Contents of Lecture 8

- POWER sync instructions for C11
- Cell
- OpenMP
- Rust
Consider the following code and assume the compiler has deduced size is one:

```c
i = atomic_load(q, memory_order_consume);
a = b[i % size];
```

What happens if the compiler transforms the code to the following?

```c
i = atomic_load(q, memory_order_consume);
a = *b;
```
There is now **no dependence** between the two statements!

```c
i = atomic_load(q, memory_order_consume);
a = *b;
```

Without a dependence the reads are not ordered.

This is not what the programmer expected!!!

Therefore, optimizing C compilers must analyse all dependences **before** any code transformation and preserve them.

What does "preserve" mean, then?
Preserving the data dependency means letting the hardware know about them.

That can be done in different ways for different processors.

One way is to insert instructions so that there will be a chain of dependences for the processor pipeline to see.

Compilers thus must preserve such dependences in the complete program!

Writing optimizing C compilers all of a sudden became twice as interesting with C11.

There is, however, a very simple first version implementation: treat all memory_order_consume as memory_order_acquire which automatically will order the memory accesses.
One More Data Dependency

a = atomic_load(p, memory_order_consume);
atomic_store(q, a, memory_order_relaxed);
atomic_store(r, b, memory_order_relaxed);

- Since there is no data dependency between the consume and the second store, they are not ordered.
- As programmers we need to be very careful about what we expect to be ordered!
Implementation on a specific architecture: POWER.

Illustrations of what actually is ordered for several situations.

We will begin with the POWER synchronization instructions.

Approximate clock counts below are not fixed but depends on what is happening in the machine at the moment.

Numbers from Christoph von Praun: "Deconstructing Redundant Memory Synchronization" (while at IBM Research).

<table>
<thead>
<tr>
<th>mnemonic</th>
<th>name</th>
<th>POWER4 cycles</th>
<th>POWER5 cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldarw/stwxc</td>
<td>load and reserve / store conditional</td>
<td>80</td>
<td>75</td>
</tr>
<tr>
<td>hwsync</td>
<td>heavy weight sync</td>
<td>140</td>
<td>50</td>
</tr>
<tr>
<td>lwsync</td>
<td>light weight sync</td>
<td>110</td>
<td>25</td>
</tr>
<tr>
<td>isync</td>
<td>instruction sync</td>
<td>30</td>
<td>10</td>
</tr>
<tr>
<td>eieio</td>
<td>enforce inorder execution of I/O</td>
<td>not measured</td>
<td>not measured</td>
</tr>
</tbody>
</table>

We want to use functions which use the less costly instructions!

Such as the consume operation which is only an ordinary load instruction on POWER!
Some processors use atomic test-and-set instructions while others use pairs of special load and store instructions.

Load and reserve is also called load-locked and load-linked.

In addition to POWER, it’s used by ARM and MIPS.

Test-and-set, or compare-and-swap, is used by e.g. x86.

The purpose with load-and-reserve/store conditional is to simplify the design of the pipeline.
The load instruction fetches data from a memory location, and makes a reservation $R$ in memory.

If a different processor modifies the same memory location, the reservation $R$ is lost.

If/when the processor with a reservation makes a conditional store, the memory location is modified only if the reservation was not lost in between.

Therefore it’s an atomic read-modify-write.

The \texttt{stwcx.} conditional store in POWER sets a condition code to indicate whether the store succeeded or not (the dot in the mnemonic indicates that the condition code is set by an operation on POWER).
The POWER memory fence instructions create a memory fence with two sets of instructions:

- the $A$-set with instructions $a_i$ preceding the fence, and
- the $B$-set with instructions $b_j$ following the fence.

Depending on which fence instruction is used, some $b_j$ instructions may be reordered with some $a_i$ instructions.
The $A$-set consists of all instructions preceding the $\text{hwsync}$.  

The $B$-set consists of memory access instructions following the $\text{hwsync}$.  

Except for the instruction $\text{icbi}$ which invalidates an instruction cache block, no $B$-set instruction may be reordered with any $A$-set instruction.  

This is the most costly POWER synchronization instruction.  

Not only for cached data but for any storage.  

\texttt{hwsync} used e.g. to implement sequentially consistent write on POWER:  
\begin{verbatim}
stwx r1,r2,r3
\text{hwsync}
\end{verbatim}
The $A$ and $B$ sets consist of all memory access instructions preceding and following the `lwsync`, respectively.

Only the following pairs of $a_i$, $b_j$ instructions are ordered:
- A-side load $\rightarrow$ B-side load
- A-side load $\rightarrow$ B-side store
- A-side store $\rightarrow$ B-side store

Thus, A-side store $\rightarrow$ B-side load are not ordered.

The `dcbz` data cache block zero is counted as a store.

`lwsync` used e.g. to implement a release:

```
stwx r1,r2,r3  # modify shared data...
stwx r4,r5,r6
...
stwx r29,r30,r31  # last write in critical section
lwsync
stwx r8,r9,r10  # set lock to free
```
isync Memory Fence for Acquire Operation

- The $A$ and $B$ sets consist of all instructions preceding and following the isync, respectively.
- An instruction which cannot raise an exception in the pipeline can be allowed to complete and instructions following the isync therefore actually execute without order.
- Consider the sequence:
  
  ```
  ldw r1,r2,r3
  isync
  stw r4,r5,r6
  ```
- The store may execute before the load if the load had e.g. a cache miss.
- This is not what we want and to overcome that problem we can exploit that POWER does not permit speculative execution of store instructions.
By inserting a conditional branch, \texttt{bc}, before the \texttt{isync} both the \texttt{isync} and \texttt{stw} become speculative until the branch outcome is known.

We can use \texttt{beq} as follows:

\begin{verbatim}
ldw      r1,r2,r3  # r1 = MEMORY[r2+r3]
cmp.    r1,r1     # certainly true but the \texttt{beq} must
beq      # wait according to the specification
isync    # since no speculative \texttt{stw} is allowed.
stw      r4,r5,r6
\end{verbatim}

Since the store may not execute speculatively it must wait for the branch outcome.

This memory fence is the fastest.

The previous two, however, can order instructions from different processors due to the \texttt{hwsync/lwsync} are \textbf{cumulative} — see below.
eieio Memory Fence

- (unique five vowel instruction)
- Enforce in order execution of I/O.
- It only orders stores.
If the memory accesses ordered by a memory fence executed by one processor $P_i$ also take into account memory accesses executed by other processors as described below the fence is **cumulative**.

By **applicable** storage accesses for a fence is meant the storage access which are ordered by that fence.

Two rules:

- The $A$-set also includes all applicable storage accesses made by other processors which have completed with respect to $P_i$ before the fence is created (by executing the fence instruction).
- The $B$-set also includes all applicable accesses made by any processor $P_j$ after $P_j$ has executed a load that returned a value stored by an instruction in the $B$-set.

- The $B$-set expands recursively.

The next example will clarify this.
Example of Cumulative Ordering

```c
// int a = b = 0; // Thread 1
1:a = 1;  Thread 2
2:lwsync
3:x = a;  // a = 1
4:y = b;
5:lwsync
6:b = 2;
7:x = b;  // b = 2
8:lwsync
9:y = a;  // a = 1
```

- `lwsync` is cumulative
- `bc;isync` is not

**If** Thread 2 reads 1 in access 3 **and** Thread 3 reads 2 in access 7 **then**
   Thread 3 will read 1 in access 9.

Using instead `bc;isync` Thread 3 may read zero in access 9.
Implementation on POWER

- The following is based on the ISO C/C++ standardization document ISO/EIC JTC1 SC22 WG21 N2745 written by Paul E McKenney and Raul Silvera from IBM.
- There are other implementations possible.
1 Using sequential consistency on machines with relaxed memory models makes it easier to write correct parallel code which will be slow.

2 Even if your code is safety critical (when people can die due to bugs) and performance is not an issue, you should not use SC, use a single threaded C program instead.

<table>
<thead>
<tr>
<th>Load relaxed</th>
<th>ld</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load consume</td>
<td>ld</td>
</tr>
<tr>
<td>Load acquire</td>
<td>ld; cmp; bc; isync</td>
</tr>
<tr>
<td>Load seq cst</td>
<td>hwsync;ld; cmp; bc; isync</td>
</tr>
<tr>
<td>Store relaxed</td>
<td>st</td>
</tr>
<tr>
<td>--------------</td>
<td>----</td>
</tr>
<tr>
<td>Store release</td>
<td>lwsync;st</td>
</tr>
<tr>
<td>Store seq cst</td>
<td>hwsync;st</td>
</tr>
</tbody>
</table>

1. It's easier to program under sequential consistency!
2. Yes, but, why would you want to use synchronization instructions before every store in a critical section since nobody should look at the data before you have left the critical section anyway?
3. Write buffering permitted by stores in relaxed memory models (C/Java) allows the machine to pipeline all the stores in the critical section.
4. At the end of the critical section you use store release and wait for the remaining preceding stores (now possibly waiting in a write buffer) to complete (i.e. invalidate the other copies if there are any left).
<table>
<thead>
<tr>
<th>Fence Type</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acquire fence</td>
<td>lwsync</td>
</tr>
<tr>
<td>Release fence</td>
<td>lwsync</td>
</tr>
<tr>
<td>Acq rel fence</td>
<td>lwsync</td>
</tr>
<tr>
<td>Seq cst fence</td>
<td>hwsync</td>
</tr>
</tbody>
</table>
Acquire and Release a Spin Lock

# acquire
loop:  lwarx r6,0,r3  # load lock and reserve
cmpw  r4,r6    # r4 values means unlocked
bne-  loop    # restart if locked.
stwcx. r5,0,r3 # try to store
bne   loop    # restart if store failed.
isync # store succeeded.
lwzx  r7,r8,r9 # first load of shared data

# release
stwx  r7,r8,r10 # last store of shared data
lwsync # export shared data
stw   r4,0,r3  # unlock the lock. same r4 as above.
C11 threads are similar to but simpler than Pthreads.

- `thrd_t` — thread type
- `once_flag` — a type for performing initializations exactly one time
- `mtx_t` — mutex type
- `cnd_t` — condition variable type
- `tss_t` — thread specific storage (not the same as `_Thread_local`)
void call_once(once_flag* flag, void (*func)(void));

The flag should be initialized with:

    once_flag flag = ONCE_FLAG_INIT;

If multiple threads invoke call_once with the same flag, the function will only be called one time, and the others will wait until the call to func returns.
- `thrd_success` — indicates an operation succeeded.
- `thrd_error` — indicates an operation failed but not why.
- `thrd_busy` — an operation failed due to a resource was already in use.
- `thrd_nomem` — an operation failed due to memory allocation failed.
- `thrd_timeout` — a timed wait operation timed out.
Mutex Options for `mtx_init` Function

- `mtx_plain` — the mutex should support none of below options.
- `mtx_recursive` — set mutex to support recursive locking.
- `mtx_timed` — set mutex to support timed wait.
- `mtx_try` — set mutex to support test and return.
The struct `xtime` contains at least the following members.

- `time_t sec;`
- `long nsec;`

They may be declared in any order in the struct.
Condition Variable Functions

```c
int cnd_init(cnd_t* cond);
void cnd_destroy(cnd_t* cond);
int cnd_signal(cnd_t* cond);
int cnd_broadcast(cnd_t* cond);
int cnd_wait(cnd_t* cond, mtx_t* mtx);
int cnd_timedwait(cnd_t* cond, mtx_t* mtx, const xtime* xt);
```

These functions are all similar to the corresponding Pthreads functions, except that Pthread condition variables can have certain attributes and be statically initialized.

```c
int pthread_cond_init(
    pthread_cond_t* restrict cond,
    const pthread_condattr_t* restrict attr);

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```
int mtx_init(mtx_t* mtx, int type);
void mtx_destroy(mtx_t* mtx);
int mtx_lock(mtx_t* mtx);
int mtx_timedlock(mtx_t* mtx, const xtime* xt);
int mtx_trylock(mtx_t* mtx);
int mtx_unlock(mtx_t* mtx);

- The type should be one of:
  - mtx_plain
  - mtx_timed
  - mtx_try
  - mtx_plain | mtx_recursive
  - mtx_timed | mtx_recursive
  - mtx_try | mtx_recursive

- Only mtx_trylock can return thrd_busy.
- A prior call to mtx_unlock synchronizes with a successful call to a mtx_lock function.
Thread Functions

int thrd_create(thrd_t* thr, int (*func)(void*), void* arg);
void thrd_exit(int res);
int thrd_join(thrd_t thr, int* res);
int thrd_detach(thrd_t thr);
thrd_t thrd_current(void);
int thrd_equal(thrd_t u, thrd_t v);
void thrd_sleep(const xtime* xt);
void thrd_yield(void);
Key Ideas of the Cell Processor

- Highest possible performance at a low power dissipation
- 8 SIMD vector processors added to a multithreaded PowerPC processor
- Cost: more complex programming
- Benefit for the right program: superior performance per watt.
History of the Cell Processor

1999  A partnership with IBM is proposed by Sony CEO Nobuyuki Idei.
2001  Sony, Toshiba and IBM form the STI Design Center in Austin.
2004  First Cell processor manufactured and runs faster than 4 GHz.
2005  PS3 is shown by Sony.
2006  PS3 is sold by Sony and IBM releases a Linux SDK for the PS3.
2008  IBM’s Roadrunner with 12,960 Cell processors and Opterons becomes the fastest computer in the world.
2010  IBM say they might integrate Cell technology in normal Power processors.
The Cell Broadband Engine Architecture

- A Cell processor consists of:
  - One multithreaded 64-bit PowerPC processor (currently two threads)
  - Eight SIMD vector SPU processors with 128 registers and 256 KB RAM
  - One programmable memory interface controller
  - Two input/output interfaces

- The PowerPC processor is optimized for general purpose computing and SIMD vector processing.

- The SPUs (synergistic processor units) are optimized for SIMD vector processing and concurrent data transfers.

- The SPUs have no caches and both data and instructions must be explicitly transferred from system memory by software using DMA.

- DMA = essentially hardware which acts as a for-loop and copies data.

- The idea with DMA is that the CPU can do other things.

- If programmed cleverly, the SPUs can achieve very high performance.
The Synergistic Processor Unit

- The SPU contains two pipelines, the even and the odd pipelines.
  - Even: fixed point unit and floating point unit.
  - Odd: fixed point unit, load/store unit, and Channel and DMA unit.
- 128 128-bit registers.
- All ISO C integer data types and float and double.
- 256 KB local RAM memory for instructions and data.
- No hardware caches!
- The Channel and DMA unit is used for synchronization with other processors and to transfer data.
Filling the SPU Instruction Pipeline

- Firstly, the instruction must already be in the local store.
- Before an instruction can start doing work in a functional unit in one of the even or odd pipelines, it must go through the following:
  1. Fetch: 7 cycles.
  2. Decode: 3 cycles.
  3. Issue: 2 cycles.
- ILB = instruction line buffer
- Instructions are fetched into one of ILB1 and ILB2, which can hold 32 instructions each.
- When an ILB is empty 32 new instructions are fetched from the local store. Every instruction is 32 bits.
- Instruction fetch has the lowest priority after DMA and load/store accesses.
There is no dynamic branch prediction in the SPU. Branches to lower addresses are predicted to be taken. The idea is that a loop’s branch to the next iteration is backward and is usually taken. A mispredicted branch costs 18 cycles. There are also special branch hint instructions which software can use to tell SPU about future branches.
Recall: SIMD = single instruction multiple data

E.g. a SIMD add may perform four float add operations in parallel

Both the PPU and SPU support SIMD instructions.

On the PPU, it’s the VMX or AltiVec instruction set.

On the SPU, all instructions are SIMD.

The two SIMD instruction sets are similar but not equal.

Use the vector extensions in GCC to avoid having to use specific SIMD instructions.
The size of a DMA transfer must be 1, 2, 4, 8, or a multiple of 16 and at most 16 KB.

Better performance for 128-byte aligned data.

The start address of objects must be 16-byte aligned — allocate extra memory with `malloc`.

The lower 4 bits of the effective address and the LS address must be equal.

If above rules are not followed, the program will get a bus error and terminate.
A DMA request needs the following parameters:
- effective address
- local store address
- size
- tag
- operation

The tag identifies the DMA request and is a value in the range 0..31.

The operation is either MFC_PUT_CMD or MFC_GET_CMD.

This looks like a normal memory load or store access, except for the tag and larger data size.

What is so special that the Cell processor?
Comparison with Data Prefetching

- Recall that cache misses can be difficult to avoid on multiprocessors and that data prefetching is not a universal solution.
- For instance, the prefetched data might be evicted before being used — even by other prefetches!
- The difference between normal multicores (multiprocessors) and the Cell is that you have complete control over the local store parameter in the DMA request.
- You can avoid overwriting useful data simply by using different local store addresses.
- More control in other words.
- The DMA operations are similar to data prefetches in that the processor continues with other work and the MFC performs the data transfer.
The Local Store is Not a Cache

- If prefetched data does not arrive in the cache before it’s needed, the processor will wait until it does arrive.
- That will not happen automatically in the SPU.
- Recall that the tag DMA parameter is used to identify DMA request.
- You can tell the SPU to wait until a subset of all pending DMA operations have completed.
- This is done by setting a bit corresponding to the tag you want to wait for and then issue a special command.
- This will cause the SPU to wait until all DMA operations of the specified tags have completed.
- In fact, you can use one tag for multiple DMA operations so you can have more than 32 pending operations if you wish.
- With this architecture you control parallelism between SPU’s as well as data transfer and computation within each SPU.
Ideally optimizing compilers would be able to parallelize existing C/C++ and FORTRAN codes.

From our example of the dataflow program (and other programs), I think it’s rather difficult to write such a compiler.

Instead of writing new sequential programs we can use e.g.
- C/Pthreads, C11 Threads, Java Threads
- Scala Actors

What about all existing codes?
Another option is tool support for manual parallelization:

- Programmer annotates the source code and guarantees the validity of parallelization of a loop.
- Tool support: generating parallel code for a loop

GCC supports the OpenMP standard for this approach.

Include `<omp.h>` and annotate e.g. as:

```c
#pragma omp parallel
#pragma omp for
for (i = 0; i < n; ++i) {
    /* ... */
}
```

Compile with `gcc -fopenmp`
The Main Advantage with OpenMP

- We don’t want to rewrite millions of C/C++ and FORTRAN codes from scratch.
- Using a new and relatively untested language may be a big risk.
- Untested = less than 20 years of experience...
- With OpenMP we can parallelize our applications **incrementally**.
- We can focus on one for-loop at a time.
Origin of OpenMP

- All supercomputer companies had their own compiler directives to support this “semi-automatic” parallelization.
- When SGI and Cray (one of the three Cray companies) merged they needed to define a common set of compiler directives.
- Kuck and Associates, a compiler company, and the U.S. ASCI (Accelerated Strategic Computing Initiative) joined SGI and Cray.
- In 1997 IBM, DEC, HP and others were invited to join the group now called OpenMP.
- In 1997 the specification for OpenMP 1.0 for FORTRAN was released.
- Next year the specification for C/C++ was released.
- The current version is OpenMP 4.0 and was published 2013.
- GCC 4.4 supports OpenMP.
1. Compiler directives. `#pragma` in C/C++.
2. A runtime library
3. Environment variables, like `OMP_NUM_THREADS`.

Jonas Skeppstedt (jonasskeppstedt.net)
A barrier is a synchronization primitive which makes all threads reaching the barrier wait for the last.

This is not a "memory barrier" in the sense of a C11 memory fence.

This barrier needs a lock, a counter, and knowing the number of threads.

When the last thread has reached the barrier, all threads can proceed and continue after the barrier.
A structured block of code is either
- a compound statement, i.e. a block enclosed in braces, or
- a for-loop.

The pragma `omp parallel` is used before a structured block of code and specifies all threads should execute all of that block.

Note that this is typically not what we want in a for-loop, see below.

A default number of threads is used, which can be changed with the environment variable `OMP_NUM_THREADS`, which can be larger than the number of processors in the machine.

This pragma creates an implicit barrier after the structured block.
Example

```c
#include <omp.h>
#include <stdio.h>

int main(void)
{
    #pragma omp parallel
    {
        int tid; // thread id.
        tid = omp_get_thread_num();
        printf("hello world from thread %d\n", tid);
    }
    return 0;
}
```

- Since tid is declared in the compound statement, it becomes private.
- `omp_get_thread_num()` returns an id starting with zero.
The OpenMP runtime library creates the threads it needs using Pthreads on Linux.

After a parallel block, the threads wait for the next their work and are not destroyed in between.

This model of parallelism is called **fork-join** and only the master thread executes the sequential code.

It’s possible to nest parallel regions.
Two OpenMP Functions

- To specify in the program how many threads you want, use
  ```
  omp_set_num_threads(nthread);
  ```

- To measure elapsed wall clock time in seconds, use
  ```
  double start, end;
  
  start = omp_get_wtime();
  /* work. */
  end = omp_get_wtime();
  ```
In addition to the `#pragma omp parallel` you must also specify `#pragma omp for` before the loop.

Without the second pragma each thread will execute all iterations.

Note that it’s the programmer’s responsibility to check that there are no data dependences between loop iterations.
For Loop Scheduling

There are three ways to schedule for-loops:

- **schedule(static)**
  - The iterations are assigned statically in contiguous blocks of iterations.
  - Static scheduling has the least overhead, obviously, but may suffer from poor load imbalance, e.g. in an LU-decomposition.

- **schedule(dynamic) or schedule(dynamic, size)**
  - The default size is one iteration.
  - A thread is assigned size contiguous iterations at a time.

- **schedule(guided) or schedule(guided, size)**
  - The default size is one iteration.
  - With a size, a thread never (except possibly at the end) is assigned fewer than size contiguous iterations at a time.
  - The number of iterations assigned to a thread is proportional to the number of unassigned iterations and the number of threads.

In addition, runtime can be specified which uses the environment variable `OMP_SCHEDULE` which must be one of above three but without the size.
An Example

```c
#include <omp.h>
#include <stdio.h>

#define N (1024)
float a[N][N];
float b[N][N];
float c[N][N];

int main(void)
{
    int i;
    int j;
    int k;

    #pragma omp parallel private(i,j,k)
    #pragma omp for schedule(static, N/omp_get_num_procs())
    for (i = 0; i < N; ++i)
        for (k = 0; k < N; ++k)
            for (j = 0; j < N; ++j)
                a[i][j] += b[i][k] * c[k][j];

    return 0;
}
```

- We need private i, j and k since they are declared before the pragma.
- If a function is called in a parallel region, all its local variables become private.
Each section is executed in parallel.
By a **reduction** is meant computing a scalar value from an array such as a sum.

The loop has a data dependence on the `sum` variable.

How can we parallelize it anyway?

```c
float a[N];
float sum;
int i;

for (sum = i = 0; i < N; ++i)
    sum += a[i];
```
By introducing a sum variable private to each thread, and letting each thread compute a partial sum, we can parallelize the reduction:

```c
float a[N];
float sum;
int i;

#pragma omp parallel
#pragma omp for
#pragma omp reduction(+:sum)
for (sum = i = 0; i < N; ++i)
    sum += a[i];
```

We can write the pragmas on one line if we wish:

```c
#pragma omp parallel for reduction(+:sum)
for (sum = i = 0; i < N; ++i)
    sum += a[i];
```

There are reductions for: \(+\) \(-\) \(*\) \& \(|\) \&\& \(||\) with suitable start values such as 1 for \(*\) and \(~0\) for \&\&.
Critical Sections

A critical sections is created as in:

```c
#pragma omp critical
{
    point->x += dx;
    point->y += dy;
}
```
Atomic Update

When one variable should be updated atomically, we can use:

```c
#pragma omp atomic
count += 1;
```
Explicit Barriers

- Recall there is an implicit barrier at the end of a parallel region.
- To create a barrier explicitly, we can use:

```c
#pragma omp barrier
```
Work for One Thread

- Recall only the master executes the sequential code between parallel regions.
- If we wish only the master should execute some code in a parallel region, we can use
  
  \#pragma omp master

- If it doesn’t matter which thread performs the work, we can instead use
  
  \#pragma omp single

- There is a difference between the two above constructs: an implicit barrier is created after a single directive.
Locks

- OpenMP supports two kinds of locks: plain locks and recursive locks.
- Recall a thread can lock a recursive lock it already owns without blocking forever.
- Recursive locks are called nested locks in OpenMP.
- The lock functions are `omp_init_lock`, `omp_set_lock`, `omp_unset_lock`, `omp_test_lock`, and `omp_destroy_lock`, and `omp_nest_init_lock`, `omp_nest_set_lock`, `omp_nest_unset_lock`, `omp_nest_test_lock`, and `omp_nest_destroy_lock`.
Weak ordering is the consistency model for OpenMP.

The required synchronization instructions are inserted implicitly with the above introduced directives.

A for loop can be created without an implicit barrier using `nowait` and in that case `#pragma omp flush` makes caches consistent.

A list of variables to write back can be specified:

```plaintext
#pragma omp flush(a,b,c)
```
The following compilers support OpenMP

- GNU
- Clang
- IBM XL
- Oracle
- Intel
- Portland Group
- Pathscale
- Absoft
- Fujitsu
- Microsoft
- HP
- Cray
• Simple example
• Object ownership for single-threaded programs
• Message passing
• Threads
• Shared memory objects in multi-threaded programs
fn main()
{
    println!("hello, world");
}

- Save in a.rs and type rustc a.rs && ./a
- Or in src/main.rs and type cargo run
- In the latter case you need a file Cargo.toml with some definitions
fn main()
{
    let s = String::from("world");
    println!("hello, {} {}", s, "again");
}

- Creates a string from the heap (Java new and C malloc)
- {} takes the next parameter
- Output is hello, world again
- The memory for an object can be deallocated with the function drop
- The drop function is called automatically when reaching the }
```java
class a {
    public static void main(String[] args) {
        String s = new String("hello, world");
        String t = s;
        System.out.println(t);
    }
}
```

- Only one string object
- Garbage collection takes care of the memory for the string object, of course
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
    char* s;
    int n;

    n = 1 + strlen("hello");
    s = malloc(n);
    strcpy(s, "hello");
    printf("%s\n", s);
    free(s);
}
```
int main()
{
    char* s;
    char* t;
    int n;

    n = 1 + strlen("hello");
    s = malloc(n);
    strcpy(s, "hello");

    t = s;
    printf("%s\n", t);
    free(s);
    free(t); // disaster will follow
}
Rust heap objects

- Java’s garbage collection can be slow
- It can be bad if it occurs at the wrong moment (e.g. landing airplane, or updating big database)
- The C explicit allocation and deallocation is fine if you are careful
- Rust has no garbage collection like Java but strict rules about how pointers can be used
- A purpose with Rust is to be fast systems programming language without the headaches of C (their interpretation)
- Or, (my interpretation) a new Gulag without the freedom of C
- It is interesting to study, though, and has many nice ideas
Rust upholds the safety and zero-cost claims. Using Rust has been found to aid in achieving an improved, shorter, more expressive architecture. The learning curve is a bit steep, but productivity has been found to be high once learned. Tooling support is mature, but IDEs are not yet full featured.
fn main()
{
    let s = String::from("world");
    let t = s;

    println!("hello, {}", t);
}

- Similar to the Java program and no complaints
- The variable s becomes useless however
- The string object has been **moved** to t which owns it from the =
- Only one owner at a time!
fn main() {
    let s = String::from("world");
    let t = s;
    println!("hello, {}", s);
}

error[E0382]: borrow of moved value: 's'
    --> a.rs:5:31
     3 | let s = String::from("world");
        | - move occurs because 's' has type 'std::string::String',
        |    which does not implement the 'Copy' trait
     4 | let t = s;
        | - value moved here
     5 | println!("hello, {}", s);
        | - value borrowed here after ^ move
```rust
fn main()
{
    let s = String::from("world");
    let t = s.clone();
    println!("hello, {}", s);
}
```

This works but gets complaint about unused variable t
fn f(t: String) { }

fn main()
{
    let s = String::from("world");
    f(s);
    println!("hello, {s}");
}

Also invalid since the call also moves the string
fn f(s: String) -> String
{
    s
}

fn main()
{
    let s = String::from("world");
    s = f(s);
    println!("hello, {}", s);
}

- Ownership can be returned from a function
- Still wrong though: cannot assign to s twice (but see mut below)
fn f(s: String) -> String
{
    s
}

fn main()
{
    let s = String::from("world");
    let t = f(s);
    println!("hello, {}", t);
}

- Now correct
- But we may want to modify s instead of introducing t
fn f(s: &String, t: &String, u: &String) {
}

fn main() {
    let s = String::from("world");
    f(&s, &s, &s);
    println!("hello, {}", s);
}

- Safe to give away references to s (even multiple)
- Keep ownership using &
- f is not allowed to modify s
fn f(s: &mut String)
{
    s.push_str(" with some added text");
}

fn main()
{
    let mut s = String::from("world");
    f(&mut s);
    println!("hello, {}", s);
}

- Declare mutable to allow modification
- Only one reference can borrow an object at a time when mutable
use std::thread;

fn main() {
    let h = thread::spawn(|| {
        println!("thread");
    });

    h.join().unwrap();
    println!("main");
}

- Create a thread with spawn
- Wait for it with join
- `unwrap` means controlled exit if something is wrong
```rust
use std::thread;
use std::sync::mpsc; // multi-producer single-consumer

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
    });

    let received = rx.recv().unwrap();
    println!("got {}", received);
}
```

- The send moves `val` from sender to receiver
- Note the move near `spawn` — see next slide