Multiprocessor Parallelism
ASD Shared Memory HPC Workshop

June 2023

School of Computing
Australian National University
Canberra, Australia

Schedule - Day 3 - Multiprocessor Parallelism

Time	Lecture Topics	Hands-On Exercise
9:00	OS thread support Mutual exclusion	Non-determinism
10:30	*Morning Tea*	
11:00	Higher-level synchronization constructs Memory consistency	Implementing synchronization constructs / Deadlock
12:30	*Lunch*	
13:30	OpenMP	OpenMP
15:00	*Afternoon Tea*	
15:30	OpenMP Tasks	OpenMP Tasks

Reference Material

- *Introduction to Parallel Computing*, A Grama, A Gupta, G Karypis, and V Kumar, Addison Wesley 2003 (ISBN 0 201 64865 2)
- *Parallel Computer Architecture and Programming*, Kayvon Fatahalian, Carnegie Mellon University Course 15-418/618 (http://15418.courses.cs.cmu.edu/spring2015/)
- *Concurrency: State Models and Java Programming*, J. Magee and J. Kramer, 2nd edn, Wiley, 2006 (ISBN-10: 0470093552)
- *The C++ Programming Language, 4th Edition*, Bjarne Stroustrup, Pearson Education 2013 (ISBN 978-0-321-56384-2)
- *Shared Memory Application Programming: Concepts and Strategies in Multicore Application Programming 1st Edition*, Victor Alessandrini, Morgan Kaufmann, 2015 (ISBN 978-0128037614)

Outline

OS Support for Threads

Mutual Exclusion

Synchronization Constructs

Memory Consistency

The OpenMP Programming Model

Concurrent vs Parallel Processing

- Concurrent Processing: Environment in which tasks are defined and allowed to execute in any order
  - Does not imply a multiple processor environment
  - E.g. spawn a separate thread to do I/O while CPU intensive task continues to do another operation
- Parallel Processing: The simultaneous execution of concurrent tasks on different CPUs

All parallel processing is concurrent, but NOT all concurrent processing is parallel

O/S Thread Support

- O/S's were originally designed for process not thread support
- Require the O/S to
  - Treat threads equally and ensure that they all get equal (or user defined) access to machine resources
  - Know what to do when a thread issues a fork command
  - Be able to deliver a signal to the correct thread
- Libraries executed by a multi-threaded program need to be thread-safe
  - Potential static data structures to be overwritten
- Hardware support
  - Some hardware provides support for very fast context switches between threads, e.g. Cray MTA or more recently hyper-threading

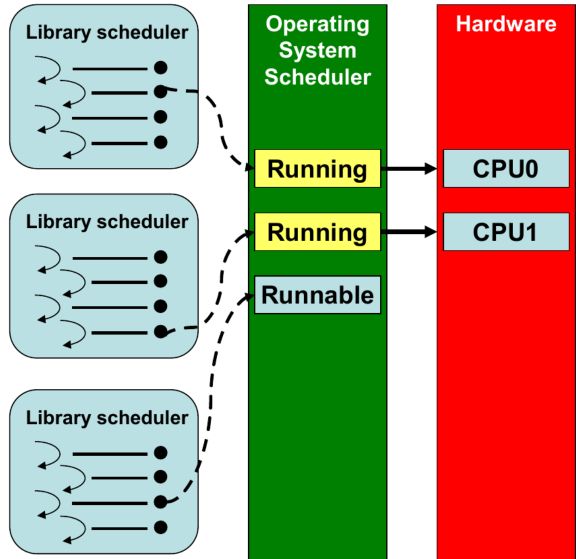
Thread Implementations

- Three categories
  - Pure user space
  - Pure kernel space
  - Mixed
- User Mode: When a process executes instructions within its program or linked libraries
- Kernel Mode: When the operating system executes instructions on behalf of a user program
  - Often as a result of a system call
  - In kernel mode, the program can access kernel space

User Space Threads

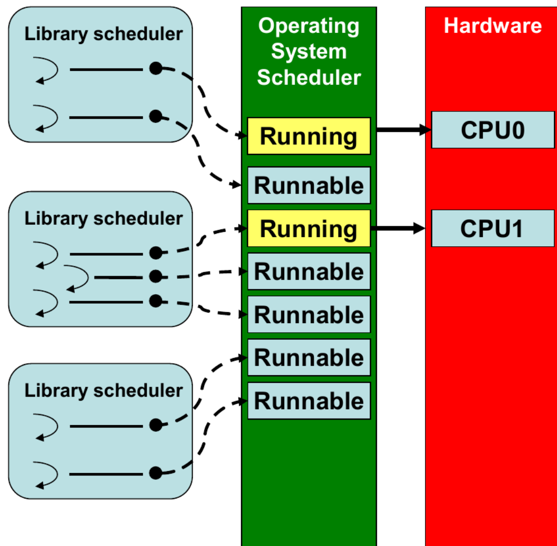
- No kernel involvement in providing a thread
- Kernel has no knowledge of threads and continues to schedule processes only
- The thread library selects which thread to run

OK for concurrency, no good for parallel programming



Kernel Space Threads

- A kernel-level thread is created for each user thread
- O/S must manage on a per-thread basis information typically managed on a process basis
- Potentially poor scaling when too many threads as O/S gets overloaded



User-space v Kernel-space

- User Space Advantages
  - No changes to core O/S
  - No kernel involvement means they may be faster for some applications
  - Can create many threads without overloading the system
- User Space Disadvantages
  - Potentially long latency during system service blocking (e.g. 1 thread stalled on I/O request)
  - All threads compete for the same CPU cycles
  - No advantage gained from multiple CPUs
- Mixed scheme
  - A few user threads are mapped to one kernel thread

Outline

OS Support for Threads

Mutual Exclusion

Synchronization Constructs

Memory Consistency

The OpenMP Programming Model

Accessing Shared Data

- Concurrent read is no problem
  - but what about concurrent write?

Code	P0	P1
x=x+1	ld r0, addr(x) add, r1,r0,#1 st addr(x), r1	ld s0, addr(x) add, s1,s0,#1 st addr(x), s1

Time ↓

- Result could be x+1 or x+2
- Similar problems with access to shared resources like I/O
- Critical Sections: a mechanism to ensure controlled access to shared resources
  - Processes enter the critical section under mutual exclusion

Locks

- Simplest mechanism for providing mutual exclusion
- A lock is a 1-bit variable, a value of
  - 1 indicates a process is in the critical section
  - 0 indicates no process is in the critical section
- At its lowest level a lock is a protocol for coordinating processes,
  - The CPU is not physically prevented from executing those instructions

```
1  while (lock == 1) do_nothing;      /*busy-wait loop */
   lock = 1;                          /*enter critical section*/
3  critical section
   lock = 0;                          /*exit critical section*/
```

- The above is an incorrect example of a spin-lock, that uses a mechanism called busy-waiting

Spin-Lock Assembly

```
lock:        ld    R0, mem[addr]     ; load word into R0
             cmp   R0, #0            ; if 0, store 1
             bnz   lock              ; else, try again
             st    mem[addr], #1

unlock:      st    mem[addr], #0     ; store 0 to address
```

Problem: data race because LOAD-TEST-STORE is not atomic!

- Processor 0 loads address X, observes 0
- Processor 1 loads address X, observes 0
- Processor 0 writes 1 to address X
- Processor 1 writes 1 to address X

(See http://15418.courses.cs.cmu.edu/spring2015/lecture/synchronization)

Test and Set Lock

- Test-and-set instruction:

```
ts    R0, mem[addr]          ; atomically load mem[addr] into R0
                             ; if mem[addr] is 0 then set mem[addr] to 1
```

```
lock:     ts   R0, mem[addr]       ; load word into R0
          cmp  R0, #0              ; if 0, lock obtained
          bnz  lock

unlock:   st mem[addr], #0         ; store 0 to address
```

  What does 'atomic' mean in this context?

- p threads contending for the lock will require $O(p)$ attempts, thus $O(p^2)$ time (and energy!)
- we can reduce the number of attempts (but still $O(p)$) by only trying the `ts` when we see `mem[addr]` is 0

Common Atomic Operations

- Test and Set

```
bool TestAndSet(bool *lock) {
  bool initial = *lock;
  lock = true;
  return initial;
}
```

- Fetch and Add (operate)

```
int FetchAndAdd(int *location, int inc) {
  int value = *location;
  *location = value + inc;
  return value;
}
```

- Compare and Swap (Intel, CMPXCHG)

```
bool cas(int *p, old: int, new: int) {
  if *p != old
    return false;
  *p = new;
  return true;
}
```

Locks: Phases and Properties

- Phases
  - Waiting to acquire lock (busy wait or de-schedule and woken later)
  - Lock acquisition
  - Releasing lock
- Properties
  - Fast to acquire in the absence of contention
  - Low interconnect traffic
  - Scalability
  - Low storage cost
  - Fairness
  - Does not degrade the overall system (e.g. when more threads than CPUs contend for a lock)

Pthread Lock Routines

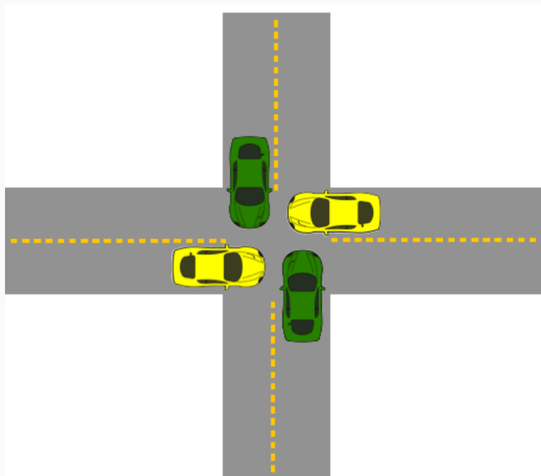
- Locks are implemented as mutually exclusive lock variables or mutex variables
- To use a variable, it must be declared as type `pthread_mutex_t`
  - Usually in the main thread as it needs to be visible to all threads using it

```
pthread_mutex_t mutex1;
pthread_mutex_init(&mutex1, NULL); //(NULL specifies default attributes for
    mutex)
...
pthread_mutex_lock(&mutex1);
/* CRITICAL SECTION */
pthread_mutex_unlock(&mutex1);
```

Synchronization Terminology

- Deadlock
- Livelock
- Starvation

Deadlock

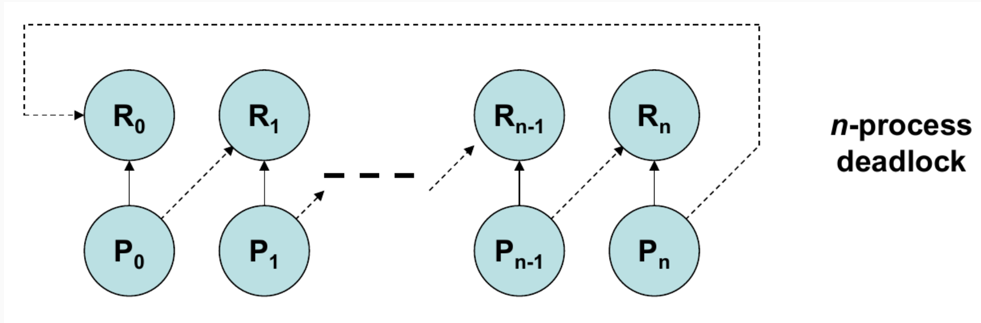


http://15418.courses.cs.cmu.edu/spring2015/lecture/snoopimpl

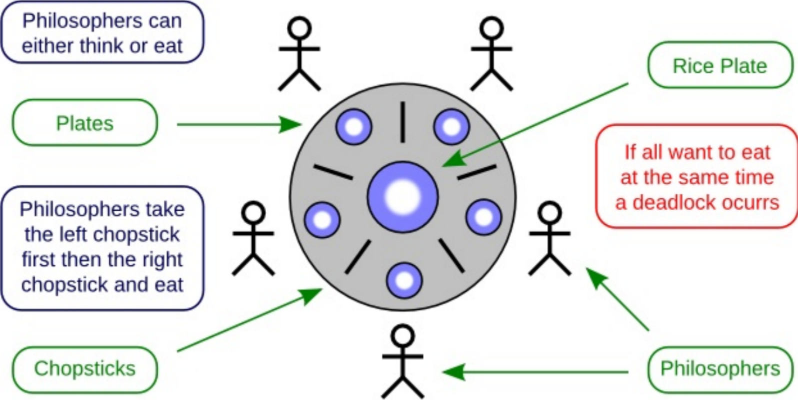
- Deadlock is a state where a system has outstanding operations to complete, but no operation can make progress
- Can arise when each operation has acquired a shared resource that another operation needs
- In a deadlock situation, there is no way for any thread (or, in this illustration, a car) to make progress unless some thread relinquishes a resource ("backs off")

Deadlock

- Can occur with multiple processes seeking to acquire resources that cannot be shared



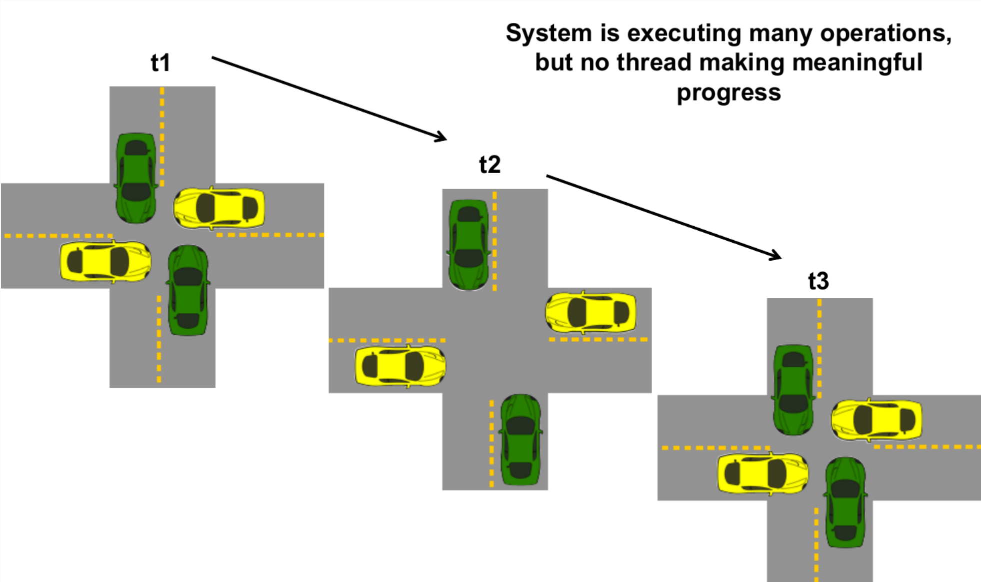
- Pthreads provides pthread_mutex_trylock() to help address these circumstances


The Dining Philosophers Problem (Deadlock)
Philosophers can either think or eat
Plates
Philosophers take the left chopstick first then the right chopstick and eat
Chopsticks
Rice Plate
If all want to eat at the same time a deadlock ocurrs
Philosophers
EE4007: Distributed Software Systems
M. Eberhard

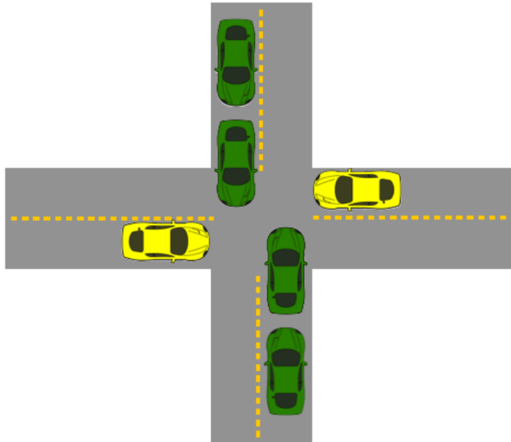
Deadlock Conditions

- Mutual exclusion: the resources involved must be unsharable
- Hold and wait: processor must hold the resources they have been allocated while waiting for other resources needed to complete an operation
- No pre-emption: processes cannot have resources taken away from them until they have completed the operation they wish to perform
- Circular wait: exists in the resource dependency graph

Livelock



Starvation



http://15418.courses.cs.cmu.edu/spring2015/lecture/snoopimpl

- A state where the system is making overall progress, but some processes make no progress (green cars make progress, but yellow cars are stopped)
- Starvation is usually not a permanent state (as soon as green cars pass, yellow cars can go)

In this example: assume traffic moving left/right (yellow cars) must yield to traffic moving up/down (green cars)

Follow-On Issues

- Lots of stuff on specific barrier implementation
- Lock free data structures
- Transactional memory

Outline

OS Support for Threads

Mutual Exclusion

Synchronization Constructs

Memory Consistency

The OpenMP Programming Model

Semaphores: Concepts

- **Semaphores** are widely used for dealing with inter-process synchronization in operating systems
- A **semaphore** `s` is an integer variable that can take only non-negative values
- The only operations permitted on `s` are `up(s)` and `down(s)`
- `down(s)`:

```
    if (s > 0)
        decrement s
    else
        block execution of the calling process
```

- `up(s)`:

```
    if there are processes blocked on s
        awaken one of them
    else
        increment s
```

- Blocked processes or threads are held in a FIFO queue

Semaphores: Implementation

- Binary semaphores have only a value of 0 or 1
  - Hence can act as a lock
- The waking up of threads via an `up(s)` occurs via an OS signal
- Posix semaphore API

```c
#include <semaphore.h>
...
int sem_init      (sem_t *sem, int pshared, unsigned int value);
int sem_destroy   (sem_t *sem);
int sem_wait      (sem_t *sem);  //down()
int sem_trywait   (sem_t *sem);
int sem_timedwait (sem_t *sem, const struct timespec *abstime);
int sem_post      (sem_t *sem);  //up()
int sem_getvalue  (sem_t *sem, int *value);
```

- `pshared`: if non-0, generate a **semaphore** for usage between processes
- `value` delivers the number of waiting processes/threads as a negative integer, if there are any waiting on this **semaphore**

Monitors and Condition Synchronization

Concepts: **monitors**:

- encapsulated data + access function
- **mutual exclusion + condition synchronization**
- only a single access function can be active in the **monitor**

Practice:

- private shared data and **synchronized functions** (exclusion)
  - the latter requires a per object lock to be acquired in each function
  - only a single thread may be active in the monitor at a time
- **condition synchronization**: achieved using a **condition variable**
  - `wait()`, `notify_one()` and `notify_all()`

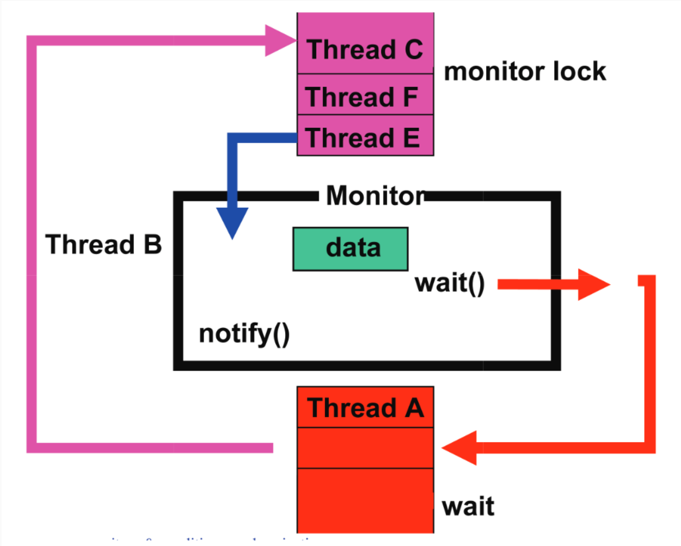
Note: monitors serialize all accesses to the encapsulated data!

Monitors in ‘C’ / POSIX (types and creation)

```c
int pthread_mutex_lock      (pthread_mutex_t *mutex);
int pthread_mutex_trylock   (pthread_mutex_t *mutex);
int pthread_mutex_timedlock (pthread_mutex_t *mutex,
                             const struct timespec*abstime);
int pthread_mutex_unlock    (pthread_mutex_t *mutex);
int pthread_cond_wait       (pthread_cond_t *cond,
                             pthread_mutex_t *mutex);
int pthread_cond_timedwait  (pthread_cond_t *cond,
                             pthread_mutex_t *mutex,
                             const struct timespec*abstime);
int pthread_cond_signal     (pthread_cond_t *cond);
int pthread_cond_broadcast  (pthread_cond_t *cond);
```

- `pthread_cond_signal()` unblocks at least one thread
- `pthread_cond_broadcast()` unblocks all threads waiting on `cond`
- `lock` calls can be called anytime (multiple lock activations are possible)
- the API is flexible and universal, but relies on conventions rather than compilers

Condition Synchronization



Condition Synchronization in Posix

Synchronization between POSIX-threads:

```
typedef ... pthread_mutex_t;
typedef ... pthread_mutexattr_t;
typedef ... pthread_cond_t;
typedef ... pthread_condattr_t;
int pthread_mutex_init(pthread_mutex_t *mutex,
       const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_cond_init(pthread_cond_t *cond,
                      const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);
```

Mutex and condition variable attributes include:

- semantics for trying to lock a mutex which is locked already by the same thread
- `pthread_mutex_destroy()`: undefined, if lock is in use
- `pthread_cond_destroy()` undefined, if there are threads waiting

Bounded Buffer in Posix

```c
typedef struct bbuf_t {
  int count, N; ...
  pthread_mutex_t bufLock;
  pthread_cond_t notFull;
  pthread_cond_t notEmpty;
} bbuf_t;
int getBB(bbuf_t *b) {
  pthread_mutex_lock(&b->bufLock);
  while (b->count == 0)
    pthread_cond_wait(
      &b->notEmpty,
      &b->bufLock);
  b->count--;
  int v; // remove item from buffer
  pthread_cond_signal(&b->notFull);
  pthread_mutex_unlock(&b->bufLock);
  return v;
}
```

```c
void putBB(int v, bbuf_t *b) {
  pthread_mutex_lock(&b->bufLock);
  while (b->count == b->N) {
    pthread_cond_wait(
      &b->notFull,
      &b->bufLock);
  }
  b->count++;
  // add v to buffer
  pthread_cond_signal(&b->notEmpty);
  pthread_mutex_unlock(&b->bufLock);
}
```

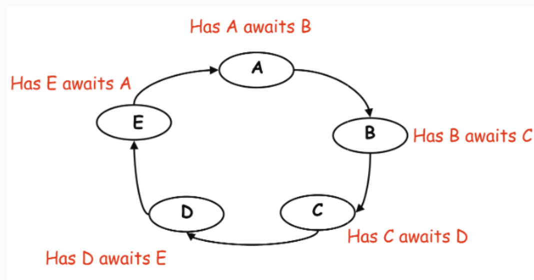
Deadlock and its Avoidance

- Ideas for deadlock:
  - concepts: system deadlock: no further progress
  - model: deadlock: no eligible actions
  - practice: blocked threads
- Our aim: **deadlock avoidance**: to design systems where deadlock cannot occur
- Done by removing one of the **four necessary and sufficient conditions**:
  - **serially reusable resources**: the processes involved share resources which they use under mutual exclusion
  - **incremental acquisition**: processes hold on to resources already allocated to them while waiting to acquire additional resources
  - **no pre-emption**: once acquired by a process, resources cannot be pre-empted (forcibly withdrawn) but are only released voluntarily.
  - **wait-for cycle**: a circular chain (or cycle) of processes exists such that each process holds a resource which its successor in the cycle is waiting to acquire

Wait-for Cycles

- operating systems must deal with deadlock arising from processes requesting resources (e.g. printers, memory, co-processors)

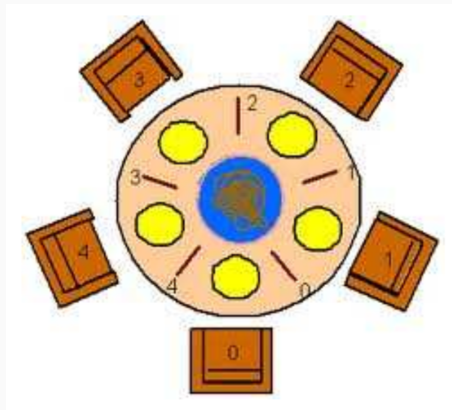
- **pre-emption** involves constructing a **resource allocation graph**, detecting a cycle and removal of a resource
- **avoidance** involves never granting a request that could lead to such a cycle (the Banker's Algorithm)



(courtesy Magee & Kramer: Concurrency)

The Dining Philosophers Problem

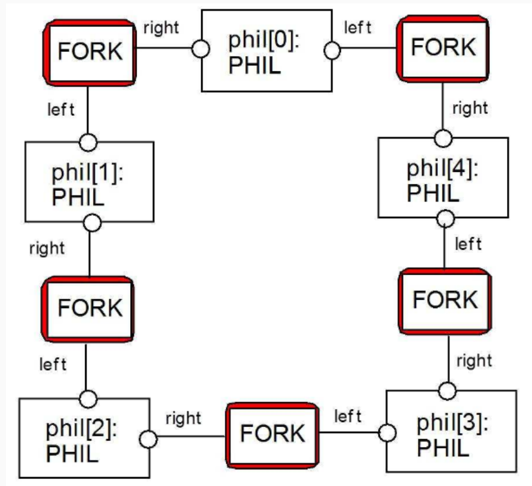
Five philosophers sit around a circular table. Each philosopher spends his life alternately **thinking** and **eating**.
In the centre of the table is a large bowl of spaghetti. A philosopher needs two forks to eat a helping of spaghetti. One fork is placed between each pair of philosophers and they agree that each will only use the fork to his immediate right and left.
If all 5 sit down at once and take the fork on his left, there will be **deadlock**: a **wait-for cycle** exists!



(courtesy Magee & Kramer: Concurrency)

Dining Philosophers: Model Structure Diagram

Each `FORK` is a shared resource with actions `get` and `put`.
When hungry, each `Phil` must first get his right and left forks before he can start eating.
Each `Fork` is either in a taken or not taken state.
This is an example when the resulting **monitor state** can be distributed (resulting in greater system concurrency).



(courtesy Magee & Kramer: Concurrency)

Hands-on Exercise: Non-Determinism

Objective:

- To observe the effect of non-determinism in the form of race conditions on program behaviour and how it can be avoided

Outline

OS Support for Threads

Mutual Exclusion

Synchronization Constructs

Memory Consistency

The OpenMP Programming Model

Consider the following

```
                initially flag1 = flag2 = 0

                              |
    //Process 0               |          //Process 1
                              |
                              |
    flag1 = 1                 |          flag2 = 1
    if (flag2 == 0)           |          if (flag1 == 0)
      print "Hello";          |            print "World";
                              |
```

What is printed?

Time Ordered Events

```
    Process 0                                         Process 1
--------------------------------------------------------------------------------
1  flag1 = 1
2  if (flag2 == 0) print "Hello";
3                                                   flag2 = 1
4                                                   if (flag1 == 0) print "World";

                              Output: Hello
--------------------------------------------------------------------------------
1  flag1 = 1
2                                                   flag2 = 1
3  if (flag2 == 0) print "Hello";
4                                                   if (flag1 == 0) print "World";

                          Output: (Nothing Printed)
--------------------------------------------------------------------------------
1                                                   flag2 = 1
2                                                   if (flag1 == 0) print "World";
3  flag1 = 1
4  if (flag2 == 0) print "Hello";

                              Output: World
--------------------------------------------------------------------------------
```

- Never Hello and World?
    - But what fundamental assumption are we making?

Memory Consistency

- *To write correct and efficient shared memory programs, programmers need a precise notion of how memory behaves with respect to read and write operations from multiple processors* (Adve and Gharachorloo)
- Memory/cache coherency defines requirements for the observed behaviour of reads and writes to the same memory location (ensuring all processors have consistent view of same address)
- Memory consistency defines the behavior of reads and writes to different locations (as observed by other processors)

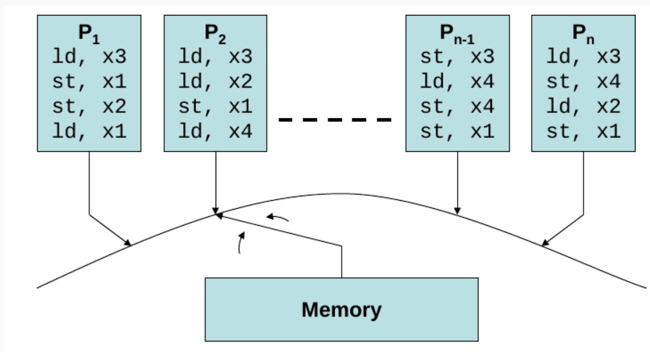
(See: Sarita V. Adve and Kourosh Gharachorloo. 1996. Shared Memory Consistency Models: A Tutorial. Computer 29, 12, 66-76.

DOI=10.1109/2.546611)

Sequential Consistency

- Lamport's definition: [A multiprocessor system is sequentially consistent if] the result of any execution is the same as if the operation of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program
- Two aspects:
  - Maintaining program order among operations from individual processors
  - Maintaining a single sequential order among operations from all processors
- The latter aspect make it appear as if a memory operation executes atomically or instantaneously with respect to other memory operations

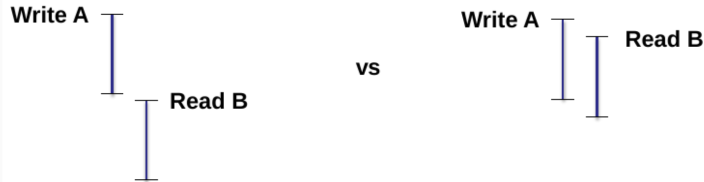
Programmer's View of Sequential Consistency



- Conceptually:
  - There is a single global memory and a switch that connects an arbitrary processor to memory at any time step
  - Each process issues memory operations in program order and the switch provides the global serialization among all memory operations

Motivation for Relaxed Consistency

- To gain performance
  - Hide the latency of independent memory accesses with other operations
  - Recall that memory accesses to a cache coherent system may involve much work



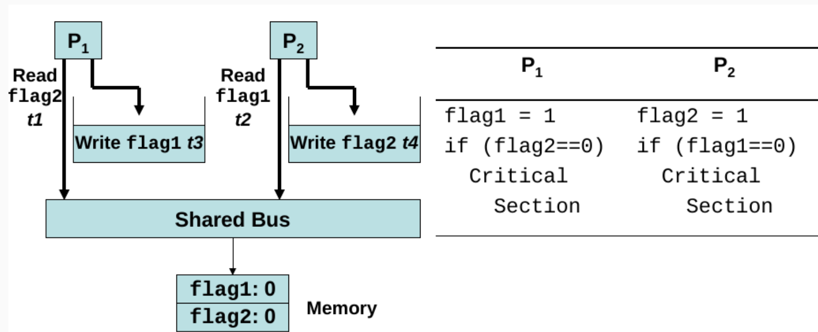
- Can relax either the program order or atomicity requirements (or both)
  - e.g. relaxing write to read and write to write allows writes to different locations to be pipelined or overlapped
  - Relaxing write atomicity allows a read to return another processor's write before all cached copies have been updated

Possible Reorderings

- We consider first relaxing the program order requirements
- Four types of memory orderings (within a program):
  - W→R: write must complete before subsequent read (RAW)
  - R→R: read must complete before subsequent read (RAR)
  - R→W: read must complete before subsequent write (WAR)
  - W→W: write must complete before subsequent write (WAW)

  Normally, *different* addresses are involved in the pair of operations
- Relaxing these can give:
  - W→R: e.g. the write buffer (aka store buffer)
  - everything: the weak and release consistency models

Breaking W → R Ordering: Write Buffers



(Adve and Gharachorloo DOI=10.1109/2.546611)

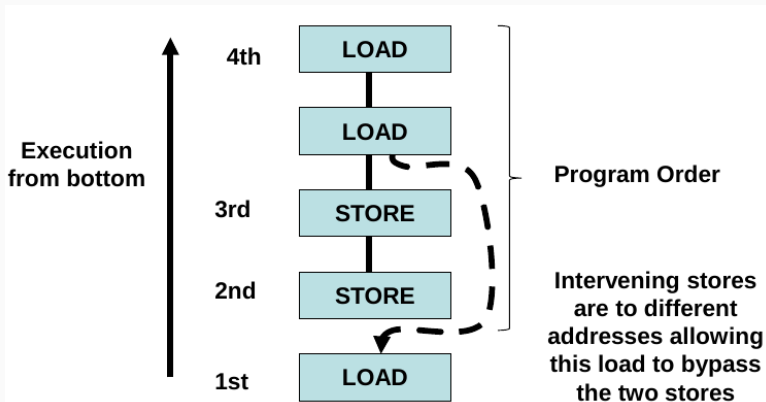
- Write buffer (very common)
  - Process inserts writes in write buffer and proceeds, assuming it completes in due course
  - Subsequent reads bypass previous writes, using the value in the write buffer
  - Subsequent writes are processed by the buffer in order (no W→W relaxations)

Allowing reads to move ahead of writes

- Total Store Ordering (TSO)
  - Processor P can read B before its write to A is seen by all processors (process can move its own reads in front of its own writes)
  - Reads by other processors cannot return the new value of A until the write to A is observed by all processors
- Processor consistency (PC)
  - Any processor can read new value of A before the write is observed by all processors
- In TSO and PC, only the W $\rightarrow$ R order is relaxed. The W $\rightarrow$ W constraint still exists. Writes by the same thread are not reordered (they occur in program order)

See http://15418.courses.cs.cmu.edu/spring2015/lecture/consistency

Processor Consistency



- Before a LOAD is allowed to perform wrt. any processor, all previous LOAD accesses must be performed wrt. everyone
- Before a STORE is allowed to perform wrt. any processor, all previous LOAD *and* STORE accesses must be performed wrt. everyone

Four Example Programs

Assume A and B are initialized to 0
Assume prints are loads

1

Thread 1 (on P1)
```
A = 1;
flag = 1;
```
Thread 2 (on P2)
```
while (flag == 0);
print A;
```

2

Thread 1 (on P1)
```
A = 1;
B = 1;
```
Thread 2 (on P2
```
print B;
print A;
```

3

Thread 1 (on P1)
```
A = 1;
```
Thread 2 (on P2)
```
while (A == 0);
B = 1;
```
Thread 3 (on P3)
```
while (B == 0);
print A;
```

4

Thread 1 (on P1)
```
A = 1;
print B;
```
Thread 2 (on P2)
```
B = 1;
print A;
```

Do (all possible) results of execution match that of sequential consistency (SC)?

	1	2	3	4
Total Store Ordering (TSO)	✓	✓	✓	✗
Processor Consistency (PC)	✓	✓	✗	✗

See http://15418.courses.cs.cmu.edu/spring2015/lecture/consistency

Clarification

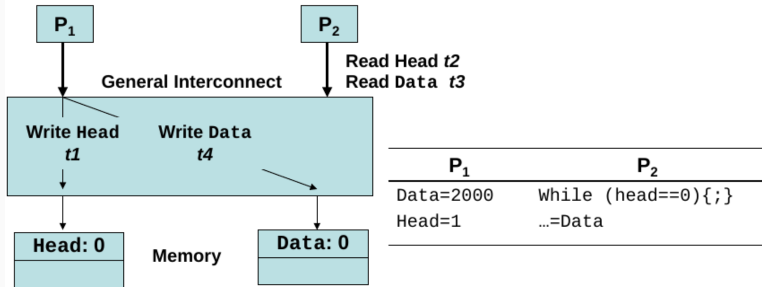
- The cache coherency problem exists because of the optimization of duplicating data in multiple processor caches. The copies of the data must be kept coherent.
- Relaxed memory consistency issues arise from the optimization of reordering memory operations (this is unrelated to whether there are caches in the system).

Allowing writes to be reordered

- Four types of memory operations orderings
  - ~~W→R : write must complete before subsequent read~~
  - R→R: read must complete before subsequent read
  - R→W: read must complete before subsequent write
  - ~~W→W: write must complete before subsequent write~~
- Partial Store Ordering (PSO)
  - Execution may not match sequential consistency on program 1 (P2 may observe change to flag before change to A)

```
Thread 1 (on P1)                Thread 2 (on P2)
-----------------------------------------------------------
A = 1;                          while (flag == 0);
flag = 1;                       print A;
```

Breaking W → W Ordering: Overlapped Writes



(Adve and Gharachorloo DOI=10.1109/2.546611)

General (non-bus) Interconnect with multiple memory modules

- Different memory ops issued by same processor serviced by different memory modules
- Writes from P1 injected into memory system in program order: may complete out of order
- Multiple processors coalesced write to same cache line in write buffer: could lead to similar effects

Allowing all reorderings

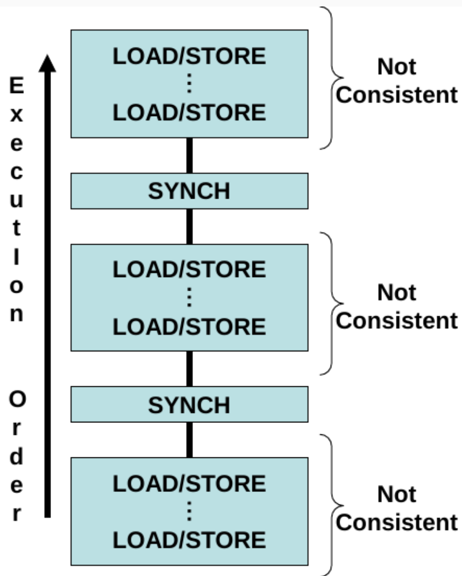
- Four types of memory operations orderings
  - ~~W→R: write must complete before subsequent read~~
  - ~~R→R: read must complete before subsequent read~~
  - ~~R→W: read must complete before subsequent write~~
  - ~~W→W: write must complete before subsequent write~~
- Examples:
  - Weak Ordering (WO)
  - Release Consistency (RC)
    - Processors support special synchronization operations
    - Memory accesses before memory fence instruction must complete before the fence issues
    - Memory accesses after fence cannot begin until fence instruction is complete

```
reorderable reads
and writes here
...
<MEMORY FENCE>
...
reorderable reads
and writes here
....
<MEMORY FENCE>
```

Weak Consistency

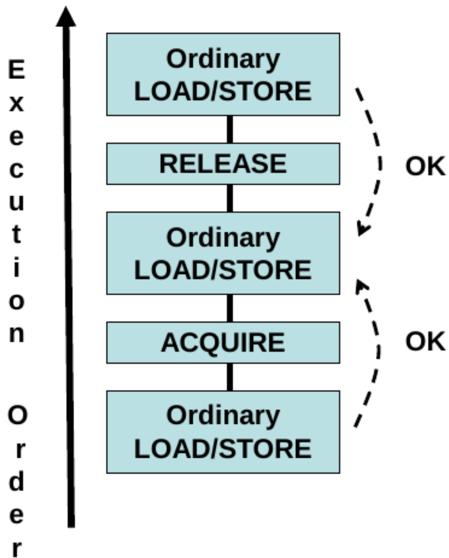
- Relies on the programmer having used critical sections to control access to shared variables
  - Within the critical section no other process can rely on that data structure being consistent until the critical section is exited
- We need to distinguish critical points when the programmer enters or leaves a critical section
  - Distinguish standard load/stores from synchronization accesses

Weak Consistency



- Before an ordinary LOAD/STORE is allowed to perform wrt. any processor, all previous SYNCH accesses must be performed wrt. everyone
- Before a SYNCH access is allowed to perform wrt. any processor, all previous ordinary LOAD/STORE accesses must be performed wrt. everyone
- SYNCH accesses are sequentially consistent wrt. one another

Release Consistency



- Before any ordinary LOAD/STORE is allowed to perform wrt. any processor, all previous ACQUIRE accesses must be performed wrt. everyone
- Before any RELEASE access is allowed to perform wrt. any processor, all previous ordinary LOAD/STORE accesses must be performed wrt. everyone
- Acquire/Release accesses are processor consistent wrt. one another

Enforcing Consistency

- The hardware provides underlying instructions that are used to enforce consistency
  - Fence or Memory Bar instructions
- Different processors provide different types of fence instructions

Example: Synchronization in relaxed models

- Intel x86/x64 - total store ordering
  - Provides sync instructions if software requires a specific instruction ordering not guaranteed by the consistency model
    - `mm_lfence`("load fence": waits for all loads to complete)
    - `mm_sfence`("store fence": waits for all stores to complete)
    - `mm_mfence`("mem fence": waits for all mem operations to complete)
- ARM processors: very relaxed consistency model

Summary: relaxed consistency

- Motivation: obtain higher performance by allowing recording of memory operations for latency hiding (reordering is not allowed by sequential consistency)
- One cost is software complexity: programmer or compiler must correctly insert synchronization to ensure certain specific ordering
  - But in practice complexities encapsulated in libraries that provide intuitive primitives like lock/unlock, barrier (or lower level primitives like fence)
  - Optimize for the common case: most memory accesses are not conflicting, so don't pay the cost as if they are
- Relaxed consistency models differ in which memory ordering constraints they ignore

Final Thoughts

- What consistency model best describes pthreads?

Hands-on Exercises: and Deadlock

Objectives:

- to understand POSIX thread creation
- to observe race conditions
- to understand the use and implementation of bounded buffer monitors

Outline

OS Support for Threads

Mutual Exclusion

Synchronization Constructs

Memory Consistency

The OpenMP Programming Model

OpenMP Reference Material

- https://www.openmp.org/
- *Introduction to High Performance Computing for Scientists and Engineers*, Hager and Wellein, Chapter 6 & 7
- *High Performance Computing*, Dowd and Severance, Chapter 11
- *Introduction to Parallel Computing, 2nd Ed*, A. Grama, A. Gupta, G. Karypis, V. Kumar
- *Parallel Programming in OpenMP*, R. Chandra, L.Dagum, D.Kohr, D.Maydan. J.McDonald, R.Menon

Shared Memory Parallel Programming

- Explicit thread programming is messy
  - low-level primitives
  - originally non-standard, although better since pthreads
  - used by system programmers, but ···

    ··· application programmers have better things to do!
- Many application codes can be usefully supported by higher level constructs
  - led to proprietary *directive* based approaches of Cray, SGI, Sun etc
- OpenMP is an API for shared memory parallel programming targeting Fortran, C and C++
  - standardizes the form of the proprietary directives
  - avoids the need for explicitly setting up mutexes, condition variables, data scope, and initialization

OpenMP

- Specifications maintained by OpenMP Architecture Review Board (ARB)
  - members include AMD, Intel, Fujitsu, IBM, NVIDIA ··· cOMPunity
- Versions 1.0 (Fortran ’97, C ’98), 1.1 and 2.0 (Fortran ’00, C/C++ ’02), 2.5 (unified Fortran and C, 2005), 3.0 (2008), 3.1 (2011), 4.0 (2013), 4.5 (2015)
- Comprises compiler directives, library routines and environment variables
  - C directives (case sensitive)

    ```
    #pragma omp directive_name [clause-list]
    ```

  - library calls begin with `omp_`

    ```
    void omp_set_num_threads(nthreads)
    ```

  - environment variables begin with `OMP_`

    ```
    export OMP_NUM_THREADS=4
    ```

- OpenMP requires compiler support
  - activated via `-fopenmp` (gcc) or `-qopenmp` (icc) compiler flags

The `Parallel` Directive

- OpenMP uses a fork/join model, i.e. programs execute serially until they encounter a `parallel` directive:
  - this creates a group of threads
  - the number of threads dependent on an environment variable or set via function call
  - the main thread becomes master with thread id 0

```
#pragma omp parallel [clause-list]
    /* structured block */
```

- Each thread executes a *structured block*

Fork-Join Model

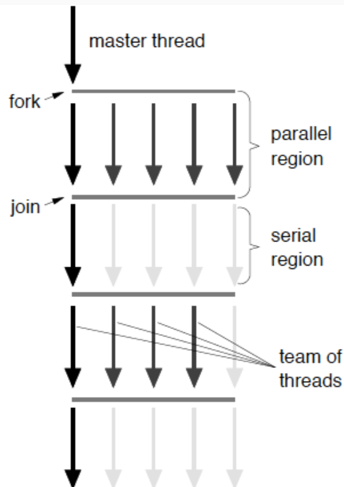


Figure 6.1: Model for OpenMP thread operations: The master thread "forks" team of threads, which work on shared memory in a parallel region. After the parallel region, the threads are "joined," i.e., terminated or put to sleep, until the next parallel region starts. The number of running threads may vary among parallel regions.

Introduction to High Performance Computing for Scientists and Engineers, Hager and Wellein, Figure 6.1

`Parallel` Clauses

Clauses are used to specify

- **Conditional Parallelization:** to determine if parallel construct results in creation of threads

```
if (scalar expression)
```

- **Degree of concurrency:** explicit specification of the number of threads created

```
num_threads (integer_expression)
```

- **Data handling:** to indicate if specific variables are local to thread (allocated on its stack, the default), global, or "special"

```
private(variable_list)
shared(variable_list)
firstprivate(variable_list)
```

Compiler Translation: OpenMP to pthreads

- OpenMP code

```
int a,b;
main() {
  //serial segment
  #pragma omp parallel num_threads(8) private(a) shared(b)
  {
    //parallel segment
  }
  //rest of serial segment
}
```

- pthreads equivalent (structured block is *outlined*)

```
int a, b;
main() {
  //serial segment
  for (i=0; i<8; i++) pthread_create(....., internal_thunk,...);
  for (i=0; i<8; i++) pthread_join(........);
  //rest of serial segment
}
void *internal_thunk(void *packaged_argument) {
  int a;
  // parallel segment
}
```

Parallel Directive Examples

```
#pragma omp parallel if (is_parallel == 1) num_threads(8) \
                private(a) shared(b) firstprivate(c)
```

- If value of variable is_parallel is one, eight threads are used
- Each thread has private copy of a and c, but shares a single copy of b
- Value of each private copy of c is initialized to value of c before parallel region

```
#pragma omp parallel reduction(+:sum) num_threads(8) default(none)
```

- Eight threads get a copy of variable sum
- When threads exit, the values of these local copies are accumulated into the sum variable on the master thread
  - other reduction operations include *, -, &, |, ^, && and ||
- All variables are private unless otherwise specified

Example: Computing Pi

- Compute π by generating random numbers in square with side length of 2 centered at (0,0) and counting numbers that fall within a circle of radius 1
  - The area of square = 4, area of circle = $\pi r^2 = \pi$
  - The ratio of points in circle to the outside approaches $\pi/4$

```
#pragma omp parallel shared(npoints) reduction(+: sum) num_threads(8)
{ seed = omp_get_thread_num();
  num_threads = omp_get_num_threads();
  sample_points_per_thread = npoints/num_threads;
  sum = 0;
  for (i = 0; i < sample_points_per_thread; i++){
    rand_x = (double) rand_range(&seed, -1, 1);
    rand_y = (double) rand_range(&seed, -1, 1);
    if ((rand_x * rand_x + rand_y * rand_y) <= 1.0)
      sum++;
  }
}
```

- The OpenMP code is very simple - try writing the equivalent pthread code!

The `for` Worksharing Directive

- Used in conjunction with `parallel` directive to partition the `for` loop immediately afterwards

```
#pragma omp parallel shared(npoints) reduction(+: sum) num_threads(8)
{
  sum = 0;
  seed = omp_get_thread_num();
  #pragma omp for
  for (i = 0; i < npoints; i++) {
    rand_x = (double) rand_range(&seed, -1, 1);
    rand_y = (double) rand_range(&seed, -1, 1);
    if ((rand_x * rand_x + rand_y * rand_y) <= 1.0)
      sum++;
  }
}
```

  - The loop index (`i`) is assumed to be private
  - Only two directives plus the sequential code (code is easy to read/maintain)
- There is implicit synchronization at the end of the loop
  - Can add a `nowait` clause to prevent synchronization

The Combined `parallel for` Directive

- The most common use case for parallelizing `for` loops

```
#pragma omp threadprivate(seed)
#pragma omp parallel
{ seed = omp_get_thread_num(); }
sum = 0;
#pragma omp parallel for shared(npoints) reduction(+: sum) num_threads(8)
for (i = 0; i < npoints; i++) {
  rand_x = (double) rand_range(&seed, -1, 1);
  rand_y = (double) rand_range(&seed, -1, 1);
  if ((rand_x * rand_x + rand_y * rand_y) <= 1.0)
    sum++;
  }
}
printf("sum=%d\n", sum);
```

- Inside the parallel region, `sum` is treated as a thread-local variable (implicitly initialized to 0)
- At the end of the region, the thread-local versions of `sum` are added to the global `sum` (here initialized to 0) to get the final value

Assigning Iterations to Threads

- The `schedule` clause of the `for` directive assigns iterations to threads

  `schedule(scheduling_clause[,parameter])`

- `schedule(static[,chunk-size])`
  - splits the iteration space into chunks of size `chunk-size` and allocates to threads in a round-robin fashion
  - no specification implies the number of chunks equals the number of threads
- `schedule(dynamic[,chunk-size])`
  - iteration space split into `chunk-size` blocks that are scheduled dynamically
- `schedule(guided[,chunk-size])`
  - chunk size decreases exponentially with iterations to minimum of `chunk-size`
- `schedule(runtime)`
  - determine scheduling based on setting of `OMP_SCHEDULE` environment variable

Loop Schedules

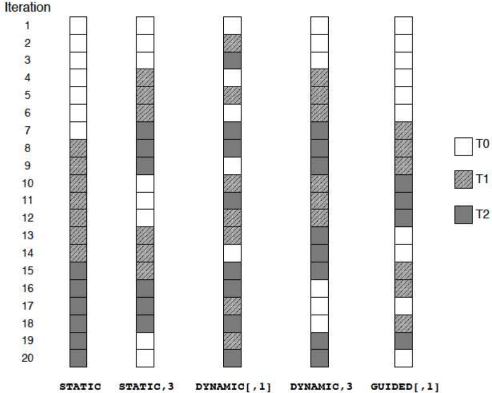


Figure 6.2: Loop schedules in OpenMP. The example loop has 20 iterations and is executed by three threads (T0, T1, T2). The default chunksize for DYNAMIC and GUIDED is one. If a chunksize is specified, the last chunk may be shorter. Note that only the STATIC schedules guarantee that the distribution of chunks among threads stays the same from run to run.

Introduction to High Performance Computing for Scientists and Engineers, Hager and Wellein, Figure 6.2

Sections

- Consider partitioning of fixed number of tasks across threads
  - much less common than `for` loop partitioning
  - explicit programming naturally limits number of threads (scalability)

```
#pragma omp sections
{
  #pragma omp section
  {
    taskA()
  }
  #pragma omp section
  {
    taskB()
  }
}
```

- Separate threads will run `taskA` and `taskB`
- Illegal to branch in or out of section blocks

Nesting Parallel Directives

- What happens for nested `for` loops

```
#pragma omp parallel for num_threads(2)
  for (i = 0; i < Ni; i++) {
    #pragma omp parallel for num_threads(2)
    for (j = 0; j < Nj; j++) {
```

- By default inner loop is serialized and run by one thread
- To enable multiple threads in nested parallel loops requires environment variable `OMP_NESTED` to be `TRUE`
- Note - the use of synchronization constructions in nested parallel sections requires care (see OpenMP specs)!

Collapsing Nested Loops

- A better approach: apply parallel directive to entire loop nest:

```
#pragma omp parallel for collapse(2)
for (i = 0; i < Ni; i++) {
  for (j = 0; j < Nj; j++) {
```

- All $N_i \times N_j$ iterations are distributed across threads

Synchronization #1

- **Barrier**: threads wait until they have all reached this point

```
#pragma omp barrier
```

- **Single**: following block executed only by first thread to reach this point
  - others wait at end of structured block unless `nowait` clause used

```
#pragma omp single [clause-list]
    /* structured block */
```

- **Master**: only master executes following block, other threads do NOT wait

```
#pragma omp master
   /* structured block */
```

- **Critical Section**: only one thread is ever in the named critical section

```
#pragma omp critical [(name)]
    /* structured block */
```

Synchronization #2

- **Atomic**: memory location(s) referenced in expression are accessed atomically

```
#pragma omp atomic
// expression-statement
```

- **Ordered**: some operations within a `for` loop must be performed in sequential order

```
cumul_sum[0] = list[0];
#pragma omp parallel for shared(cumul_sum, list, n)
for (i=1; i<n; i++) {
  /* other processing on list[i] if required*/
  #pragma omp ordered
  {
    cumul_sum[i] = cumul_sum[i-1] + list[i];
  }
}
```

- **Flush:** ensures a consistent view of memory
  - that variables have been flushed from registers into memory

```
#pragma omp flush[(list)]
```

Data Handling

- `private`: an uninitialized local copy of variable made for each thread
- `shared`: variables shared between threads
- `firstprivate`: make a local copy of an existing variable and assign it same value
  - often better than multiple reads to shared variable
- `lastprivate`: copies back to master value from thread executing the equivalent of the last loop iteration if executed serially
- `threadprivate`: creates private variables but they persist between multiple parallel regions maintaining their values
- `copyin`: like first private but for threadprivate variables

OpenMP Tasks

- A task has
  - Code to execute
  - A data environment (it owns its data)
  - An assigned thread that executes the code and uses the data
- Creating a task involves two activities: packaging and execution
  - Each encountering thread packages a new instance of a task (code and data)
  - Some thread in the team executes the task at some later time

OpenMP Task Syntax

```
#pragma omp task [clause ...]
                 if (scalar expression)
                 final (scalar expression)
                 untied
                 default (shared | none)
                 mergeable
                 private (list)
                 firstprivate (list)
                 shared (list)
   structured_block
```

- When `if` clause is false, the creating thread executes the task immediately (in its own environment)
- Task completes at thread barriers (explicit or implicit) and task barriers

```
#pragma omp taskwait
```

- Applies only to child tasks generated in the current task, not to “descendants”

OpenMP Tasks Example

```
int fib(int n) {
  if (n < 2) return n;
  else {
    int i, j;
    #pragma omp task shared(i) firstprivate(n)
    i=fib(n-1);

    #pragma omp task shared(j) firstprivate(n)
    j=fib(n-2);

    #pragma omp taskwait
    return i+j;
  }
}
int main() {
  int n = 10;
  omp_set_num_threads(4);

  #pragma omp parallel shared(n)
  { #pragma omp single
    printf ("fib(%d) = %d\n", n, fib(n));
  }
}
```

Task Issues

- Task switching
  - Certain constructs have task scheduling points at defined locations within them
  - When a thread encounters a task scheduling point, it is allowed to suspend the current task and execute another (task switching)
  - It can then return to original task and resume
- Tied Tasks
  - By default suspended tasks must resume execution on the same thread as it was previously executing on
  - "untied" clause relaxes this constraint
- Task Generation
  - Very easy to generate many tasks very quickly!
  - Generating task will be suspended and start working on a long and boring task
  - Other threads can consume all their tasks and have nothing to do

Library Functions#1

- Defined in header file

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

- Controlling threads and processors

```
void omp_set_num_threads(int num_threads)
int omp_get_num_threads()
int omp_get_max_threads()
int omp_get_thread_num()
int omp_get_num_procs()
int omp_in_parallel()
```

- Controlling thread creation

```
void omp_set_dynamic(int dynamic_threads)
int omp_get_dynamic()
void omp_set_nested(int nested)
int omp_get_nested()
```

Library Functions#2

- Mutual exclusion

```
void omp_init_lock(omp_lock_t *lock)
void omp_destroy_lock(omp_lock_t *lock)
void omp_set_lock(omp_lock_t *lock)
void omp_unset_lock(omp_lock_t *lock)
int omp_test_lock(omp_lock_t *lock)
```

- Nested mutual exclusion

```
void omp_init_nest_lock(omp_nest_lock_t *lock)
void omp_destroy_nest_lock(omp_nest_lock_t *lock)
void omp_set_nest_lock(omp_nest_lock_t *lock)
void omp_unset_nest_lock(omp_nest_lock_t *lock)
int omp_test_nest_lock(omp_nest_lock_t *lock)
```

OpenMP Environment Variables

- `OMP_NUM_THREADS`: default number of threads entering parallel region
- `OMP_DYNAMIC`: if `TRUE` it permits the number of threads to change during execution, in order to optimize system resources
- `OMP_NESTED`: if `TRUE` it permits nested parallel regions
- `OMP_SCHEDULE`: determines scheduling for loops that are defined to have `runtime` scheduling

```
export OMP_SCHEDULE="static,4"
export OMP_SCHEDULE="dynamic"
export OMP_SCHEDULE="guided"
```

OpenMP and pthreads

- OpenMP removes the need for a programmer to initialize task attributes, set up arguments to threads, partition iteration spaces etc
- OpenMP code can closely resemble serial code
- OpenMP is particularly useful for static or regular problems
- OpenMP users are hostage to availability of an OpenMP compiler
  - performance heavily dependent on quality of compiler
- pthreads data exchange is more apparent so false sharing and contention is less likely
- pthreads has a richer API that is much more flexible, e.g. condition waits, locks of different types etc
- pthreads is library based

Must balance above before deciding on parallel model

Hands-on Exercise: Programming with OpenMP

Objective:

- To use OpenMP to provide a basic introduction to shared memory parallel programming

Summary

Topics covered today - Multiprocessor Parallelism:

- Threads and OS support
- Mutual Exclusion
- Shared memory synchronization constructs
- Memory Consistency
- OpenMP

Tomorrow - Parallel Performance Optimization!