CSCI 551
Numerical and Parallel Programming

Lecture 6-1 – Shared Memory vs. Distributed Memory Parallel Programming – Accuracy, Precision and Sources of Error

Major Goals Today

- Ex #2 Grading in Progress (Return on or before Thursday)
- Take Quiz #2 BEFORE Class next Tuesday!
- Ex #3 - Turn in First MPI Program with Numerical Integration Saturday
- Wrap up introduction to MPI and Methods of Integration
  - Introduction to Accuracy, Precision, and Sources of Error
  - Details of integration methods
- Activity - Questions on Exercise #3
- Review for Exam #1 Thursday 3/7 to Tues 3/12
- **Exam #1 on Next Week on Thursday March before Spring Break in-person**
  - One Page of Notes for Part-1 Knowledge and Concepts (OCNL 254)
  - Part-2 - You may login to ECC-Linux to test code you write or code you analyze in OCNL 244
  - Exam will be completed on Canvas
    - Knowledge and Concepts (done in Discussion)
    - Code, design, analysis (done in Activity)

Accuracy & Precision (Sources of Error)

- Accuracy – How close am I to a target (truth)?
- Precision – No matter what my accuracy is, how many digits in a numeric answer can be trusted?
- Similar to a game of darts
  - Accuracy - distance from the bull's eye (target)
  - Precision - how tightly grouped the darts are in a location (near target or not)
  - Error – skill in throwing (Azimuth, Elevation and distance estimation)

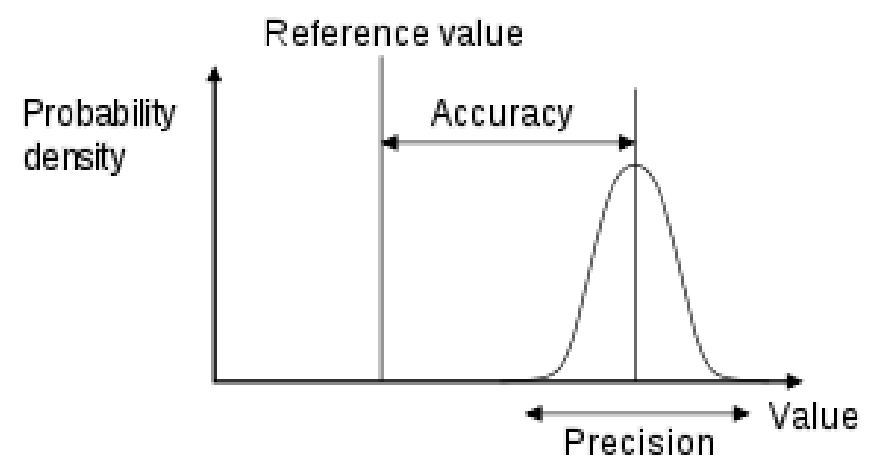
Float: 7 digits of precision
Double: 15 digits of precision

High precision bristle board
real numbers (unlimited precision)

Limited precision pegs - discrete

Difference between Accuracy and Precision

- How well can we represent a specific numeric value – digits of precision
- How close are we to a reference or target?
- While they are different concepts, the accuracy and precision together are necessary and sufficient for exactness
- E.g. hitting a target defined with a specific precision



Wikipedia: Accuracy_and_precision

Analog Systems – Continuous real valued

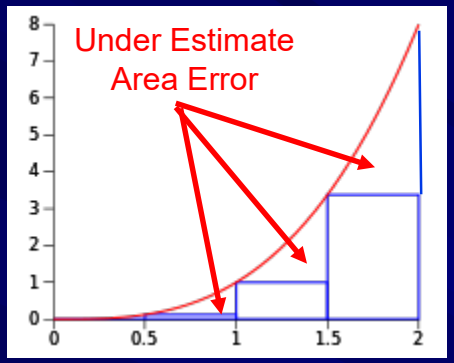
Digital Systems – Discrete values (binary)

Fixed point – whole and fractional numbers with fixed range and precision

Floating point – emulation of real numbers with adaptive range and precision (mantissa and exponent)

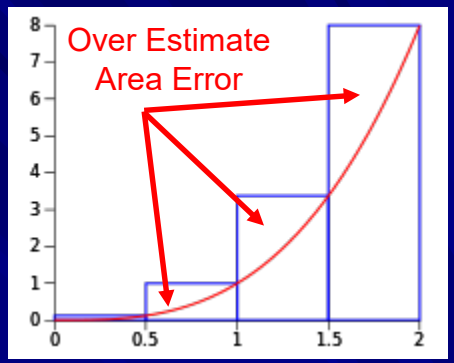
Integration Error (Riemann Sum)

Left Riemann Sum (Forward)
A_n= f(a) dx
dx=[b-a]/n



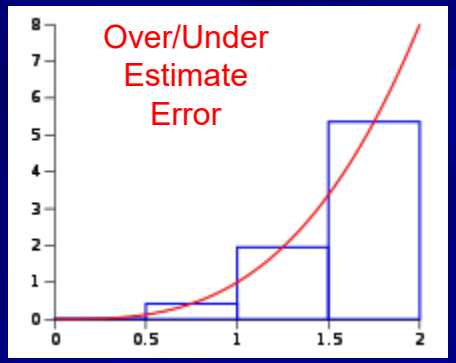
A_{b-a} = dx [(f(a) + f(a+dx) + f(a+2dx) + … + f(b-dx)]

Right Riemann Sum (Backward)
A_n= f(a+dx) dx
dx=[b-a]/n



A_{b-a} = dx [f(a+dx) + f(a+2dx) + … + f(b)]

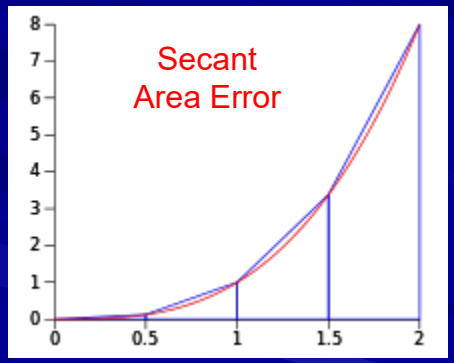
Midpoint Rule
A_n= f(a+dx/2) dx
dx=[b-a]/n



A_{b-a} = dx [f(a+dx/2) + f(a+3dx/2) + … + f(b-dx/2)]

Trapezoidal Rule

A_{trap}=½ h (f(a)+f(b))
dx=[b-a]/n
A_n=½ dx (f(a)+ f(a+dx))



½ dx [(f(a) + f(a+dx)) + (f(a+dx) + f(a+2dx)) …]
½ dx [f(a) + 2f(a+dx) + 2f(a+2dx) + … f(b)]

Error comes from method of approximation

Note: impact of function – convex or concave on accuracy of result

estimate = estimate*base_len;

MPI Code for Trapezoidal

- Initial estimate

  ½ dx [**f(a)** + 2f(a+dx) + 2f(a+2dx) + … **f(b)**]

```
estimate = (funct_to_integrate(left_endpt)
+ funct_to_integrate(right_endpt))/2.0;
```

- Add intermediate values

  ½ dx [f(a) + **2f(a+dx) + 2f(a+2dx) + …** f(b)]

```
x = left_endpt + i*base_len;
estimate += funct_to_integrate(x);
```

- Scale by dx at the end and return

  ½ **dx** [f(a) + 2f(a+dx) + 2f(a+2dx) + … f(b)]

  ½ dx [f(a) + 2f(a+dx) + 2f(a+2dx) + … f(b)]

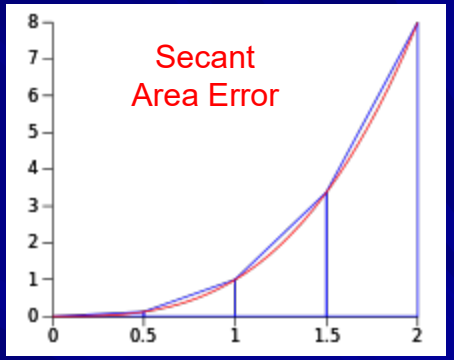
```
estimate = estimate*base_len;
return estimate;
```

Trapezoidal Rule

A_{trap}=½ h (f(a)+f(b))

dx=[b-a]/n

A_n=½ dx (f(a)+ f(a+dx))



½ dx [(f(a) + f(a+dx)) + (f(a+dx) + f(a+2dx)) …]

Add common terms together …

½ dx [f(a) + 2f(a+dx) + 2f(a+2dx) + … f(b)]

estimate = estimate*base_len;

MPI Code for Left Riemann

- Initial estimate

  dx [(**f(a)** + f(a+dx) + f(a+2dx) + … + f(b-dx)]

```
left_value = funct_to_integrate(left_endpt);
x = left_endpt;
```

- Add intermediate values and set next

  dx [(**0 + f(a)** + **f(a+dx)** + f(a+2dx) + … + f(b-dx)]

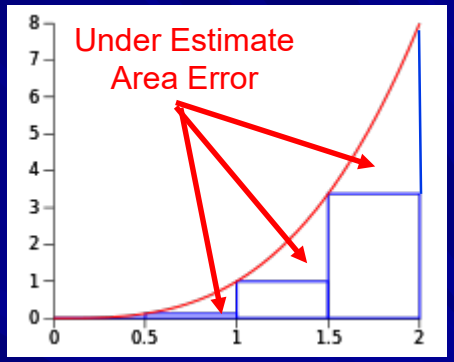
```
area += left_value * base_len;
x += base_len;
left_value = funct_to_integrate(x);
```

- Return the summed area

  dx [(f(a) + f(a+dx) + f(a+2dx) + … + f(b-dx)]

```
Return area;
```

Left Riemann Sum
(Forward)
A_n= f(a) dx
dx=[b-a]/n



A_{b-a} = dx [(f(a) + f(a+dx) + f(a+2dx) + … + f(b-dx)]

MPI Collective Comm vs. Point-to-Point

- Collective Comm abstracts loop-based MPI_Send, MPI_Recv (complex)
  - csci551/code/hello_cluster/greetings.cpp
  - csci551/code/hello_cluster/ranksumtree.c
  - csci551/code/hello_cluster/ranksumfan.c
  - csci551/code/hello_cluster/ranksumbutterfly.c
  - csci551/code/hello_cluster/compare.c (**Intermediate!**)

- MPI_Reduce and MPI_Allreduce simplify MPI code
  - Helpful for integration with multiple ranks
  - MPI_Reduce and MPI_Allreduce programs that sum up the digits for rank #
    - csci551/code/hello_cluster/ranksum.c
    - csci551/code/hello_cluster/ranksumall.c
  - Simplified by using collective comm as:
    - csci551/code/hello_cluster/ranksumreduce.c
    - csci551/code/hello_cluster/ranksumallreduce.c
  - Note how they replace a bunch of MPI_Send, MPI_Rcv code with the one liner!

- MPI scatter, gather, allscatter, and allgather, simplify vector/matrix code
  - csci551/code/hello_cluster/rankscattergather.c

- Pacheco collective xomm examples (csci551/code/MPI_Examples/)
  - csci551/code/MPI_Examples/mpi_trap4.c
  - csci551/code/MPI_Examples/vector_add.c
  - csci551/code/MPI_Examples/mpi_mat_vect_mult.c
  - csci551/code/MPI_Examples/mpi_mat_vect_time.c
  - csci551/code/MPI_Examples/mpi_many_msgs.c

Point-to-Point Send/Recv
- Low level message passing
- Harder to debug
- MPI_Barrier can help
- Coded examples (book)

Collective Comm
- Simplifies
- Adds useful operators!

Rank 0 Mgr or Worker?
- Only rank that can prompt for user input
- Can be a pure manager doing mapping and reduction only
- Or use map & reduce and be a worker too!

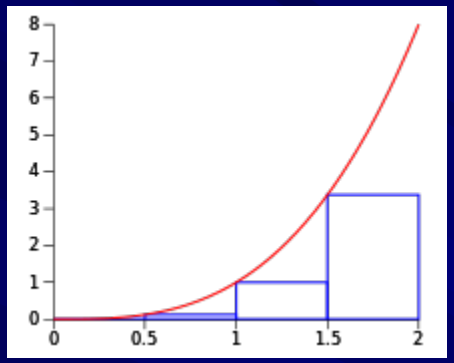
Concept of Reduce – Parallel Operator

Review - Exercise #3 Train Problem

- Note that Acceleration is a smooth non-linear sine function
  - Create a function for acceleration – mathematical (antiderivative)
  - Create a look-up table for acceleration – interpolation (Ex #4)
- Note that the anti-derivatives of sine and cosine are known
- Spreadsheet model adjusts acceleration for 122 Km goal
- Exploit the ability to make acceleration a function
  - Note that Ex #3 has functions for which “we know” antiderivative
  - Check this with Wolfram Alpha, Symbolab, or similar
  - Integrate using numerical methods to learn and because this will work for any function!
  - We may not always have well defined functions easily modeled with math
- Train should arrive at station 122 Km at 0 velocity with your method of integration (consider accuracy, precision, and error)

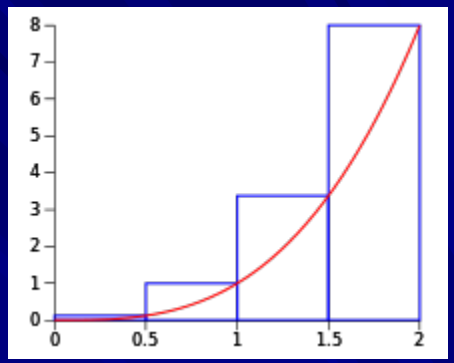
Recall Integration (Riemann Sum)

Left Riemann Sum (Forward)
A_n= f(a) dx
dx=[b-a]/n



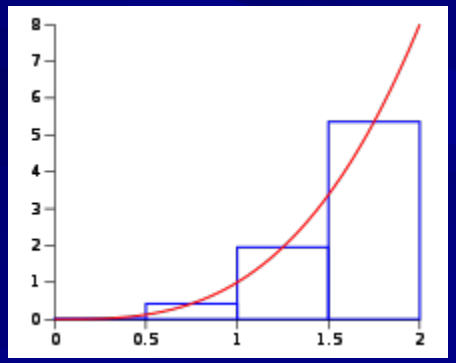
A_{b-a} = dx [(f(a) + f(a+dx) + f(a+2dx) + … + f(b-dx)]

Right Riemann Sum (Backward)
A_n= f(a+dx) dx
dx=[b-a]/n



A_{b-a} = dx [f(a+dx) + f(a+2dx) + … + f(b)]

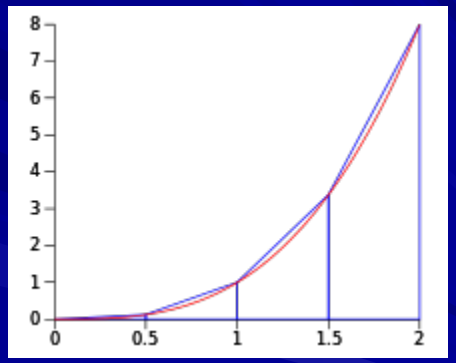
Midpoint Rule
A_n= f(a+dx/2) dx
dx=[b-a]/n



A_{b-a} = dx [f(a+dx/2) + f(a+3dx/2) + … + f(b-dx/2)]

Trapezoidal Rule

A_{trap}=½ h (f(a)+f(b))
dx=[b-a]/n
A_n=½ dx (f(a)+ f(a+dx))



½ dx [(f(a) + f(a+dx)) + (f(a+dx) + f(a+2dx)) …]
½ dx [f(a) + 2f(a+dx) + 2f(a+2dx) + … f(b)]

Note: impact of function – convex or concave on accuracy of result

Exercise #3 – Scaling?

- MPI is a trade-off
  - Distributed scaling is almost limitless (size of mesh only limit), but higher overhead than shared memory
  - Efficiency of message-passing methods, distribution of work and aggregation of results is critical!

- MPI Benchmarks (code/mpi-benchmarks-master-csu/)
  1. Be skeptical - what does this really show me, if anything?
  2. Does this show that MPI is efficient?
  3. Be optimistic, but realistic – gigabit ethernet is fast, 10GE faster!
  4. When does performance start to tail-off - # or size of messages
  5. Does it indicate overhead of MPI?
  6. Does it indicate scaling or scaling efficiency? [e.g. linear?]
  7. Which “MPI_” API functions does it employ?

MPI Distributed vs. Shared Memory Parallel (Pthread, OpenMP)

Which is better?

MPI vs. Shared Memory Threading

- MPI advantages
  - Scales as big as your interconnection network (in theory infinite)
  - Does not require messy MUTEX locks and shared resource sync
  - Message passing and collective comm are great abstractions
  - Standard with proprietary and open source tools
- Threading Advantages
  - UMA fast access to memory and data with no messaging overhead
  - Thread safe methods such as thread indexed and stack are fast
  - Threads can use message passing (e.g., POSIX mqueues)
  - Standard with proprietary and open source tools

Discussion Review – Process & Thread

- What is the difference between?
  - Process (MPI)
  - Task
  - Thread (POSIX Pthreads and ILP threading by compiler for OpenMP)
- Process and Thread (user space)
  - Increment/decrement with semaphores – incdecthread_sem
  - Twoprocs
  - SMT is hardware support for thread execution context (helps)
- Task – RTOS or Linux Kernel
  - E.g., VxWorks task code - csci551/code/two_tasks.c
- Think about examples, address space, execution context, context switching, sharing data, and sharing code
- Note that "service" and "task" are often a theoretical description of an execution context rather than concrete implementation, whereas process and thread are typically implementations

Discussion Review – Pthread & MPI

- Pthreads - Coarse, software parallelism for shared memory (uses library, OS supported)
  - Challenge – detailed API (e.g., "man –k pthread")
  - Synchronization is low level (semaphores, locks, barriers, etc.)
  - Can be SMP (OS determines CPU core to use)
  - Can be AMP (application selects CPU core to use)
  - Thread executes inside a Linux user-space process
  - Issues such as deadlock, livelock, data corruption, thread-safety in general

OS experience is a big help for Pthreads

- User space
- Not kernel
- Not driver
- Concurrency and Synch
- Many uses – concurrent, real-time,and parallel

- MPI – Coarse, software parallelism for distributed memory
  - Processes are the execution context (safer)
  - Message passing main (only) method of synchronization

Cluster knowledge

- Basic networking
- Process model

MPI vs. Pthreads vs. OpenMP

- Used for Shared Memory Parallel Processing
  - Pthreads – software parallelism, used widely for many systems
  - OpenMP – pragma-based ILP parallelism for numerical methods
- MPI – Distributed Memory Parallel Processing
- OpenMP Shared Memory Parallel vs. Pthreads
  - OpenMP abstracted compared to lower-level Pthreads
    - More focus on numerical methods with MPI and OpenMP
    - Course satisfies math requirements
      - Linear systems & Non-linear systems
      - Precision
      - Root solving
    - Minimal additional coverage on Pthreads after Exam #1
    - Use R-Pi for OpenMP - parallel-computing-in-embedded-mobile-devices/
    - Must use ECC and cluster for MPI
- Objective for Pthreads - fundamentals and solutions
  - Create, join, binary semaphore, MUTEX lock
  - Embarrassingly parallel shared memory problems (e.g., PSF convolution, DCT)
- Introduction to MPI basics before Exam #1
- Advanced MPI – harder to divide and conquer, hybrid use with OpenMP

OpenMP

- High-level, coarse-grained, shared memory, parallel programming
- Use instead of Pthreads?
- In addition?

After Exam #1 – Advanced MPI and hybrid use of MPI + OpenMP

Things that make Parallel Harder?

- Data and computational dependencies – e.g., simulation and integration of state to get next state (loop carried dependency)
- Shared memory with multiple readers and writers (**can be complex**)
  - MUTEX (e.g., example-mutex-demos/)
  - Producer/Consumer (e.g., Writer-Reader-sync-demo/)
  - **Thread indexed data and use of stack vs. global memory – keep it simple!**
- General Cases Where Parallel Helps
  - **High degree** (polynomial bounded, but complex) – NC, P-complete
  - **High dimensionality** – 3D and beyond
    - 2D Video – X,Y, color, time (4 dimensions)
    - 3D Graphics
    - Simulation of physical systems (e.g., 6 DOF robotics)
    - Data analytics
  - **Non-Linear and Linear Systems** (many equations and unknowns)
  - **High Fidelity, Resolution** and Long Duration (faster than real-time)
  - **Large search spaces**
  - **High complexity** – High degree P and NP-hard problems (e.g., cryptanalysis)

Methods of Shared Memory Synchronization

- **For Pthreads, most often use sem_init and pthread_mutex_init**
- Barriers, condition variables, spin locks, signals, and message queues

Platform	Operating Environment	Semaphores	Locks	Other (e.g., barriers, spin lock)	Composite
CE main + ISR	Microcontroller (no OS)	ARM ASM[1]	TSL	busy wait Interrupt vector, ISR	none
OS	Linux	sem_t, sem_init (sem_post, sem_wait)	pthread_mutex_init pthread_mutex_lock, pthread_mutex_unlock omp atomic statement; omp critical {<block>} omp_lock_t, omp_init_lock, omp_test_lock, omp_unset_lock	pthread_join pthread_barrier_init pthread_spin_lock POSIX signals sigqueue, sigaction OpenMP implicit barrier at end of structured block	pthread_cond_init
RTOS	VxWorks	semLib[2]: semBCreate semMCreate* semCCreate (semTake, semGive)	pthreadLib VxWorks 6.x AMP and earlier[3] taskLock intLock	SMP[4] barriers spin locks message queues signals	condVarLib pthreadLib

* semMCreate is a MUTEX, created and used like a semaphore, but most often MUTEX is considered a lock – has FIFO, priority, and inversion safe (priority inheritance)

1) https://developer.arm.com/documentation/dht0008/a/arm-synchronization-primitives/practical-uses/implementing-a-semaphore
2) https://docs.windriver.com/bundle/vxworks_7_application_core_os_sr0630-enus/page/CORE/aioPxLib.html
3) Wind River VxWorks Kernel Programmer's Guide 6.2
4) https://learning.windriver.com/vxworks-7-smp-programming-features
5) Gallmeister, Bill. *POSIX. 4 Programmers Guide: Programming for the real world.* " O'Reilly Media, Inc.", 1995.
6) Stevens, W. Richard, and Stephen A. Rago. *Advanced programming in the UNIX environment*. Addison-Wesley, 2008.
7) POSIX Standards: https://publications.opengroup.org/, http://get.posixcertified.ieee.org/

MPI largely side-steps by using message passing

Methods of Synchronization – Brief History

- Semaphore – Edsger Dijkstra (1962 Electrologica X8)
  - P=*prolaag* (“try to reduce") or take, V=*verhogen* (“increase") or give
  - Original Dutch paper
  - Binary semaphore – P=wait, V=signal has value 0 or 1
    - P(0) causes wait, P(1) does not
    - P=wait is s.t. it decrements S by 1, blocks caller if S < 0, otherwise S := S-1 and caller continues
    - V=signal is s.t. it increments S by 1, and blocked callers are unblocked from S wait FIFO
  - Counting semaphore
    - V(sem S, int I) {atomic(S := S + I)}
    - P(sem S, int I) {repeat {atomic(if S >= I then {S := S – I; break})}}
    - Semaphore can count up to “n”

- MUTEX
  - A lock for a critical section, unique to each section created (like hotel room or gas station bathroom key) - constructed using atomic Test-and-Set-Lock (TSL) or binary semaphore
  - As a result, sometimes a MUTEX is referred to as a semaphore, but really is more than a semaphore
  - Used to prevent shared memory data corruption (multiple readers and writers) and enforce one-at-time use of any resource
  - Clearly defines scope and use – enables policies for unblocking such as FIFO order, scheduling priority order, priority inheritance or priority ceiling

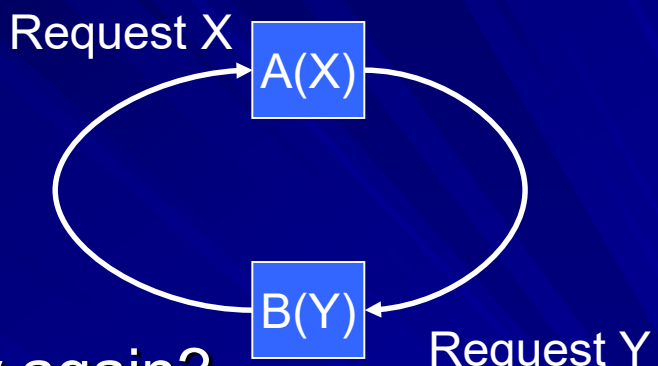
- Spin Locks – busy wait loop checks for a condition (e.g., data ready) consuming CPU (ok for multi-core)

- Atomic – single variable (memory location) made critical for indivisible operations, e.g., x = x + 1; (read x, modify, write)

- Condition Variable
  - waiting on condition with spin (wastes CPU) or delay and check in a loop (better) – common busy wait method used in simple systems (executive or RTOS)
  - best is to signal thread(s) when a condition is true – OS supported condition variable

- Monitor – Per Brinch Hansen and C.A.R. Hoare (1970’s in Simula)
  - MUTEX and condition variable allowing for threads to wait for a condition
  - a thread safe object, class or module with MUTEX methods

Resource Deadlock (Circular Wait)



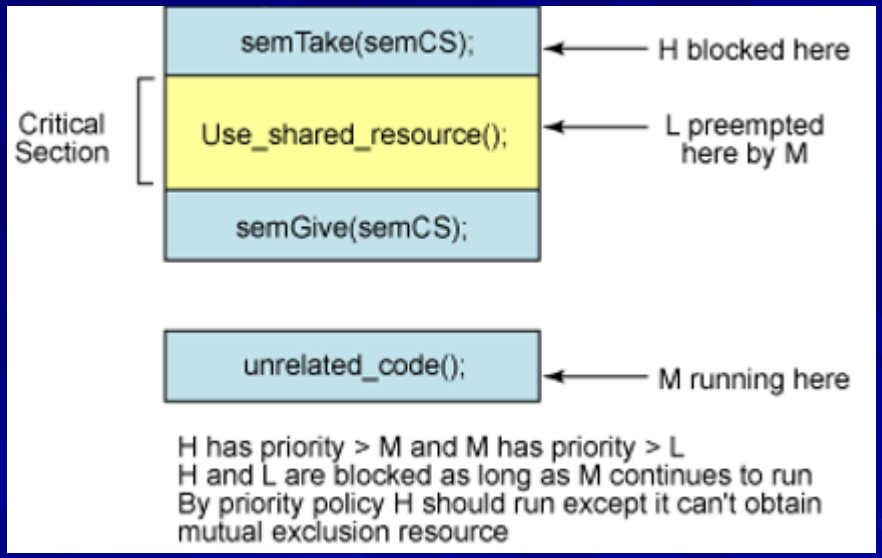
- A is holding X and would like Y
- B is holding Y and would like X
- How is this resolved?
- A and B could Block Indefinitely
- Each could release X or Y and try again?
  - Can Result in Livelock
  - They Release, A grabs X, B grabs Y, Deadlock, Detection, Release, A grabs X, B grabs Y …
  - Random Back-Off Times (Possible Solution)
- Circular Wait Can Evolve over Complex Sets of Tasks and Resources (Hard to Detect or Prevent)
- This is Unbounded Blocking
- Detection Most Often with Watch-Dog and Sanity Monitors

Shared Resource Mutually Exclusive Access

- Shared Mutex Resources Require Protection from Unintentional Non-Mutex Access
- E.g., Shared Memory State that Can't Be Updated Atomically
  - Position = {x, y, z}
  - To Update Position Requires More Than One Write Instruction
  - What Happens on Interrupt Between Update of X and Y?
  - What if That Interrupt Releases a Service that Reads Position?
  - Data Corruption!

Fair Scheduling simplest for MUTEX

H, M, L are 3 Threads Sharing Memory



H and L Share the Resource Position

- Position is a struct {x, y, z} position vector
- H must update and L must read and use
- MUTEX allows for safe use of {x, y, z}
- Priority complicates, so "fair" equal priority threads are best!

Synchronization for CSCI 551

- Most often, divide up work, compute in parallel, gather results (rank 0, main thread, automatic with OpenMP) and report them (avoid locks)

- Don't worry about Atomic, MUTEX or Sync unless or until …
  1. You need to summarize overall results (e.g., rank 0 MPI program computes final value) – use MPI methods (see hello cluster examples)
  2. Unless **you have a complex data structure with read/write threads that must share it** and you have a shared memory parallel program (OpenMP or Pthreads)
  3. You need to join threads after computation with thread specific (thread indexed) data, stack (thread parameters), or MUTEX protected shared memory
  4. Unless you have multiple processes (MPI) or threads (OpenMP and Pthreads) and they have multiple resources they must acquire and share (conditions for deadlock/livelock) – starvation

- Test programs and beware of potential for data corruption and deadlock/livelock… avoid if possible!