LINZ INSTITUTE
OF TECHNOLOGY

PARALLEL COMPUTING
Shared Memory

Univ.-Prof. Dr. Alois Zoitl
LIT | Cyber-Physical Systems Lab
Johannes Kepler University Linz

To whom honor is due....

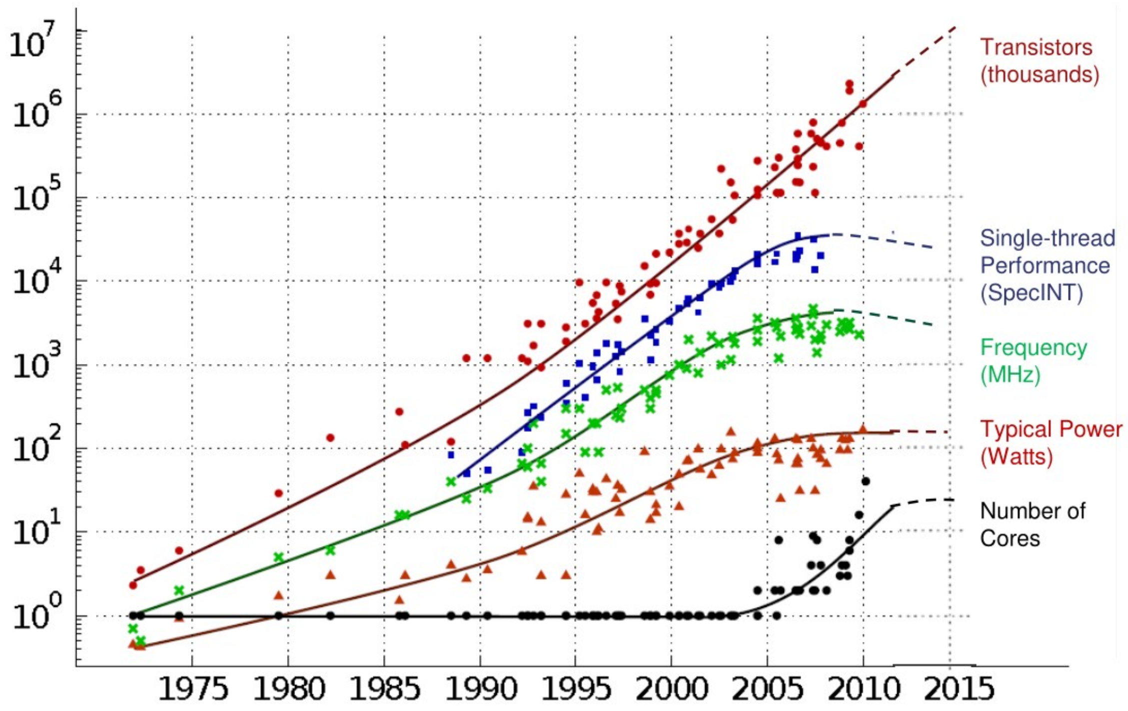
These slides are based on a slide deck from

Prof. Dr. Armin Biere

from whom I took over this lecture.
He deserves thanks for his kind permission to use them.

Why Shared Memory?

- Wide-spread availability of multi-core
  - in servers for more than 20 years
  - desktop for more than 15 years
  - GPU computing for more than 15 years
  - smart phones for more than 10 years
- Power limits in CMOS technology
  - Around 2005 frequency scaling stopped
  - Moore’s law still continued to hold
  - More cores instead of higher frequency



Source: Chuck Moore, Data Processing in Exascale-Class Computer Systems, 2011

Processes vs. Threads

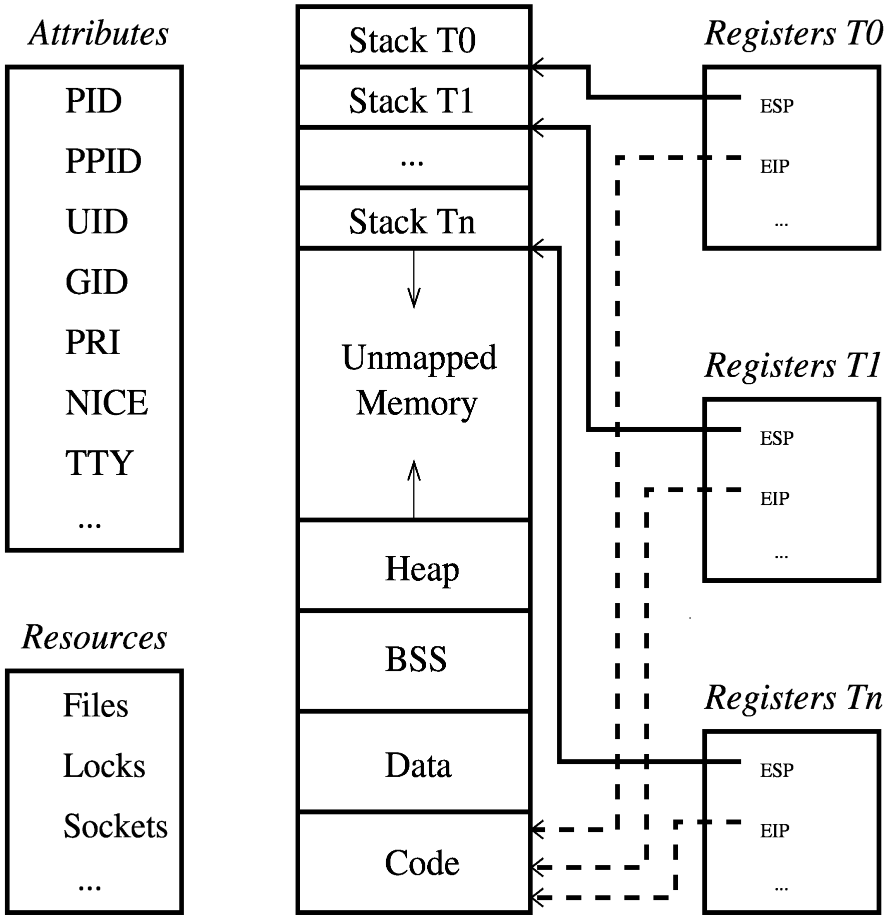
- **Processes**
  - Classical but more complicated
  - Fork / join paradigm
  - Communication over files / pipes
  - mmap (..., MAP_SHARED, ...)

- **Threads**
  - “Known” programming model
  - Similar to sequential model
  - But with globally shared memory
  - Multiple processing units

Threads vs. Processes

- Process can have multiple threads
- Thread: ***lightweight*** process
- Threads share
  - Address space
  - File descriptors
  - Sockets
  - ...
- Per-thread
  - Stack,
  - Program counter
  - Registers: thread's context
- Switching threads more efficient than switching processes

  → ***lightweight*** context



Benefits of Threading

- Parallelism
  - Computing independent tasks at the same time
    - speed-up (Amdahl's Law!)
  - Need multiprocessor HW for “true” parallelism
  - Exploiting capabilities of modern multi-core processors
- Concurrency
  - Progress despite of blocking (overlapping) operations
  - No multiprocessor HW needed
  - “Illusion” of parallelism
    - Analogy: multiple running processes in multi-tasking operating systems

- Threaded programming model
  - Shared-memory (no message passing)
  - Sequential program:
    implicit, strong synchronization via ordering of operations
  - Threaded program:
    explicit code constructs for synchronizing threads
  - Synchronization clearly designates dependencies
  - Better understanding of “real” dependencies

Costs of Threading

- Overhead
  (Synchronization, Computation)
  - Directly:
    More synchronization → less parallelism,
    higher costs
  - Indirectly:
    - Scheduling
    - Memory architecture (cache coherence)
    - Operating system
    - Calling C library
    - …
- Programming discipline
  - “thinking in parallel”
  - Careful planning
  - Avoidance of
    - **Deadlocks**: circular waiting for resources
    - **Races**: threads' speed (scheduling) determines outcome of operation

- Debugging and Testing
  - Nondeterminism:
    Timing of events depends on threads' speed (scheduling)
  - Bugs difficult to reproduce
    - e.g. what thread is responsible for invalid memory access?
  - Probe effect:
    Adding debugging information can influence behavior
  - How to test possible interleaving of threads?

When (not) to Use Threads?

- Pro threads
  - Independent computations on decomposable data
    - Example: arraysum
  - Frequently blocking operations e.g. waiting for I/O requests
  - Server applications
- Contra threads
  - Highly **sequential** programs: every operation depends on the previous one
  - Massive synchronization requirements

- Challenges in Threaded Programming (applies to parallel computation in general)
  - Amdahl's Law is optimistic (ignores underlying HW, operating system,...)
  - Keeping the sequential part small: less synchronization
  - Increasing the parallel part: data decomposition

POSIX Threads (Pthreads) Basics

POSIX Threads

- POSIX: Portable Operating System Interface
  - IEEE standards defining API of software for UNIX-like operating systems
- POSIX threads (Pthreads)
  - Standard approved 1995, amendments
  - Functions for
    - Creating threads
    - Synchronizing threads
    - Thread interaction
  - Opaque data types for
    - Thread identifiers
    - Synchronization constructs
    - Attributes
    - ...
- Usage
  - Header file: `pthread.h`
  - Compilation:

```
gcc -pthread -o prog prog.c
```

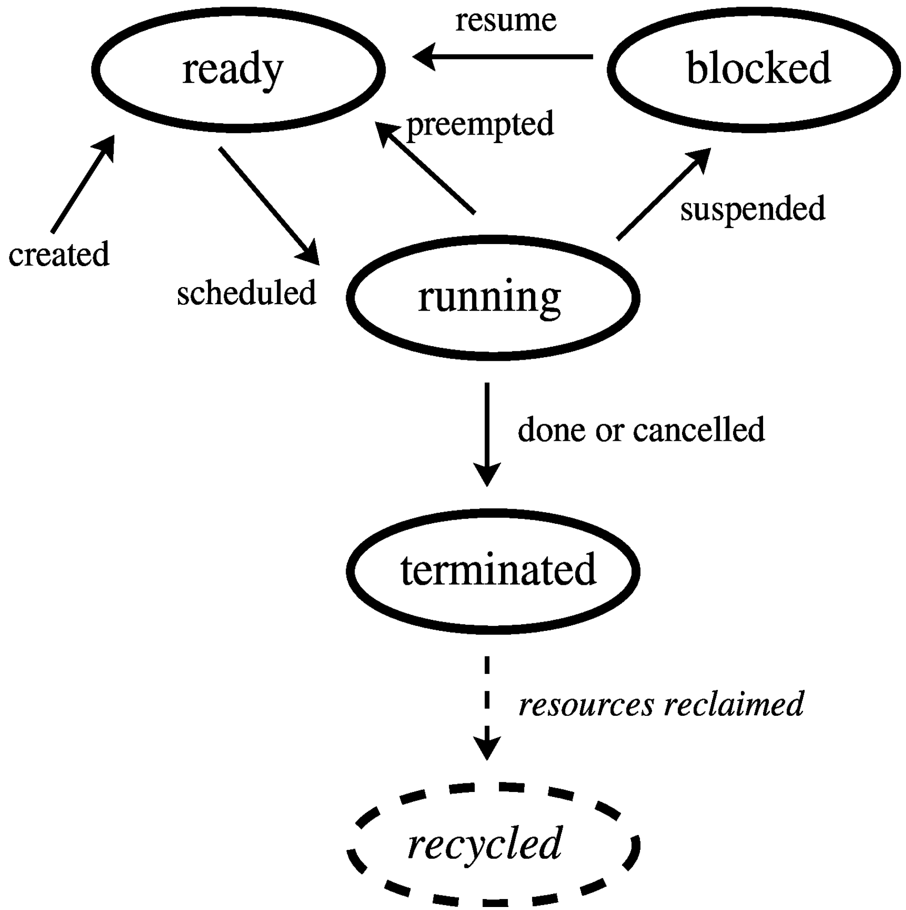
References:
D. R. Butenhof, Programming with POSIX Threads, Addison-Wesley, 1997
http://opengroup.org/onlinepubs/007908799/xsh/pthread.h.html

(P)Threads in Linux

- How can a thread-library be implemented?
- Abstraction levels:
  - Threads: created by a user program
  - Kernel entity: “process”, scheduled by operating system
  - Processor: physical device, gets assigned kernel entities by scheduler
- Design decision: How to map threads to kernel entities?
  - M-to-1:
    - All threads of process mapped to one kernel entity
    - Fast scheduling (in library), but no parallelism
  - M-to-N:
    - Threads of process mapped to different kernel entities
    - Two-level scheduling (library and kernel) incurs overhead, but allows parallelism
  - 1-to-1:
    - Each thread mapped to one kernel entity
    - Scheduling in kernel, less overhead than in M-to-N case, allows parallelism
    - Used in most modern Linux systems: Native POSIX Threads Library (NPTL)

Pthread Lifecycle: States

- Ready
  - Able to run, waiting for processor
- Running
  - On multiprocessor possibly more than one at a time
- Blocked
  - Thread is waiting for a shared resource
- Terminated
  - System resources partially released
  - But not yet fully cleaned up
    - Thread's own memory is obsolete
    - Can still return value
- (Recycled)
  - All system resources fully cleaned up
  - Controlled by the operating system



Pthread Creation

- `int pthread_create(arg0, arg1, arg2, arg3)`
  - `arg0: pthread_t *tid_ptr`
    - Where to store thread ID of type pthread_t
  - `arg1: const pthread_att_t *attr`
    - May set certain attributes at startup
    - Ignored for the moment: always pass `NULL` → set default attributes
  - `arg2: void *(*start)(void *)`
    - Pointer to thread's startup function
    - Takes exactly one void* as argument
  - `arg3: void *arg`
    - Actual parameter of thread's startup function
  - Returns zero on success, else error code
- Thread ID is stored in `*tid_ptr`
  - `pthread_t pthread_self()` returns ID of current thread
  - int `pthread_equal(pthread_t tid1, pthread_t tid2)` compares IDs
- Example: `helloworld`

Main-Thread

- Process creates thread which executes main-function ➝ “main-thread”
- Main-thread behaves slightly differently from ordinary threads:
  - Termination of main-thread by returning from main causes process to terminate
    - All threads of process terminate
    - Example: `helloworld`
  - Calling pthread_exit(...) in main-thread causes process to continue
    - All created threads continue
    - Recall lifecycle:
      - main-thread terminates ➝ resources partially released
      - **Attention:** stack may be released!
    - Memory errors: dereferencing pointers into main-thread's (released) stack
    - Example: `helloworld_buggy`

Pthread Termination

- **Generally:** thread terminates if startup function returns
- `int pthread_exit(void *value_ptr)`
  - Causes thread to terminate (special semantics in main-thread)
  - Implicitly called if thread's startup function returns (except in main-thread)
  - `value_ptr` is the thread's return value (see `pthread_join(...)`)
- `int pthread_detach(pthread_t tid)`
  - Resources of `tid` can be reclaimed after tid has terminated
  - Default: not detached
  - Any thread can detach any thread (including itself)
- `int pthread_join(pthread_t tid, void **value)`
  - Returns when `tid` has terminated (or already terminated), caller blocks
  - Optionally stores `tid`'s return value in *value
    - Return value from calling `pthread_exit(...)` or returning from startup function
  - Joined thread will be implicitly detached
  - Detached threads can not be joined

Pthread Termination - Examples

- Example: `helloworld_join`
- Returning values from threads
  - Returning values from threads via `pthread_join(...)`
    - Example: `returnval`
    - **But:** waiting for termination often not needed
    - Good practice to release system resources as early as possible
  - Alternative to `pthread_join(...)`: custom return mechanism
    - Threads store their return values on the heap
    - Example: `returnval_heap`
      - **Problem:** need to notify main-thread somehow that all threads have written results
  - **Error:** joining a detached thread
    - resources are (may be or not) already released
    - join should fail
    - Example: `returnval_buggy`
  - **Error:** returning pointer to local variable
    - Example: `returnval_buggy`

Pthread Lifecycle Revisited (1/2)

- Creation
  - Process creation ➝ main-thread creation
  - `pthread_create(...)`: new threads are ready
    - No synchronization between `pthread_create(...)` and new thread's execution
- Startup
  - Main-thread's main function called after process creation
  - Newly created threads execute startup function
- Running
  - Ready threads are eligible to acquire processor ➝ will be running
  - Scheduler assigns time-slice to ready thread ➝ threads will be preempted
  - Switching threads ➝ context (registers, stack, pc) must be saved
- Blocking
  - Running threads may block, e.g. to wait for shared resource
  - Blocking threads become ready (not running) again

Pthread Lifecycle Revisited (2/2)

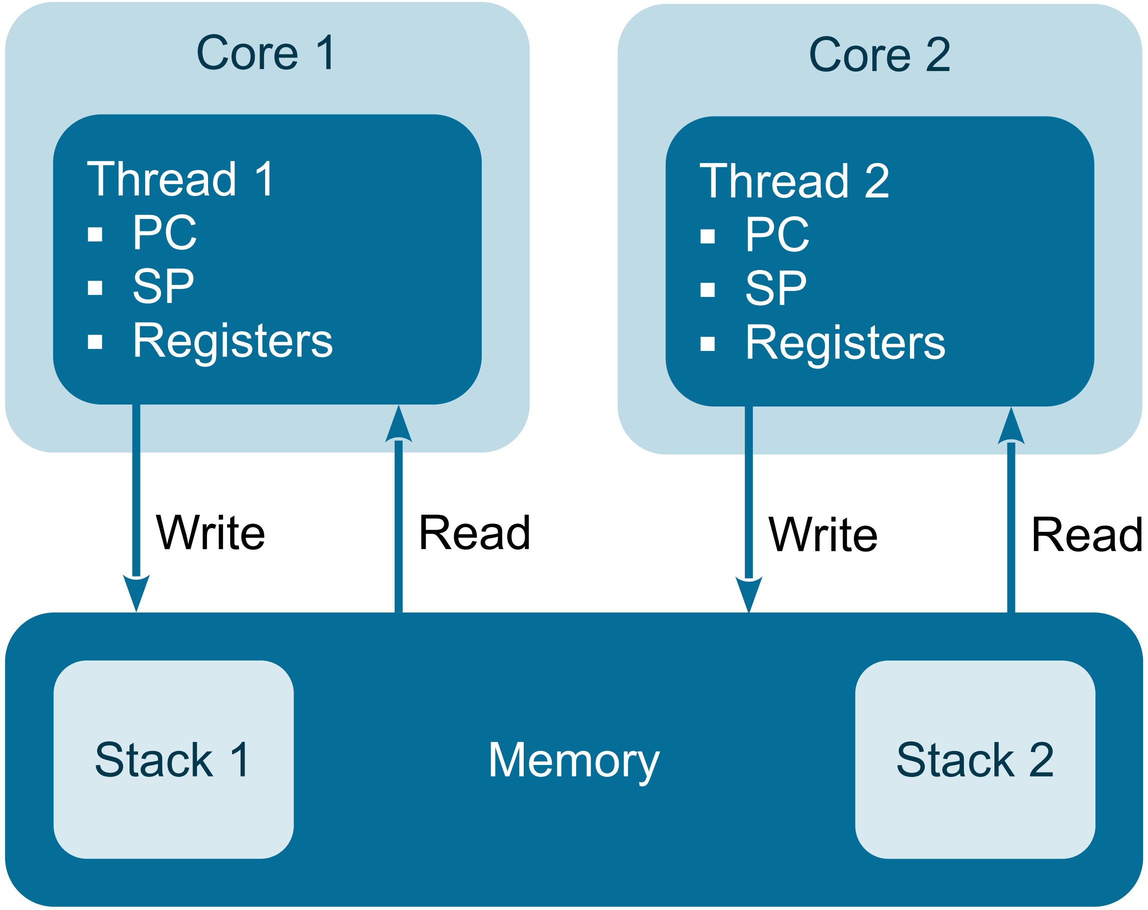
- Termination
  - Generally: when thread returns from startup function
  - `pthread_exit`
  - Can also explicitly be canceled by `pthread_cancel(...)`
  - Optional cleanup handlers are called
  - Only thread's ID and return value remain valid, other resources might be released
  - Terminated threads can still be joined or detached
  - Joined threads will be implicitly detached, i.e. all its system resources will be released
- Recycling
  - Occurs immediately for terminated, detached threads → all resources released

Creating and Using Threads: Pitfalls

- Sharing pointers into stack memory of threads
  - Perfectly alright, but handle with care
    - Passing arguments
    - returning values
- Resources of terminated, non-detached threads can not fully be released
  - Large number of threads → performance problems?
  - Should join or detach threads
- Relying on the speed/order of individual threads
  - Do not make any assumptions!
  - Need mechanism to notify threads that certain conditions are true
    - Example: returnval_heap
  - Must prevent threads from modifying shared data concurrently
    - Example: sum
- → Synchronization

Shared Memory Programming

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

- 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

Detecting Potential Races: 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 suppressed automatically
- For instance implemented in helgrind
- Major problem is that it needs “sandboxing” (interpreting memory accesses)

Detecting Potential Races: Tools

- Testing with massif load (schedule steering)
- Detection tools
  - Valgrind:
    `valgrind –tool=helgrind`
  - ThreadSanitizer:
    `gcc -fsanitize=thread`

- Code sanitizers (https://en.wikipedia.org/wiki/Code_sanitizer)
  - AddressSanitizer
  - LeakSanitizer
  - MemorySanitizer
  - UndefinedBehaviorSanitizer

Avoiding Data Races Through Locking / Mutual Exclusion

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

- Pthread offers Mutex → Slow
- How to implement locking?
  - Will first look at software only solutions
  - Hardware solutions much more efficient

Mutual Exclusion with Deadlock

```c
#include ...
pthread_t t0, t1;
int x;
int id [] = { 0 , 1 };
int flag [] = { 0 , 0 };

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

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

```c
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” = liveliness 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

```c
#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: write(flag[0], 1)` `1: write(flag[1], 1)`
  - `0: write(victim, 0)` `1: write(victim, 1)`
  - `0: read(flag[1]) = 1` `1: read(flag[0]) = 1`

- possible:
  - `0: read(flag[1]) = 0` `1: read(flag[0]) = 0`
  - `0: write(flag[0], 1)` `1: write(flag[1], 1)`
  - `0: write(victim, 0)` `1: write(victim, 1)`

Mutual Exclusion Algorithms

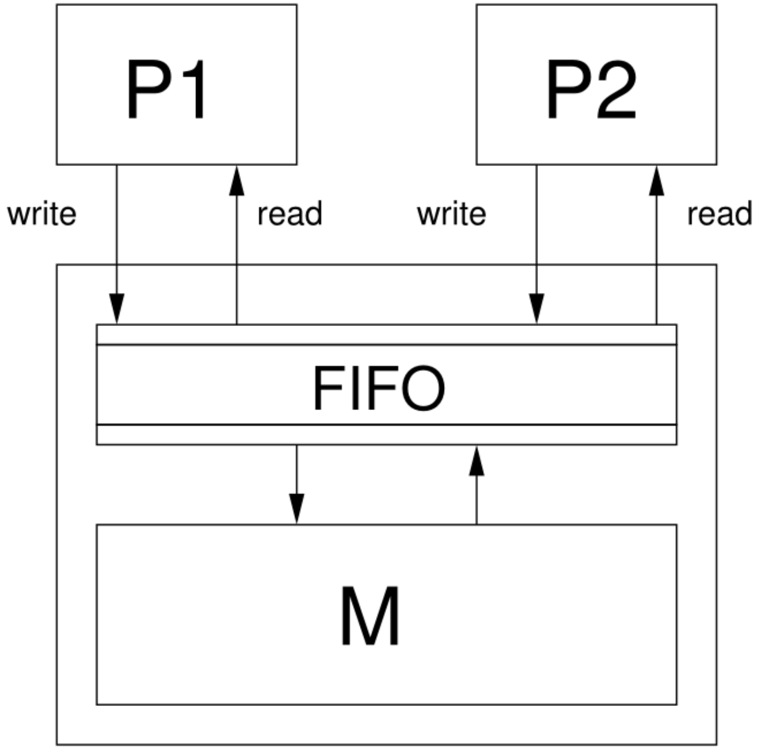
- Classical “software-only” algorithms
  - More of theoretical interest only now
  - Because memory model of multi-core machines weak (reorders reads and writes)
  - But would be on reorder-free hardware still not really efficient (in space and time)
- Need hardware support anyhow
  - Various low-level (architecture) dependent primitives
  - Atomic increment, bit-set, compare-and-swap and memory fences
  - Better use platform-independent abstractions, such as pthreads
- We will latter see how-those low-level primitives can be used

Sequential Consistency

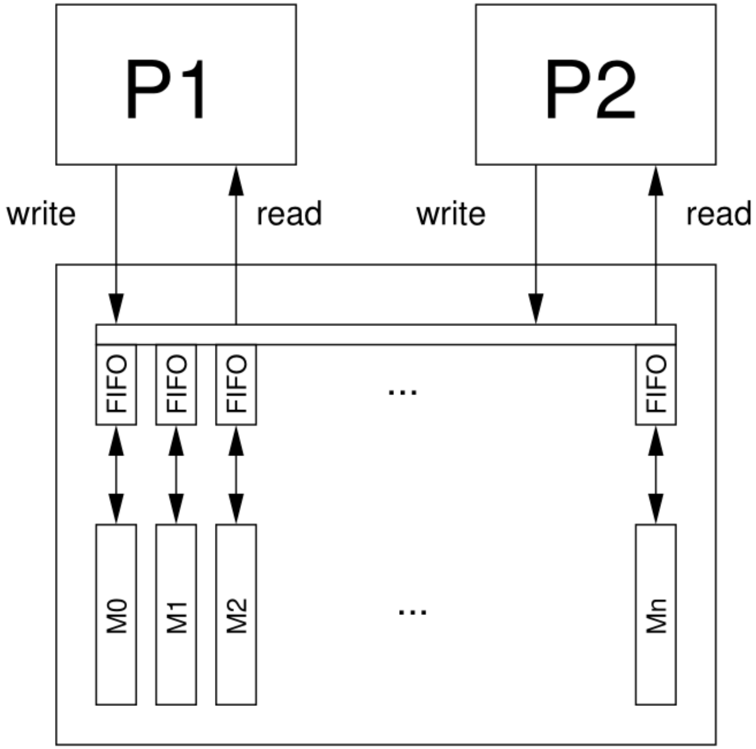
Leslie Lamport: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. IEEE Trans. Computers 28(9): 690-691 (1979)

- 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

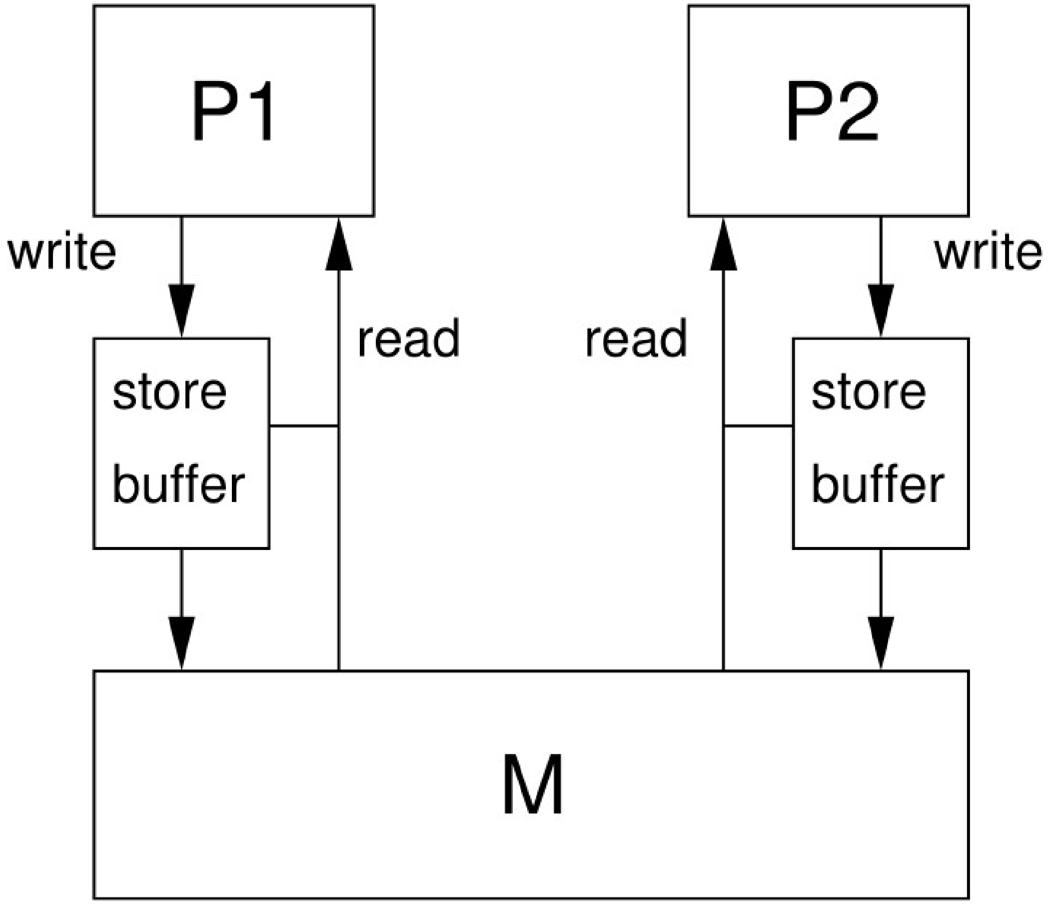


Global FIFO read / write operation gives sequential consistency



Projected to individual memory addresses too

Store Buffer / Write Buffer



Hide write latency by collecting written data and continue serving read data
(already in the cache or in the write buffer)

Out-of-Order Write-to-Read

```c
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;
}
```

```c
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;

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
  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;
}
```

```
abcdefuvm  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
1110111--  gadd madd
11101112-  madd
111011123
```

Rare Sequentially Consistent Interleaved Scenario with Result 4

```
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
  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;
}
```

```
abcdefuvm  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
```

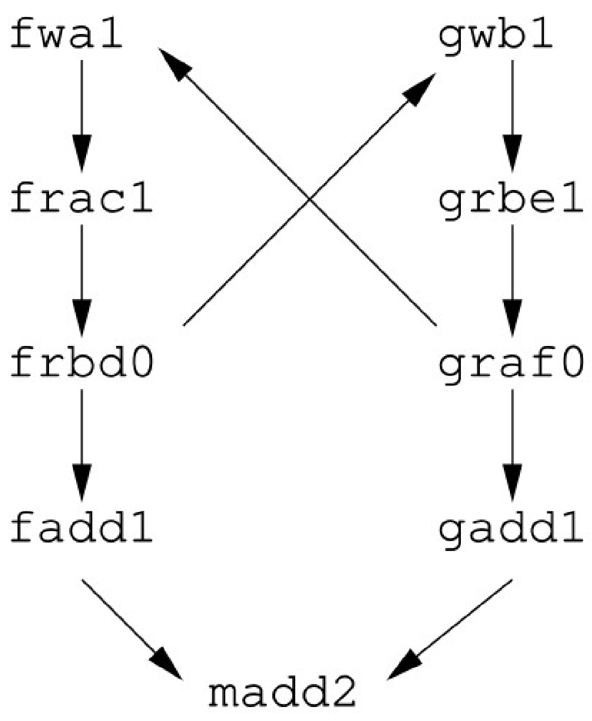
Less Frequent Sequentially *Inconsistent* Scenario with Result 2

```
long a, b;

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
  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;
}
```

```
abcdefuvm  memory-fifo
00-------  fwa1
00-------  fwa1 frac frbd
001------  fwa1 frbd          // frac
ooo
0010-----  fwa1 gwb1
0110-----  fwa1
0110-----  fwa1 grbe
01101----  fwa1 graf
011010---  fwa1
111010---  fadd
111010---  fadd gadd
111010---  fadd gadd madd
1110101--  gadd madd
11101011-  madd
111010112
```

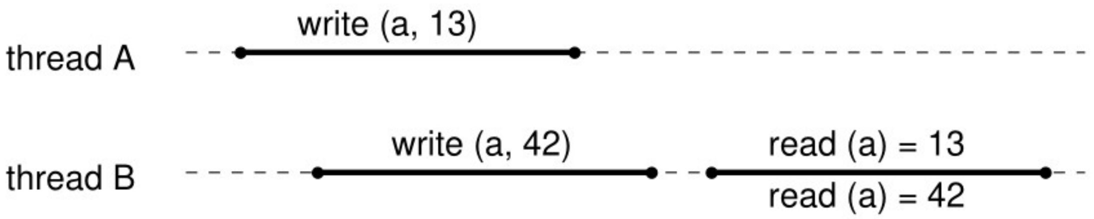
No Sequentially Consistent Scenario with Result 2

```c
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
  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;
}
```

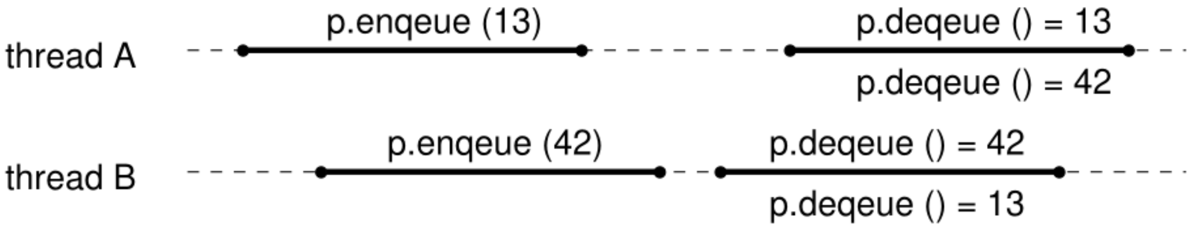


Linearizability

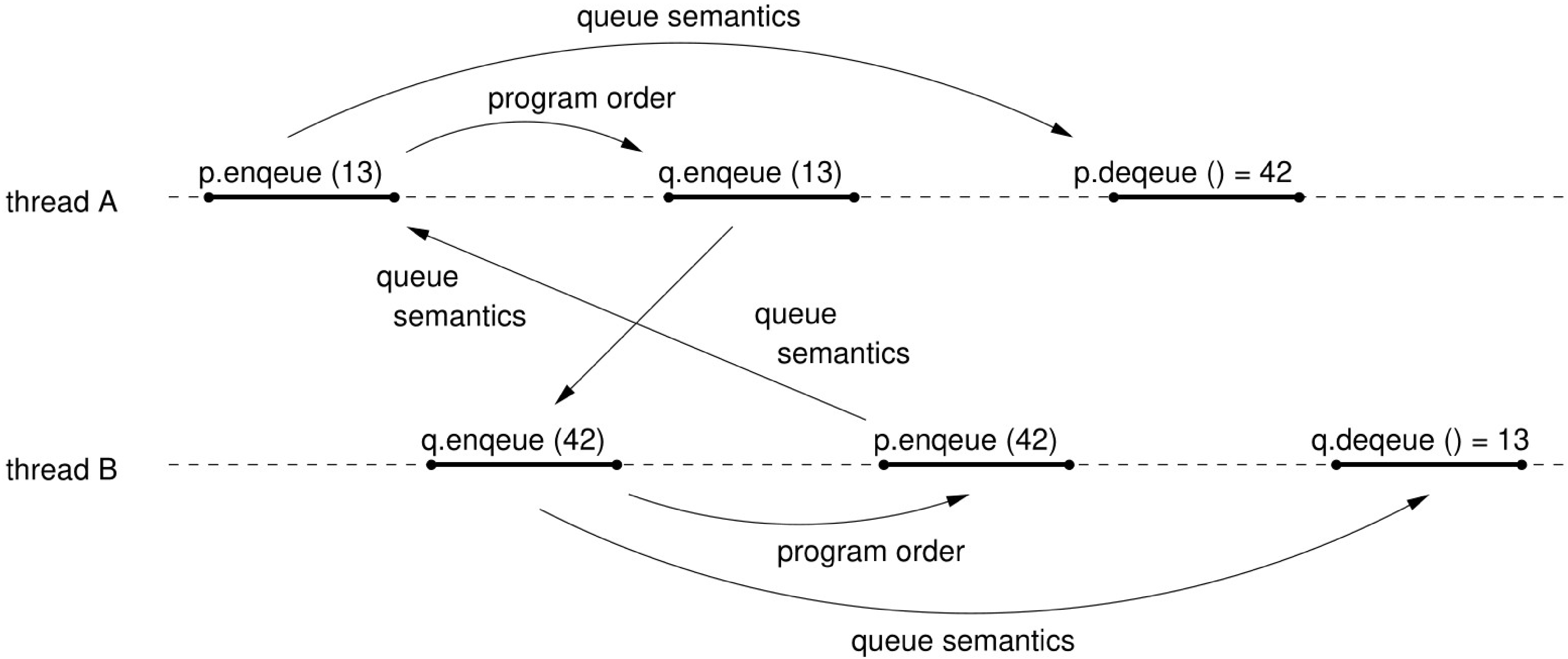
- Consistency can be extended to method calls
  - Method calls take time during a time interval: invocation to response
  - Example above with read / write on memory
  - Below with enqueue / dequeue on queue



- 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 “dequeue” is partial and “enqueue” (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.invocation $\rightarrow$ F m.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 ($\exists m$[GFm.invocation]) $\rightarrow$ GF $\exists m'$[m'.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
  - 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 foundation 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

Thank you!

Univ.-Prof. Dr. Alois Zoitl, alois.zoitl@jku.at
LIT | Cyber-Physical Systems Lab
Johannes Kepler University Linz