C-DAC Four Days Technology Workshop

ON

Hybrid Computing – Co-Processors/Accelerators Power-aware Computing – Performance of Applications Kernels

hyPACK-2013
(Mode-1:Multi-Core)

Lecture Topic:

Multi-Core Processors : Shared Memory Prog:

Pthreads Part-III

Venue : CMSD, UoHYD ; Date : October 15-18, 2013

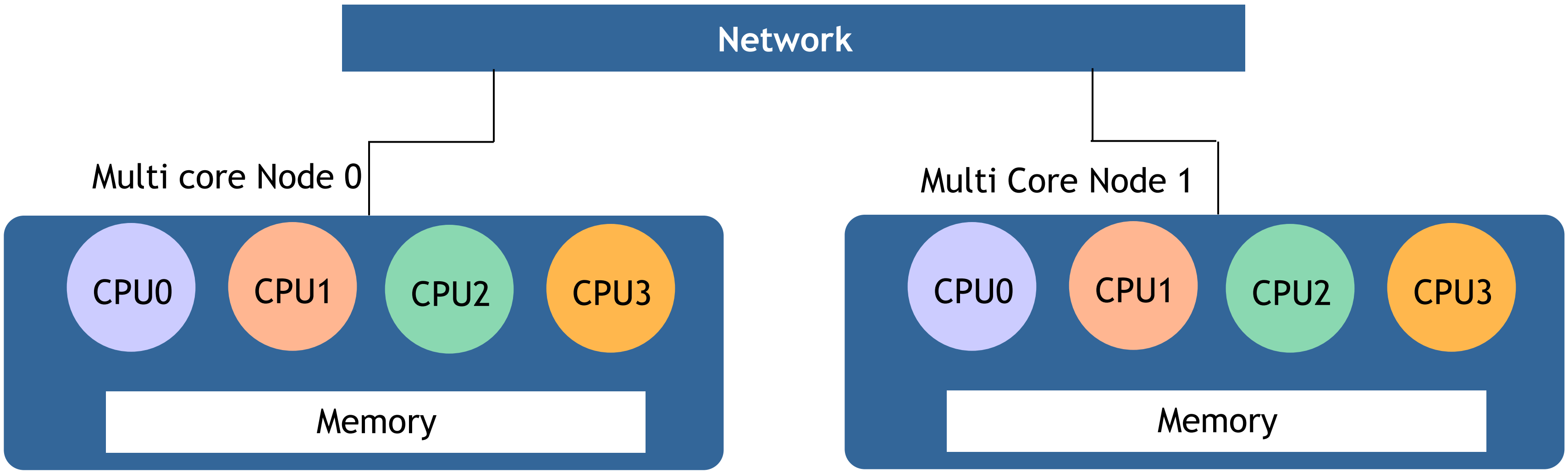
The POSIX Threads (Pthreads) Model

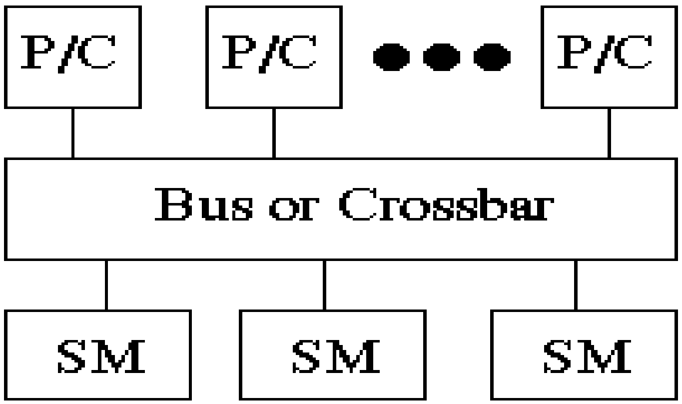
Lecture Outline

Following Topics will be discussed

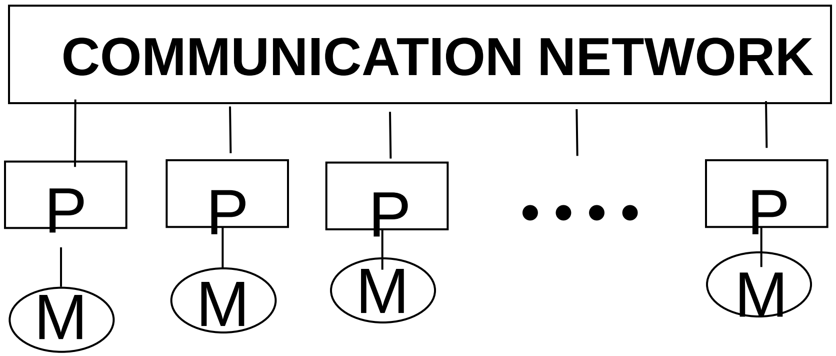
- ❖ Examples of Threaded Programs
- ❖ Understanding Pthreads implementation
- ❖ Pthread Synchronization Primitives
- ❖ Pthread - Performance issues

General-Purpose Clusters /Multi Cores





Shared Memory



Non-Shared Memory

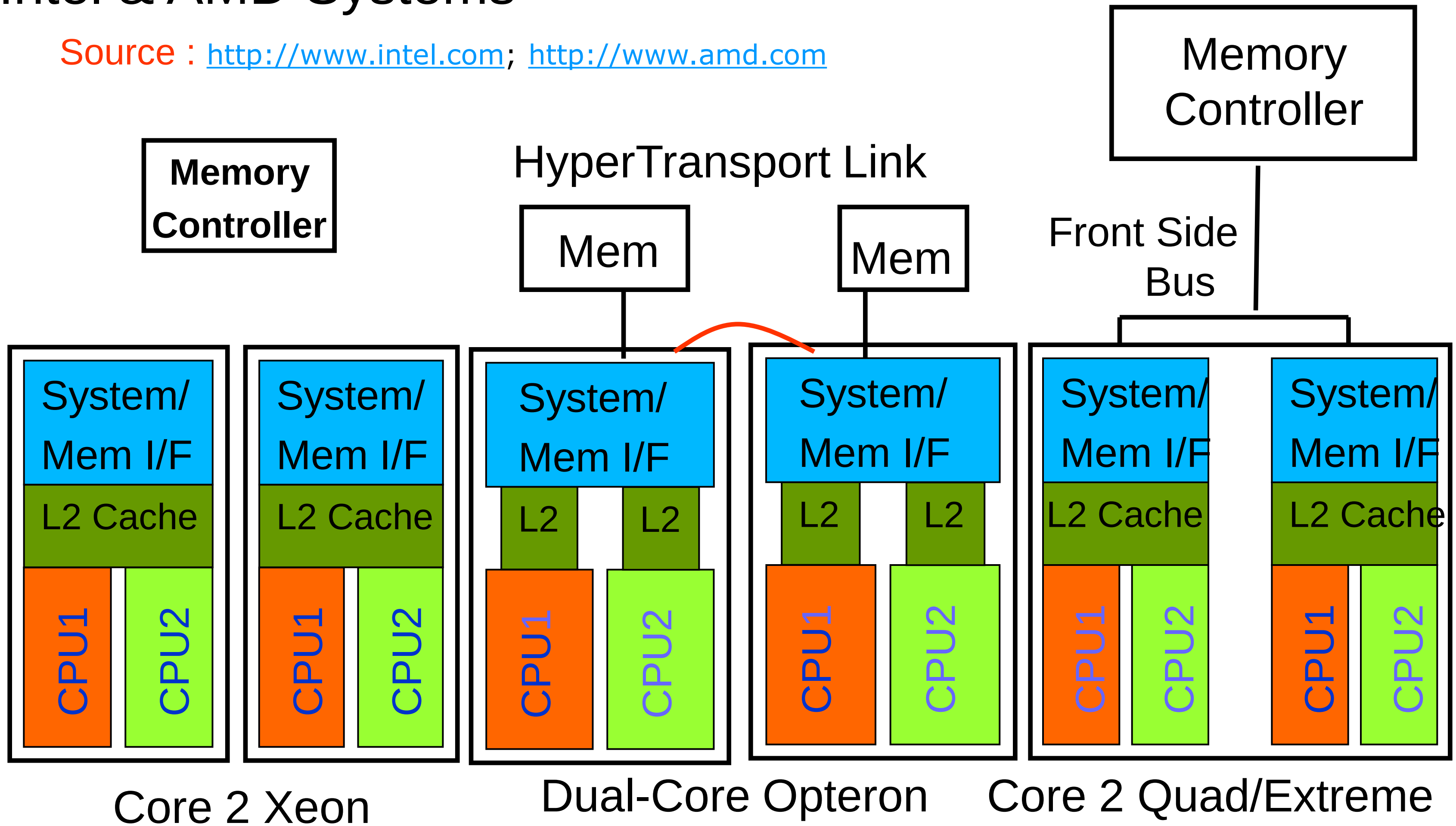
Common Performance Problems with Shared Memory

- ❖ Excessive communication
  - ➢ Large number of remote memory accesses
  - ➢ False sharing
  - ➢ False data mapping
- ❖ Frequent synchronization
  - ➢ Implicit synchronization of parallel constructs
  - ➢ Barriers, locks, …
- ❖ Load balancing
  - ➢ Uneven scheduling of parallel loops
  - ➢ Uneven work in parallel sections
- ❖ Cost of communication in shared address space machines
  - ➢ Costs are associated with read and write operations that may be local or non-local data.

Multi Cores Today

Intel & AMD Systems

Source : http://www.intel.com; http://www.amd.com



Source : Reference [4],[6], [7]

Multi-threaded Programming Models

- Parallel Program comprised of multiple Concurrent threads of Computation
- Work is partitioned amongst the threads
- Data communication between the threads via shared memory or messages
  - Shared Memory ; more convenient than explicit messages, but danger of Race Conditions
  - Message Passing : More tedious than use of Shared Memory, but lower likelihood of races
  - Examples : OpenMP, MPI, Pthreads, CUDA

Programming Multicore Processors

- Explicit Parallel Programming
  - Thread-based Programming Models.
  - Data Parallel Programming Models
  - Stream Programming Models
- Automatic Parallelization
  - Features of Most compliers for SMP systems, but currently see very little practical use
  - Polyhedral framework for dependencies and loop transformations – enabling composition of complex transformations over multiple statements.

Source : Reference [4],[6], [7]

Operational Flow of Threads for an Application

Implementation Source Code

. . .
.
.
.

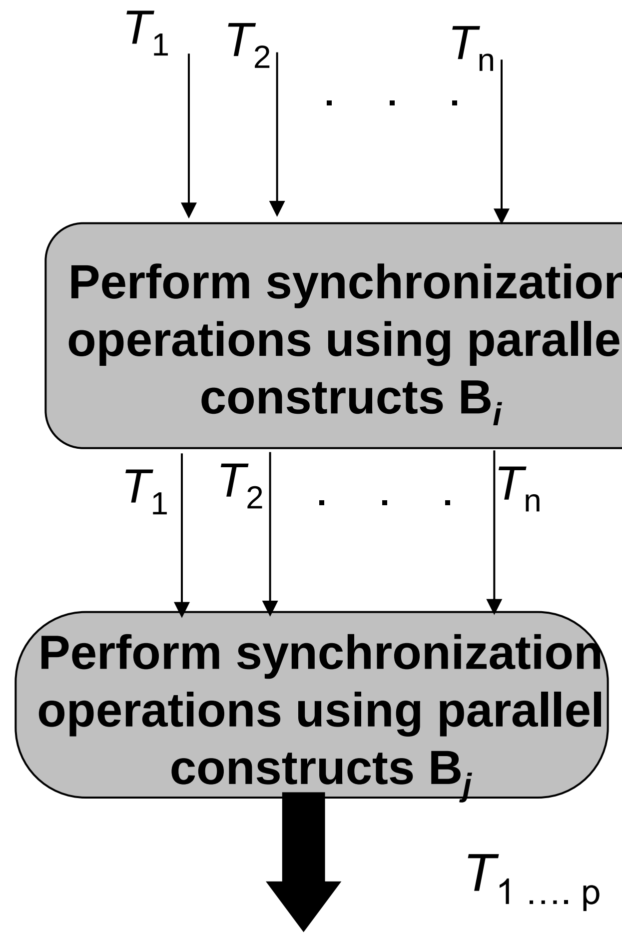
Parallel Code Block or a section needs multithread synchronization

.
.
.

Parallel Code Block

⋮

Operational Flow of Threads



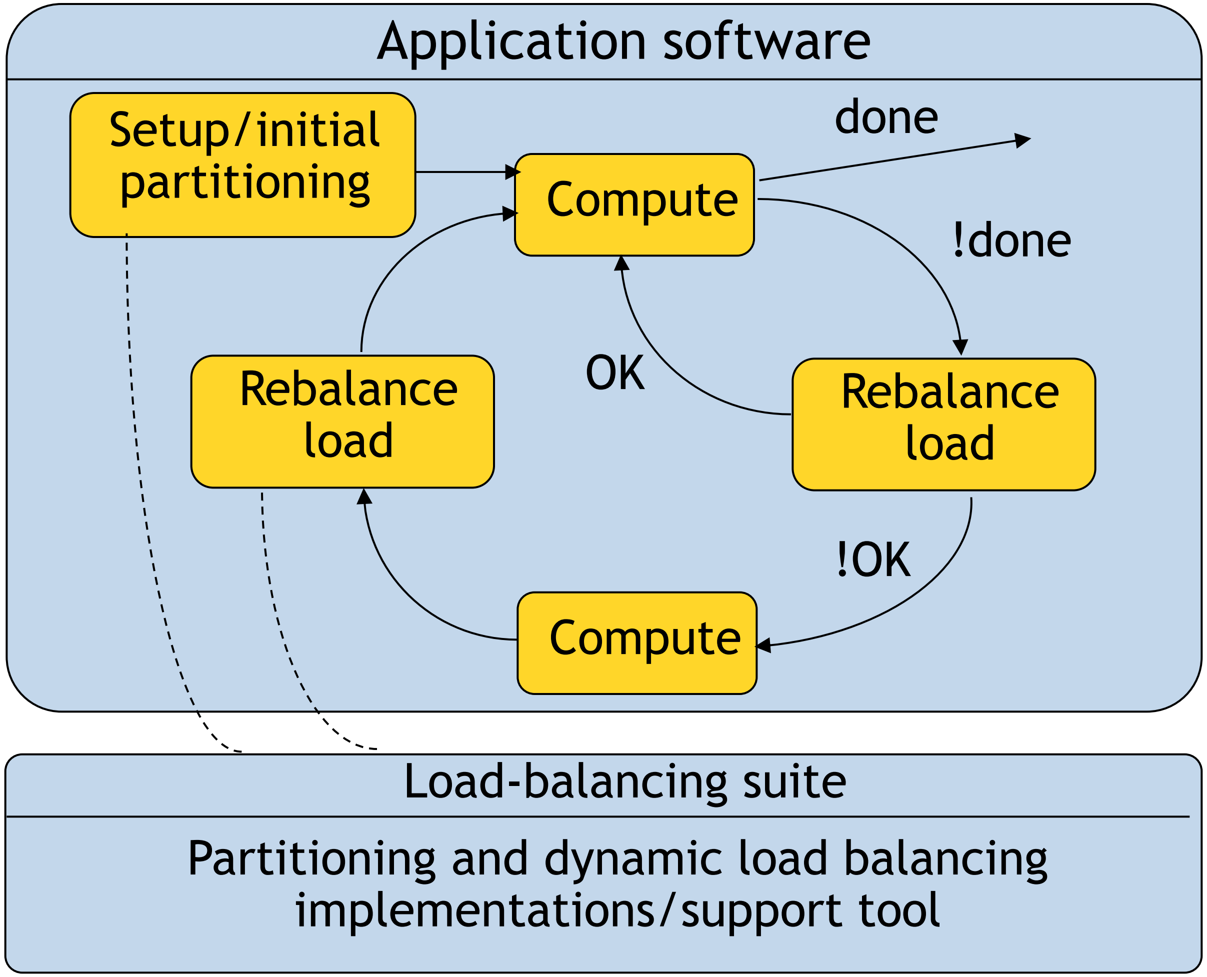
Source : Reference [4],[6], [7]

Application Perspective : Multi Cores

❖ Threads of Computation : Work is partitioned amongst the threads – Data Handling & Synchronization Issues

Computational requirements dynamically changes –

- *Cache Friendly applications*
- *I/O Intensive applications*



Source : Reference [4],[6]

Why Pthreads ? : Thread Model

Implementation specific issues of Pthreads :

- Synchronization
- Sharing Process Resources
- Communication
- Scheduling

Thread Pitfalls

- Shared data
  - 2 threads perform
    A = A + 1

Thread 1:	Thread 1:
1) Load A into R1	1) Load A into R1
2) Add 1 to R1	2) Add 1 to R1
3) Store R1 to A	3) Store R1 to A

  - Mutual exclusion preserves correctness
    - Locks/mutexes
    - Semaphores
    - Monitors
    - Java “synchronized”

- False sharing
  - Non-shared data packed into same cache line

```
int thread1data;
int thread1data;
```

  - Cache line ping-pongs between CPUs when threads access their data
- Locks for heap access
  - malloc() is expensive because of mutual exclusion
  - Use private

Thread Spawning Issues

- How does a thread know which thread it is? Does it matter?
  - Yes, it matters if threads are to work together
  - Could pass some identifier in through parameter
  - Could contend for a shared counter in a critical section
  - `pthread_self()` returns the thread ID, but doesn't help.
- How big is a thread's stack?
  - By default, not very big. (What are the ramifications?)
  - `pthread_attr_setstacksize()` changes stack size

Join Issues

- Main thread must join with child threads (`pthread_join`)
  - Why?
  - Ans: So it knows when they are done.
- `pthread_join` can pass back a 32-bit value
  - Can be used as a pointer to pass back a result
  - What kind of variable can be passed back that way? Local? Static? Global? Heap?

Thread-safe Functions and Libraries

- Memory Issues

- Bandwidth
  - Avoid Memory Contention Issues
  - How *fast* read and *Write* variables – I/O

- Working in Cache
  - Cache Un-friendly Programming
  - Loop Optimization techniques

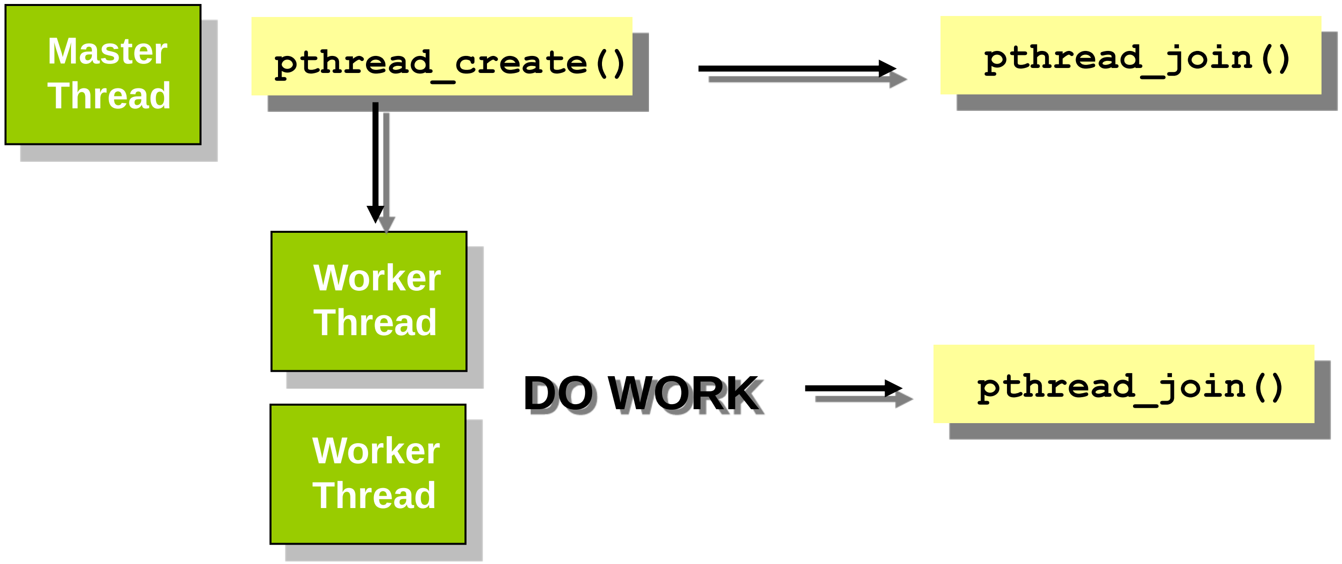
- Memory Contention
  - Read-write dependency (A core writes a cache line, and then a different core reads it
  - Write-write Dependency (Core write a cache line, and then a different writes it)

The Thread Management

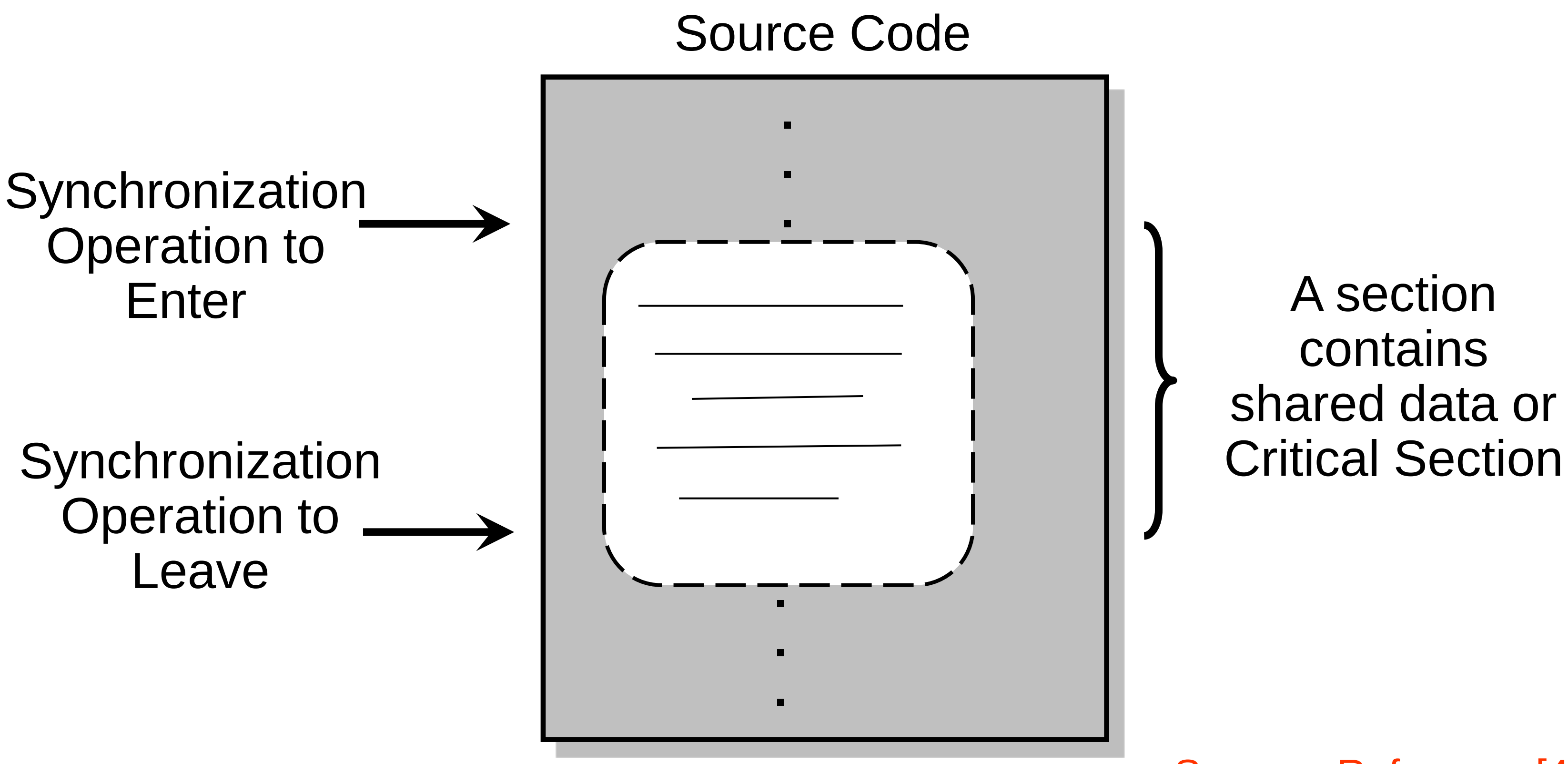
- ❖ Stack Management (pthread_attr_getstacksize, pthread_attr_setstacksize,
- ❖ Mutex Variables
  - ➢ Mutex variables are one of the primary means of implementing thread synchronization and for protecting write occur.
  - ➢ A mutex variable acts like a “lock” protecting access to a shared data resource.
  - ➢ Mutex can be used to prevent “race” conditions.
  - ➢ Creating and Destroying Mutexes
  - ➢ Locking and Unlocking Mutexes

The Thread Management

- Creating and Terminating Threads (pthread_create, pthread_exit, pthread_attr_init, pthread_attr_destroy)
- Passing Arguments to Threads
- Joining and Detaching Threads (pthread_join, pthread_detach, pthread_attr_setdetachstate, pthread_attr_getdetachstate)



Generic Representation of Synchronization Block inside Source Code



Source : Reference [4],[6], [7]

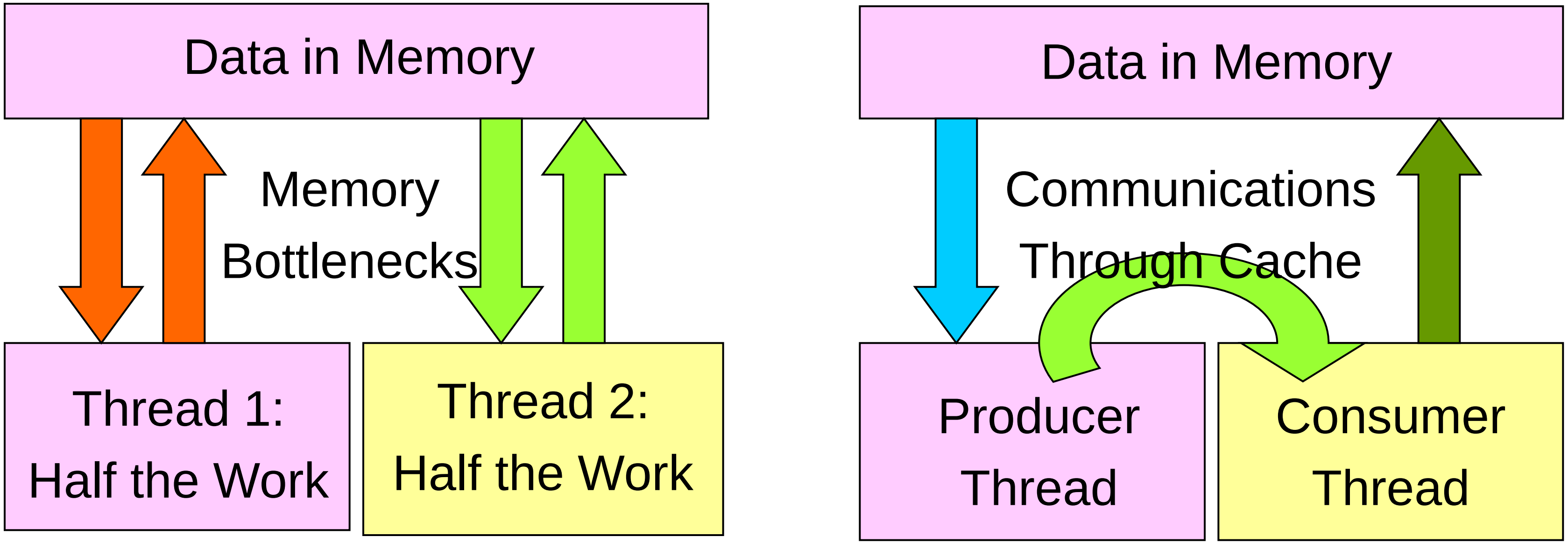
- Two types of Synchronization operations are widely used : Mutual exclusion and Condition synchronization

Synchronizing Primitives in Pthreads

- Common Synchronization Mechanism
  - *Read/Write exclusion*
  - *Thread safe* data structures
  - *Condition variable* functions
  - *Semaphores*
- *Mutex* Variables
  - To protect a shared resource from a race condition, we use a type of synchronization called *mutex* exclusion, or *mutex fort* short
  - Critical section : Provide access to the code paths or routines that access data -
  - How large does a critical section have to be to require protection through a *mutex* ?
  - *Pthread* library operations such as *mutex* locks and unlocks work properly regardless of the platform you are using and the number of CPUs in the system.

Producer/Consumer Problem : Synchronizing Issues

- ❖ Producer thread generates tasks and inserts it into a work-queue.
- ❖ The consumer thread extracts tasks from the task-queue and executes them one at a time.



Source : Reference [4],[6], [7]

Producer/Consumer Problem : Synchronizing Issues

❖ **Possibilities & Implementation Issues on Multi cores**

- ➢ The producer thread must not overwrite the shared buffer when the previous task has not been picked up by a consumer thread
- ➢ The consumer threads must not pick-up tasks until there is something present in the shared data structure.
- ➢ Individual consumer threads should pick-up tasks one at a time.

❖ Implementation can be done mutexes, condition Variables

Synchronization Primitives in Pthreads

- Controlling Thread Attributes and Synchronization
  - Attribute Objects for Threads
  - Attribute Objects for Mutexes
- Thread Cancellation
  - Clean-up functions are invoked for reclaiming the thread data structures
- Composite synchronization Primitives
  - Read-Write Locks (Data Structure is read frequently but written infrequently.
  - Issues of Multiple reads /Serial writes
  - Issues of Read Locks; read-write locks etc…

Synchronization Primitives in Pthreads

- **Barriers**
  - A barrier call is used to hold a thread until all other threads participating in the barrier have reached the barrier
  - Barriers can be implemented using a counter, a mutex, and a condition variable.
  - A single integer is used to keep track of the number of threads that have reached the Barrier
- **Remark :**
  - Barrier implementation using mutexes may suffer from the overhead of busy-wait.

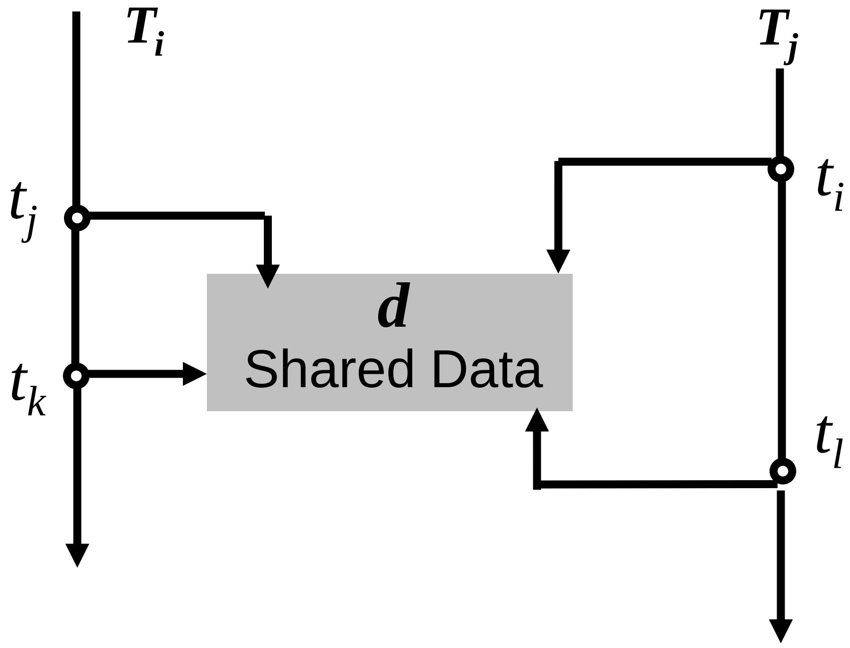
Synchronization Primitives in Pthreads

- Mutual Exclusion for Shared Variables
  - Thread APIs provide support for implementing critical sections and atomic operations using mutex-locks (mutual exclusion locks)
- Condition Variables for Synchronization
  - When thread performs a condition wait, it takes itself off the runnable list – Does not use any CPU cycle

Remark :

- Mutex Lock consumes CPU cycles as it polls for the lock
- Condition wait consumes CPU cycles when it is woken up

Synchronization order



$T = f(t)$
$d = f(t) = s(\ldots,t_i,t_j,t_k,t_l,\ldots)$

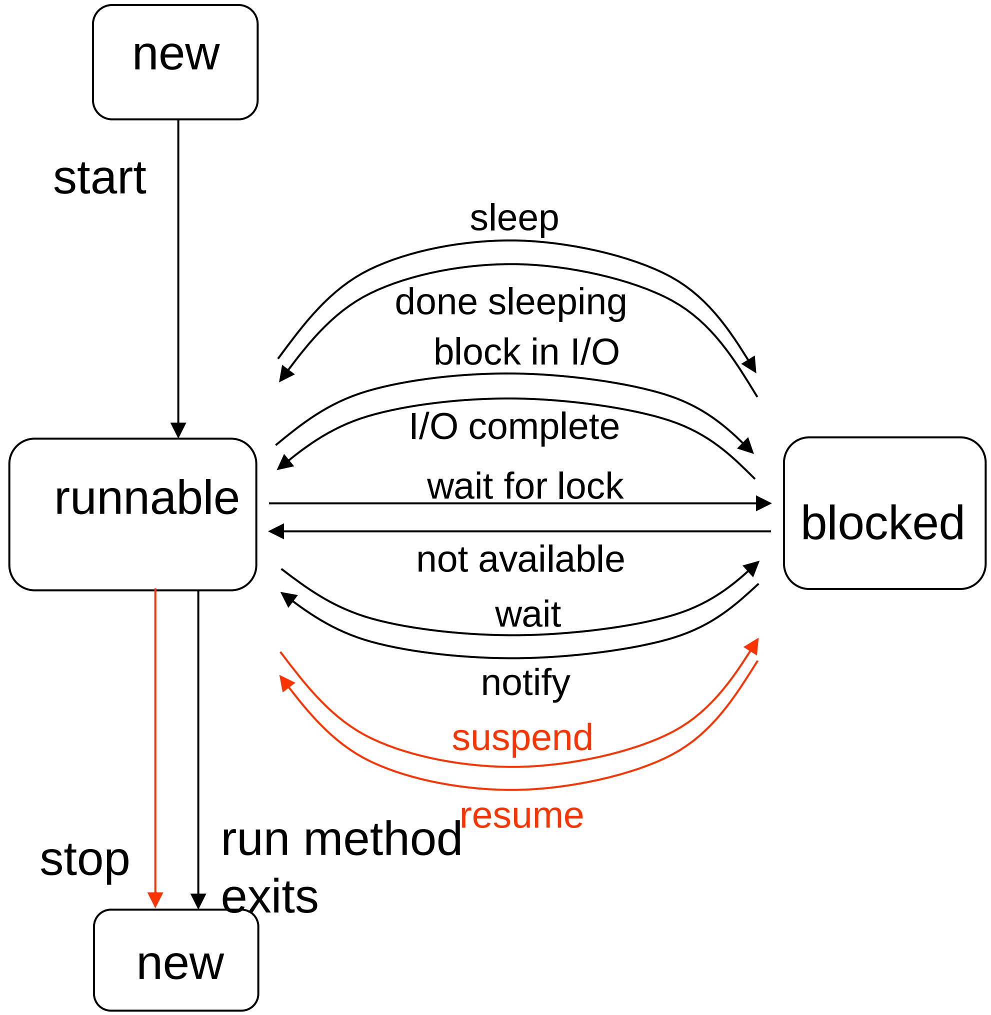
Shared data *d* depends on synchronization functions of time

Shared Data Synchronization, Where Data *d* is protected by a Synchronization Operation

Source : Reference [4],[6], [7]

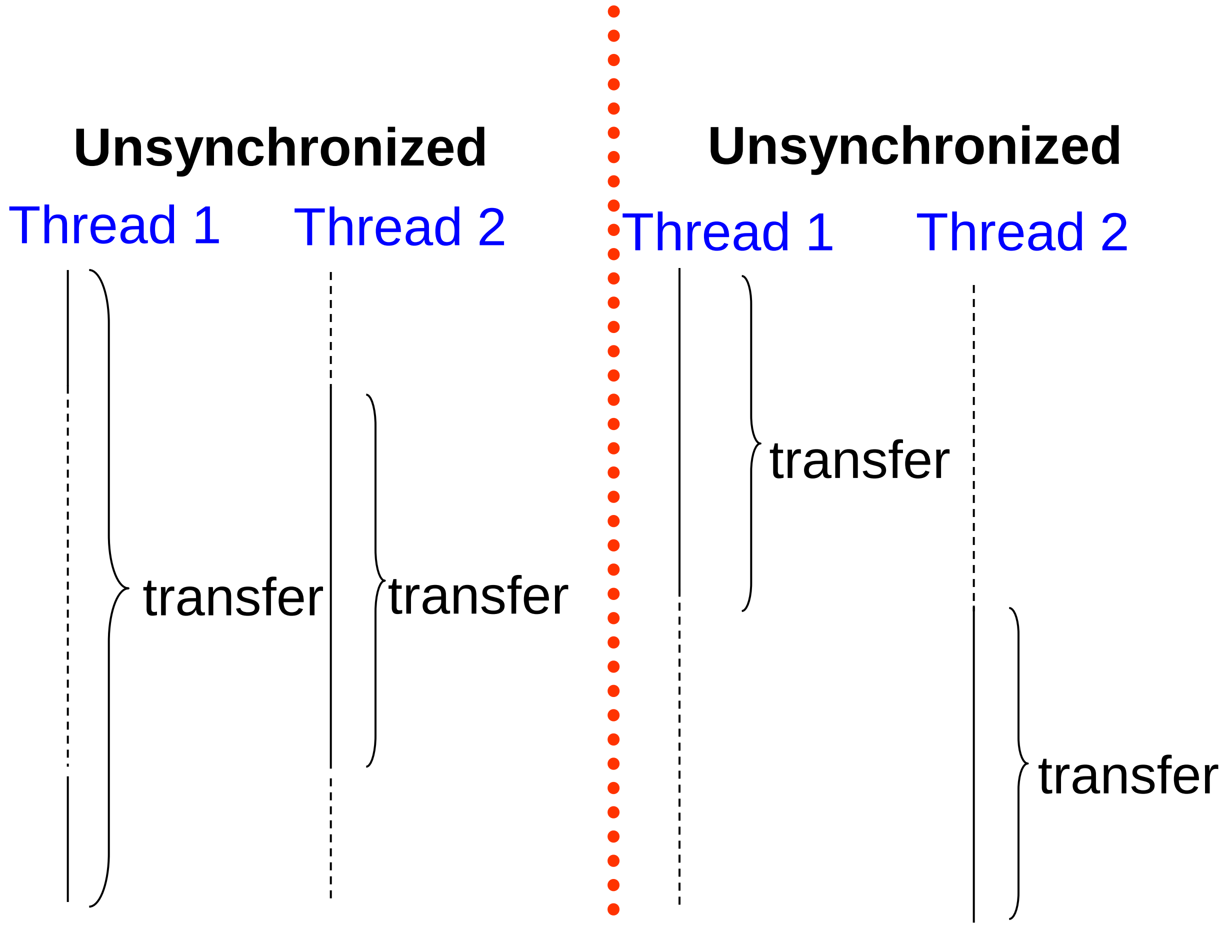
Pthreads:Synchronization & Thread States

- I/O Requests
- Read-Write Locks
- Available CPU
- Release Locks
- Critical Sections



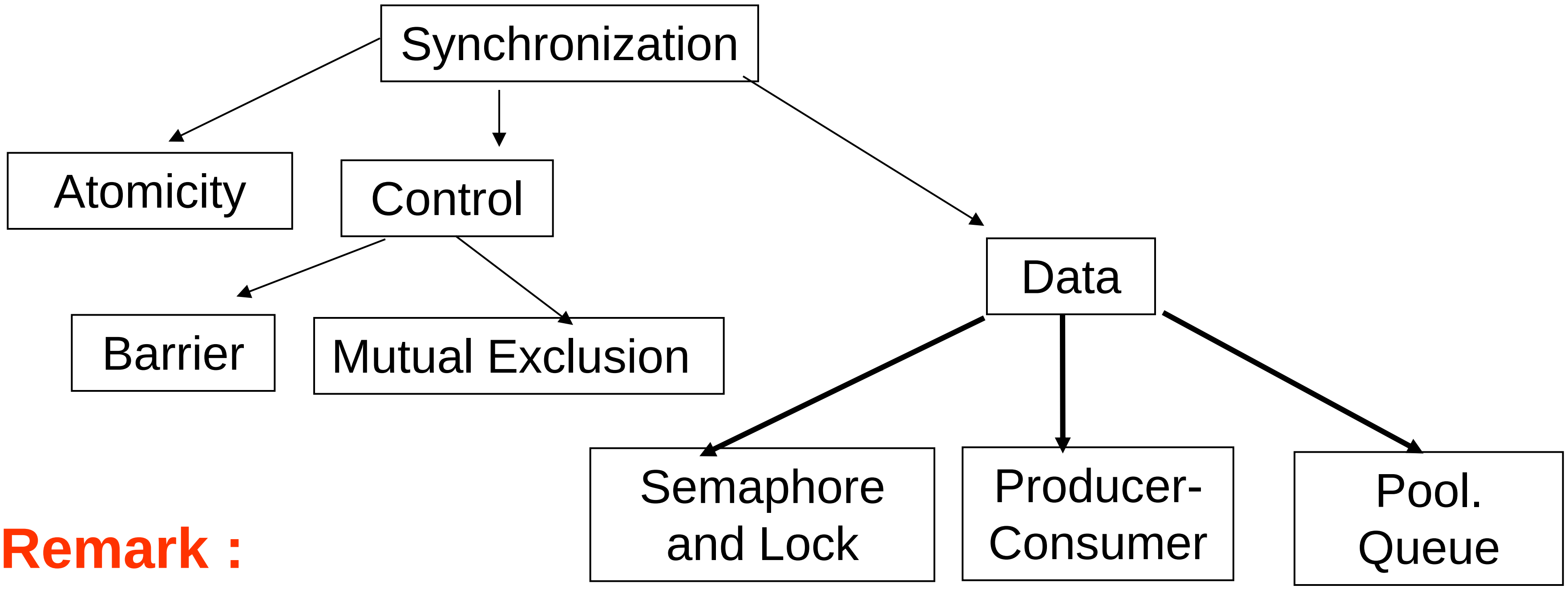
Comparison of unsynchronized / synchronized threads

- Too little / too much synchronization
  - In-Correct Results
  - Performance – Slow done the results



Source : Reference [4],[6], [7]

Pthreads : Various types of synchronization



Remark :

- ❖ Use of Scheduling techniques as means of Synchronization is not encouraged. – Thread Scheduling Policy ,High Priority & Low Priority Threads
- ❖ Atomic operations are a fast and relatively easy alternative to mutexes. They do not suffer from the deadlock.

Pthreads:Performance issues-Synchronization Overhead

- Performance depends on input workload :
  - Increasing clients and contention
    - Number of clients *vs* Ratio of Time to Completion
  - Performance depends on a good locking strategy
    - No locks at all;One lock for the entire data base; One lock for each account in the data base
  - Performance depends on the type of work threads do
    - Percentage of Thread I/O *vs* CPU and Ratio of Time to Completion

Pthreads:Performance issues-Synchronization Overhead

❖ How do your threads spend their time ?

- Profiling a program is a good step toward identifying its performance bottlenecks (CPU Utilization, waiting for locks and I/O completion

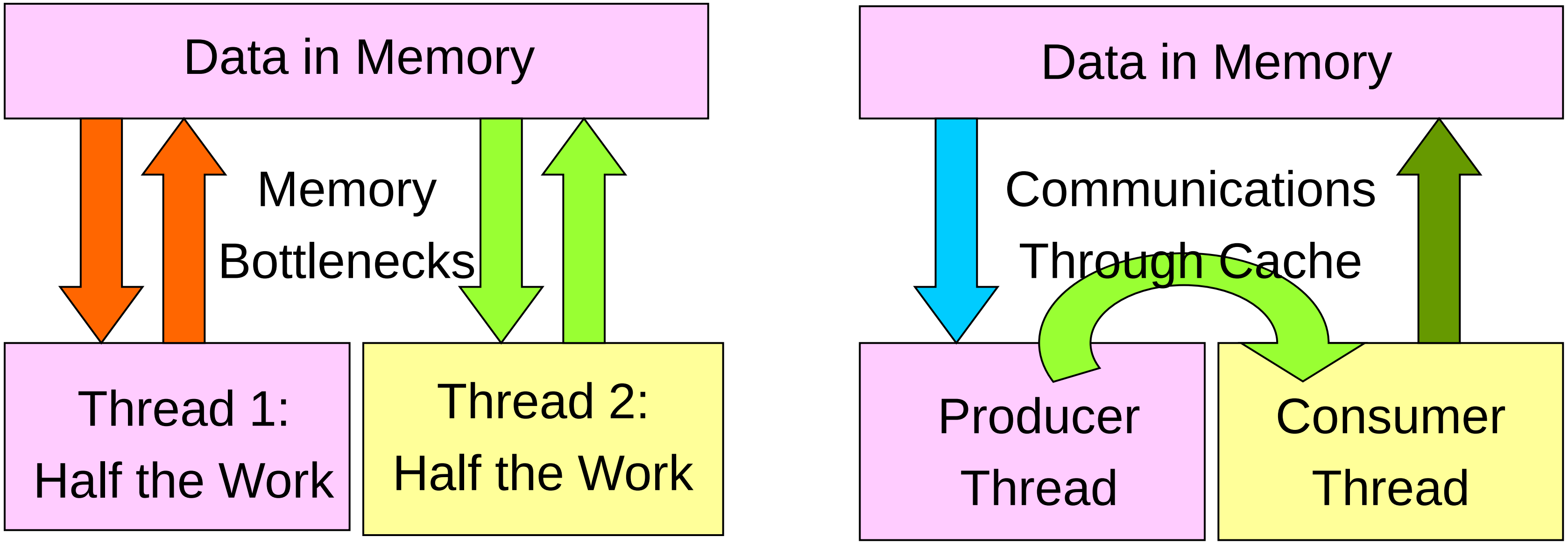
- Do the threads spend most of their time blocked, waiting for their threads to release locks ?

- Are they *runnable* for most of their time but not actually running because other threads are monopolizing the available CPUs ?

- Are they spending most of their time waiting on the completion of I/O requests ?

Producer/Consumer Problem : Multi Threaded Program

- ❖ Producer thread generates tasks and inserts it into a work-queue.
- ❖ The consumer thread extracts tasks from the task-queue and executes them one at a time.



Source : Reference [4],[6], [7]

Pthread APIs

- **Semaphores :** A semaphore is a counter that can have any nonnegative value. Threads wait on a semaphore.
- When the semaphore's value is 0, all threads are forced to wait. When the value is non-zero, a waiting thread is released to work.
- Pthreads does not implement semaphores, they are part of a different POSIX specification.
- Semaphores are used to conjunction with Pthreads' thread-management functionality

Usage : Include <semaphore.h>

- sem_init(*, *, …*);
- sem_post(*, *, …*)
- sem_wait(*, *, …*)

Producer/Consumer Problem : Psuedo code

```
Semaphore s
void producer( ) {
     while (1)   {
          <produce the nest data>
           s->release( )
     }
}
void consumer ( ) {
      while (1)  {
           s->wait( )
           <Consume the next data>
     }
}
```

Remarks : Neither producer nor consumer maintains an order. Synchronization problem exists. Buffer Size needs to be within a boundary to handle.

Producer & Consumer : (1). Using Semaphores; **(2)** Critical Directives (Mutexes – Locks); **(3).** Condition Variables

Source : Reference [4],[6], [7]

Producer/Consumer Problem : Dual Semaphores Solution

```
Semaphore sEmpty, sFull
void producer( ) {
     while (1)   {
           sEmpty->wait ( )
           <produce the nest data >
           sFull->release( )
     }
}
void consumer ( ) {
      while (1)  {
            sFull->release ( )
            <Consume the next data>
            sEmpty->wait ( )
     }
}
```

Remarks : Two independent Semaphores are used to maintain the boundary of buffer. **sEmpty,** and **sFull** retain the constraints of buffer capacity for operating threads.

Source : Reference [4],[6], [7]

What is a Data Race?

A data-race occurs under the following conditions:

- ❖ Two or more threads concurrently accessing the same memory location.
- ❖ At least one of the threads is accessing the memory
- ❖ Location for writing
- ❖ The threads are not using any exclusive locks to control their accesses to that memory.

Data Races, Deadlocks & Live Locks

- Un-synchronized access to shared memory can introduce Race conditions
  - Results depends on relative timings of two or more threads
  - Solaris, Posix Multi-threaded Programming

- **Example :**
  - Two threads trying to add to a shared variable x, which have an initial value of 0.
  - Depending on upon the relative speeds of the threads, the final value of x can be 1, 2, or 3.

Source : Reference [6]

- Parallel Programming would be lot of easier

- Multi-threaded Compiler & Tools may give clue to programmer

Data Races, Deadlocks & Live Locks

- The interactions of Memory, Cache, and Pipeline should be examined carefully.
  - Thread Private
  - Thread shared read only
  - Exclusive Access
  - Read and Write by Unsynchronized threads

Data Races, Deadlocks & Live Locks

❖**Deadlock conditions**

1. A thread is allowed to whole one resource while requesting another
2. No thread is willing to relinquish a resource that is has acquired
3. Access to each resource is exclusive

Original Code	Thread 1	Thread 2
	T = x	u = x x = u +2

Interleaving #1	(x is 0) t = x x= t + 1 (x is 1)	u = x x = u +2 (x is 2)

Deadlock is caused by Cycle of operations

Data Races, Deadlocks & Live Locks

❖ Deadlock Conditions

4. There is a cycle of threads trying to acquire resources, where each resource is held by one thread and requested by another

Deadlock Conditions can be avoided by breaking any one of the conditions

	Thread 1	Thread 2
Interleaving #2	t = x x= t + 1 (x is 1)	(x is 0) u = x x = u +2 (x is 2)
Interleaving #3	(x is 0) t = x x= t + 1 (x is 1)	u = x x = u +2 (x is 3)

Data Races, Deadlocks & Live Locks

- Locks : Locks are similar to semaphores except that a single thread handles lock at one instance. Two basic atomic operations get performed on a lock are acquire() & release ().

- Mutexes : The mutex is the simplest lock an implementation can use.

- Recursive Locks : Recursive locks are locks that may be repeatedly acquired by the thread that currently owns the lock withut causing the thread to deadlock.

- Read-Write Locks : Read-Write locks are also called shared-exclusive or multiple-read/single-write locks or non-mutual exclusion semaphores. Read-Write locks allow simultaneous read access to multiple threads but limit the write access to only one thread.

- Spin Locks : Spin locks are non-blocking locks owned by a thread. Spin locks are used mostly on multiprocessors .

Source : Reference [4],[6]

The Thread Management- Condition Variables

- Creating and Destroying Condiiton Variables (pthread_cond_init, pthread_cond_destroy, pthread_condattr_init, , pthread_condattr_destroy,
- Waiting and Signaling on Condition Variables
- Thread Scheduling
- Thread Specific Data

Cache Line Ping-Pong Caused by False Sharing

❖ Cache Related Issues

- Cache Line (Estimate the cache line size of the Multi core Systems (Remark : Dual Core Processors share L1 Cache)

- False Sharing (The data can be pushed into different cache lines, thereby pushing reduce the false sharing overhead.)

- Performance Impact may vary from problem to problem. (Cache friendly programs such as Dense Matrix Computations & Producer –Consumer using condition variables, mutexes – have different flow of computation and synchronization.)

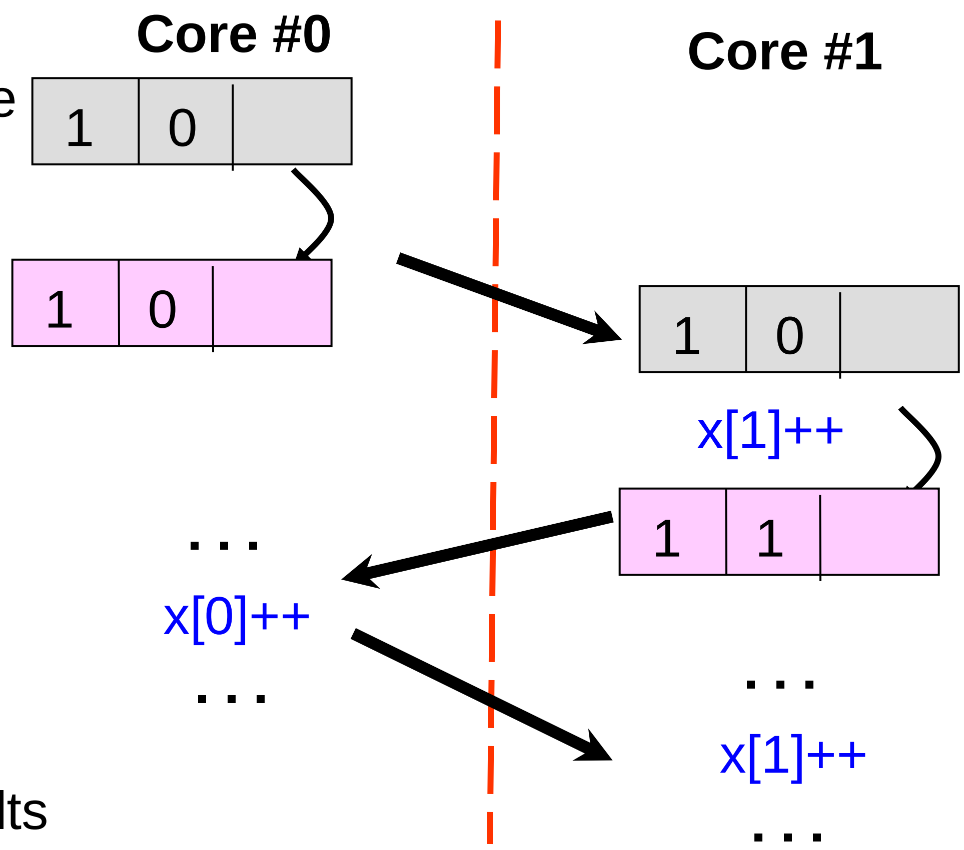
- Use of Scheduling techniques as a means of synchronization may give rise to Memory in-consistency when two threads share the data

Thread-safe Functions and Libraries

Cache Line Ping-Pong Caused by False Sharing

❖Cache Related Issues

- Cache Line
- False Sharing
- Memory consistency
- Performance Impact
- Correctness of the Results



Source : Reference [4],[6]

Cache Line Ping-Pong Caused by False Sharing

- **Remark :** Multiple threads manipulates a single piece of data
- Multiple threads manipulate different parts of large data structure, the programmer should explore ways of breaking it into smaller data structures and making them private to the thread manipulating them
- Making **memory consistency** across the threads is an important and it is for hardware efficiency.

Common Errors /Solutions : Prog. Paradigms

Key Points

- Match the number of runnable software threads to the available hardware threads
- Synchronization : In correct Answers ; Performance Issues
- Keeps Locks private
- Avoid dead-locks by acquiring locks in a consistent order
- Memory Bandwidth & contention Issues
- Lock contention (Using Multiple distributed locks)
- Design Lockless Algorithms – Advantages & dis-advantages
- Cache lines are – Hardware threads
- Writing synchronized code – Memory Consistency

Conclusions

- Important issues in Shared parallel programming -Pthreads
- Common Synchronization problems with Pthreads
- Pthreads Performance issues on Multi Core Processors

References

1. Andrews, Grogory R. **(2000),** Foundations of Multithreaded, Parallel, and Distributed Programming, Boston, MA : Addison-Wesley

2. Butenhof, David R **(1997),** Programming with POSIX Threads , Boston, MA : Addison Wesley Professional

3. Culler, David E., Jaswinder Pal Singh **(1999),** Parallel Computer Architecture - A Hardware/Software Approach , San Francsico, CA : Morgan Kaufmann

4. Grama Ananth, Anshul Gupts, George Karypis and Vipin Kumar **(2003),** Introduction to Parallel computing, Boston, MA : Addison-Wesley

5. Intel Corporation, **(2003),** Intel Hyper-Threading Technology, Technical User's Guide, Santa Clara CA : Intel Corporation Available at : http://www.intel.com

6. Shameem Akhter, Jason Roberts **(April 2006),** Multi-Core Programming - Increasing Performance through Software Multi-threading , Intel PRESS, Intel Corporation,

7. Bradford Nichols, Dick Buttlar and Jacqueline Proulx Farrell **(1996),** Pthread Programming O'Reilly and Associates, Newton, MA 02164,

8. James Reinders, Intel Threading Building Blocks – (**2007**) , O'REILLY series

9. Laurence T Yang & Minyi Guo (Editors), (**2006**) *High Performance Computing - Paradigm and Infrastructure* Wiley Series on Parallel and Distributed computing, Albert Y. Zomaya, Series Editor

10. Intel Threading Methodology ; Principles and Practices Version 2.0 copy right **(March 2003),** Intel Corporation

References

11. William Gropp, Ewing Lusk, Rajeev Thakur **(1999),** Using MPI-2, Advanced Features of the Message-Passing Interface, The MIT Press..
12. Pacheco S. Peter, **(1992),** Parallel Programming with MPI, , University of Sanfrancisco, Morgan Kaufman Publishers, Inc., Sanfrancisco, California
13. Kai Hwang, Zhiwei Xu, (**1998**), Scalable Parallel Computing (Technology Architecture Programming), McGraw Hill New York.
14. Michael J. Quinn (**2004**), Parallel Programming in C with MPI and OpenMP McGraw-Hill International Editions, Computer Science Series, McGraw-Hill, Inc. Newyork
15. Andrews, Grogory R. **(2000),** Foundations of Multithreaded, Parallel, and Distributed Progrmaming, Boston, MA : Addison-Wesley
16. SunSoft. Solaris multithreaded programming guide. SunSoft Press, Mountainview, CA, **(1996),** Zomaya, editor. Parallel and Distributed Computing Handbook. McGraw-Hill,
17. Chandra, Rohit, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon, **(2001)**,Parallel Programming in OpenMP San Fracncisco Moraan Kaufmann
18. S.Kieriman, D.Shah, and B.Smaalders **(1995),** Programming with Threads, SunSoft Press, Mountainview, CA. 1995
19. Mattson Tim, **(2002),** Nuts and Bolts of multi-threaded Programming Santa Clara, CA : Intel Corporation, Available at : http://www.intel.com
20. I. Foster **(1995,** Designing and Building Parallel Programs ; Concepts and tools for Parallel Software Engineering, Addison-Wesley (1995)
21. J.Dongarra, I.S. Duff, D. Sorensen, and H.V.Vorst **(1999),** Numerical Linear Algebra for High Performance Computers (Software, Environments, Tools) SIAM, 1999

References

22. OpenMP C and C++ Application Program Interface, Version 1.0". **(1998),** OpenMP Architecture Review Board. October 1998

23. D. A. Lewine. *Posix Programmer's Guide:* **(1991),** Writing Portable Unix Programs with the Posix. 1 Standard. O'Reilly & Associates, 1991

24. Emery D. Berger, Kathryn S McKinley, Robert D Blumofe, Paul R.Wilson, *Hoard : A Scalable Memory Allocator for Multi-threaded Applications* ; The Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX). Cambridge, MA, November (**2000**). Web site URL : http://www.hoard.org/

25. Marc Snir, Steve Otto, Steyen Huss-Lederman, David Walker and Jack Dongarra, (**1998**) *MPI-The Complete Reference: Volume 1, The MPI Core, second edition* [MCMPI-07].

26. William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir (**1998**) *MPI-The Complete Reference: Volume 2, The MPI-2 Extensions*

27. A. Zomaya, editor. Parallel and Distributed Computing Handbook. McGraw-Hill, **(1996)**

28. OpenMP C and C++ Application Program Interface, Version 2.5 (**May 2005**)", From the OpenMP web site, URL **: http://www.openmp.org/**

29. Stokes, Jon 2002 Introduction to Multithreading, Super-threading and Hyper threading *Ars Technica*, October **(2002)**

30. Andrews Gregory R. 2000, Foundations of Multi-threaded, Parallel and Distributed Programming, Boston MA : Addison – Wesley (**2000)**

31. Deborah T. Marr , Frank Binns, David L. Hill, Glenn Hinton, David A Koufaty, J . Alan Miller, Michael Upton, "Hyperthreading, Technology Architecture and Microarchitecture", Intel (**2000-01)**

Thank You

Any questions ?