {
    int me = *p;
    flag[me] = 0;
}
```

actually broken on real modern hardware

- without the memory fence
- because read in other thread can be reordered before own write (even for restricted x86 memory model)

expected:

0: <i>write</i> (flag[0], 1)	1: <i>write</i> (flag[1], 1)
0: <i>write</i> (victim, 0)	1: <i>write</i> (victim, 1)
0: <i>read</i> (flag[1]) = 1	1: <i>read</i> (flag[0]) = 1

possible:

0: <i>read</i> (flag[1]) = 0	1: <i>read</i> (flag[0]) = 0
0: <i>write</i> (flag[0], 1)	1: <i>write</i> (flag[1], 1)
0: <i>write</i> (victim, 0)	1: <i>write</i> (victim, 1)

# Mutual Exclusion Algorithms

- classical “software-only” algorithms
  - more of theoretical interest only now
  - because memory model of multi-core machines weak (reorders reads and writes)
  - but would be on reorder-free hardware still not really efficient (in space and time)
- need hardware support anyhow
  - various low-level (architecture) dependent primitives
  - atomic increment, bit-set, compare-and-swap and memory fences
  - better use platform-independent abstractions, such as pthreads
- we will later see how-those low-level primitives can be used

# Sequential Consistency

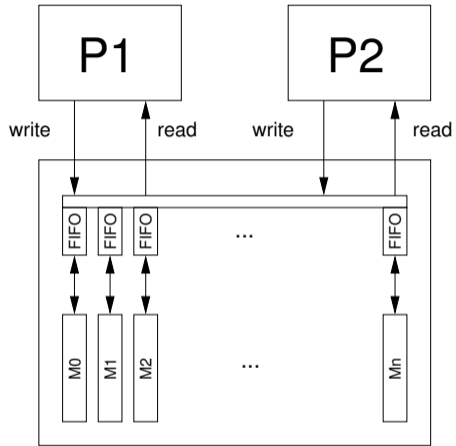
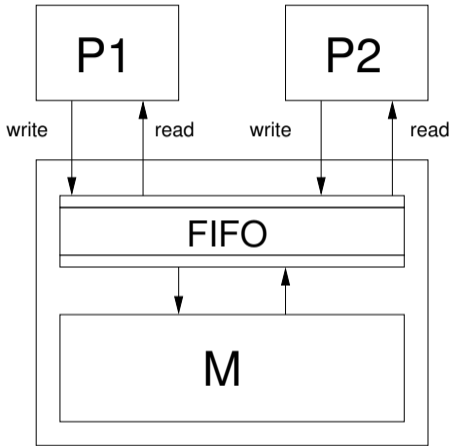
Leslie Lamport:

How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs.

IEEE Trans. Computers 28(9): 690-691 (1979)

- systems with processors (cores) and memories (caches)
  - think HW: processors and memories work in parallel
  - processors read (fetch) values and write (store) computed values to memories
  - common abstraction: consider each memory address as single memory module
- (single) processor **sequential** iff programs (reads / writes) executed sequentially
  - sequentially means without parallelism
  - between memories and the single processor
- processors **sequentially consistent** iff
  - every parallel execution of programs
  - can be reordered into a sequential execution
  - such that sequential semantics of programs and memories are met
  - sequential (single) program semantics: read / writes executed in program order
  - sequential (single) memory semantics: read returns what was written (array axioms in essence)

# FIFO Read / Write Order



global FIFO read / write operation gives sequential consistency (left)  
projected to individual memory addresses too (right)

## Out-of-Order Write-to-Read

```
long a, b;
```

```
void * f (void * q) {  
    a = 1;  
    long c = a;  
    long d = b;  
    long u = c + d;  
    return (void*) u;  
}
```

```
void * g (void * p) {  
    b = 1;  
    long e = b;  
    long f = a;  
    long v = e + f;  
    return (void*) v;  
}
```

```
pthread_t s, t;
```

```
int main () {  
    pthread_create (&s, 0, f, 0);  
    pthread_create (&t, 0, g, 0);  
    long u, v;  
    pthread_join (s, (void**) &u);  
    pthread_join (t, (void**) &v);  
    long m = u + v;  
    printf ("%ld\n", m);  
    return 0;  
}
```

## Out-of-Order Write-to-Read

```
long a, b;
```

```
long f () { a = 1; long tmp = a; return tmp + b; }
```

```
long g () { b = 1; long tmp = b; return tmp + a; }
```

```
void * f (void * q) {
```

```
    a = 1;                // fwa1 = f writes a value 1 to memory
```

```
    long c = a;          // frac = f reads a value c from memory
```

```
    long d = b;          // frbd = f reads b value d from memory
```

```
    long u = c + d;      // fadd = f adds c and d locally
```

```
    return (void*) u;
```

```
}
```

```
void * g (void * p) {
```

```
    b = 1;                // gwb1 = g writes b value 1 to memory
```

```
    long e = b;          // grbe = g reads b value e from memory
```

```
    long f = a;          // graf = g reads a value f from memory
```

```
    long v = e + f;      // gadd = g adds e and f locally
```

```
    return (void*) v;
```

```
}
```



## common sequentially consistent interleaved scenario with result 3

```
long a, b;

void * f (void * q) {
    a = 1;           // fwa1
    long c = a;      // frac
    long d = b;      // frbd
    long u = c + d;  // fadd
    return (void*) u;
}

void * g (void * p) {
    b = 1;           // gwb1
    long e = b;      // grbe
    long f = a;      // graf
    long v = e + f;  // gadd
    return (void*) v;
}

abcdefghijklmnop memory-fifo
00----- fwa1
00----- fwa1 frac frbd
10----- frac frbd
101----- frbd
1010----- gwb1
1010----- gwb1 grbe
1010----- gwb1 grbe graf
1110----- grbe graf
11101---- graf
111011--- fadd
111011--- fadd gadd
111011--- fadd gadd madd
1110112-- gadd madd
11101122- madd
111011223
```

## rare sequentially consistent interleaved scenario with result 4

```
long a, b;

void * f (void * q) {
    a = 1;           // fwa1
    long c = a;      // frac
    long d = b;      // frbd
    long u = c + d;  // fadd
    return (void*) u;
}

void * g (void * p) {
    b = 1;           // gwb1
    long e = b;      // grbe
    long f = a;      // graf
    long v = e + f;  // gadd
    return (void*) v;
}

abcdefghijklmnop memory-fifo
00----- fwa1
00----- fwa1 gwb1
00----- fwa1 gwb1 frac
00----- fwa1 gwb1 frac grbe
00----- fwa1 gwb1 frac grbe frbd
00----- fwa1 gwb1 frac grbe frbd graf
10----- gwb1 frac grbe frbd graf
11----- frac grbe frbd graf
111----- grbe frbd graf
111-1---- frbd graf
11111---- graf
111111--- fadd
111111--- fadd gadd
111111--- fadd gadd madd
1111112-- gadd madd
11111122- madd
111111224
```

## less frequent sequentially inconsistent scenario with result 2

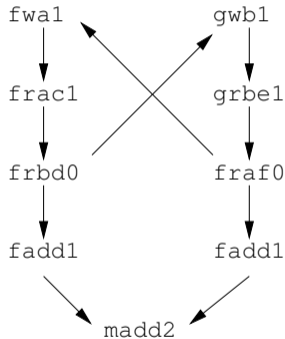
```
long a, b;                                abcdefuvm memory-fifo
                                           00----- fwa1
void * f (void * q) {                     00----- fwa1 frac frbd      # frac 0o0
    a = 1;                                // fwa1    00----- fwa1 frbd      # frbd 0o0
    long c = a;                            // frac    001----- fwa1
    long d = b;                            // frbd    0010----- fwa1 gwb1
    long u = c + d;                        // fadd    0010----- fwa1 gwb1 grbe
    return (void*) u;                     0010----- fwa1 gwb1 grbe graf # grbe 0o0
}                                           00101---- fwa1 gwb1 graf      # graf 0o0
                                           001010--- fwa1 gwb1
void * g (void * p) {                     101010--- gwb1
    b = 1;                                // gwb1    111010--- fadd
    long e = b;                            // grbe    111010--- fadd gadd
    long f = a;                            // graf    111010--- fadd gadd madd
    long v = e + f;                        // gadd    1110101-- gadd madd
    return (void*) v;                     11101011- madd
}                                           111010112
```

## no sequentially consistent scenario with result 2

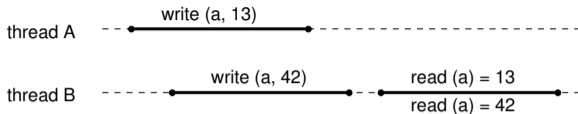
```
long a, b;

void * f (void * q) {
    a = 1;           // fwa1
    long c = a;     // frac
    long d = b;     // frbd
    long u = c + d; // fadd
    return (void*) u;
}

void * g (void * p) {
    b = 1;           // gwbl
    long e = b;     // grbe
    long f = a;     // graf
    long v = e + f; // gadd
    return (void*) v;
}
```

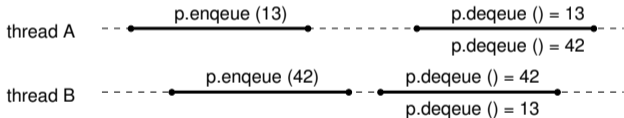


# Linearizability



## ■ consistency can be extended to method calls

- method calls take time during a time interval: invocation to response
- example above with read / write on memory
- below with enqueue / dequeue on queue

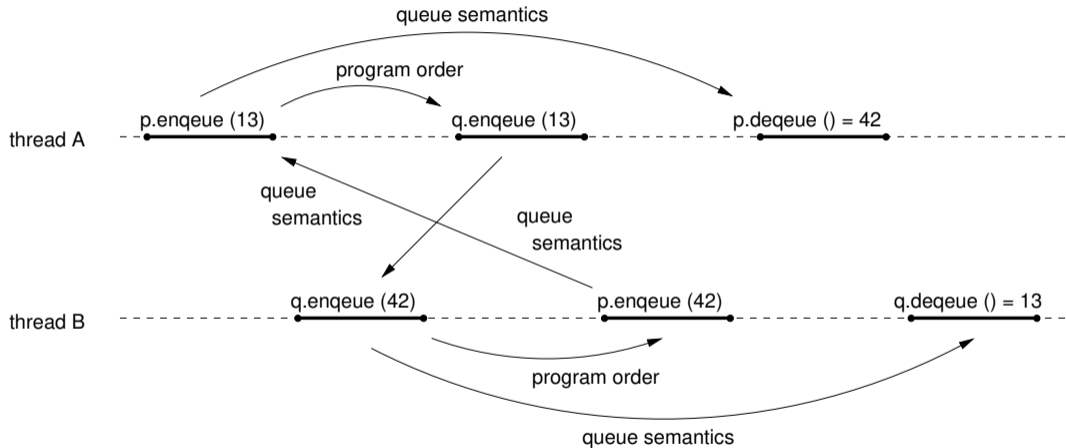


## ■ execution **linearizable** iff

there is a linearization point between invocation and response  
where the method appears to take effect instantaneously

## ■ at the linearization point the effect of a method becomes visible to other threads

# locally sequentially consistent but globally not (nor linearizable)



## Progress Conditions: Wait-Free, Lock-Free

- a **total** method is defined in any state, otherwise **partial**
  - like “enqueue” is partial and “dequeue” (in an unbounded queue) is total
  - same for “read” and “write”
- method is **blocking** iff response can not be computed immediately
  - common scenario in multi-processor systems
- linearizable computations can always be extended with pending responses of total messages
  - so in principle pending total method responses never have to be blocking
  - but it might be difficult to compute the actual response
- method  $m$  **wait-free** iff every invocation eventually leads to a response
  - in the strong liveness sense, e.g., within a finite number of steps
  - or in LTL  $\forall m[G(m.\text{invocation} \rightarrow F m.\text{response})]$
- method  $m$  **lock-free** iff infinitely often some method call finishes
  - so some threads might “starve”, but the overall system makes progress
  - or in LTL  $(\exists m[GF m.\text{invocation}]) \rightarrow GF \exists m'[m'.\text{response}]$
- every wait-free method is also lock-free
  - wait-free provides stronger correctness guarantee
  - usually minimizes “latency” and leads to less efficiency in terms of through put
  - and is harder to implement

## Compare-And-Swap (CAS)

```
// GCC's builtin function for CAS

bool __sync_bool_compare_and_swap (type *ptr, type oldval, type newval);

// it atomically executes the following function

bool CAS (type * address, type expected, type update) {
    if (*address != expected) return false;
    *address = update;
    return true;
}
```

- considered the “mother” of all atomic operations
  - many modern architectures support CAS
  - alternatives: load-linked / store-conditional (LL/SC)
  - see discussion of memory model for RISC-V too
- compiler uses CAS or LL/SC to implement other atomic operations
  - if processors does not support corresponding operations
  - like atomic increment
  - C++11 atomics



# Treiber Stack

Treiber, R.K..

Systems programming: Coping with parallelism.

IBM, Thomas J. Watson Research Center, 1986.

- probably first lock-free data-structure
- implements a parallel stack
- suffers from ABA problem
- see demo

## Others

hazard pointers

false sharing

queues (Michael & Scott Queue)

relaxed data structures ( $k$ -stack)

Andreas Haas, Thomas Hütter, Christoph M. Kirsch, Michael Lippautz, Mario Preishuber, Ana Sokolova:  
Scal: A Benchmarking Suite for Concurrent Data Structures.

NETYS 2015: 1-14

<http://scal.cs.uni-salzburg.at>

Paul E. McKenney

Is Parallel Programming Hard, And, If So, What Can You Do About It?

<https://mirrors.edge.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html>