Week 6 Discussion Questions
Chapter 4.1 – 4.6:

1. When threads update a shared global why is cache coherency not enough to guarantee correct results?

2. Why is implementing a critical section using a mutex preferred over using busy-waiting?

3. Consider the following two critical section implementations:

Option A:

```
for (i=firstIndex; i <= lastIndex; i++) {
  pthread_mutex_lock(&updateLock);
  update shared variable
  pthread_mutex_unlock(&updateLock);
} /* end for (i */
```

Option B:

```
pthread_mutex_lock(&updateLock);
for (i=firstIndex; i <= lastIndex; i++) {
  update shared variable
} /* end for (i */
pthread_mutex_unlock(&updateLock);
```

Assuming all but the shared variable are local to the thread.

a) Are both options correct (i.e., give the correct shared variable)?

b) If they are both correct, then which option would be better and why?

4. Suppose we want to perform matrix (2D arrays) addition: C = A + B where each matrix has dimensions: rows x columns. Sequentially the code would be:

```
for (r=0; r < rows; r++) {
   for (c=0; c < columns; c++) {
      C[r][c] = A[r][c] + B[r][c];
   } /* end for (c */
} /* end for (r */
```

Assuming we are using 4 pthreads in the programming language C. How should the work (i.e., calculating elements of the C matrix) be allocated to each thread to make the parallel code most efficient?

Learning Objectives:
- Apply pthread library commands to initialize (create) threads
- Apply mutexes to create critical sections
- Compare different data-decomposition techniques (i.e., block vs. cyclic) on performance.

To start the lab:
- watch the Lab 6 Video on the eLearning system
- download `lab6.zip` from the eLearning system and unzip/extract it locally on your computer
- copy the `lab6` directory to `student.cs.uni.edu` using a secure ftp client (winSCP, FileZilla, scp, etc.)
- log-on to `student.cs.uni.edu` using Putty/ssh

Part A: Using an editor on `student.cs.uni.edu` open the file `sumNoSync.c` which contains a C program that allows the user to enter two integer command-line arguments: the number of values and the number of threads. It creates a 1-dimensional array of that size, fills the array with random floating-point numbers, and uses the specified number of pthreads to sum the array values. WARNING: the pthreads can generate an incorrect sum, but the correct sum is also calculated sequentially.

Compile the program using NO optimization (i.e., compiler option -O0):

```
gcc -o sumArray -O0 sumNoSync.c -lpthread
```

and run the program <u>repeatedly until the thread's sum differs from the sequential sum</u>. (REMEMBER: you can use the up-arrow, ↑, key to recall the last command-line command. Use the command-line of:

```
sumArray 10000 4
```

a) What different values did you get for both:
Thread's sum:

Seq. sum:

b) What do notice that's strange about the difference between the two sums? (Explain the difference.)

c) Examine at the `threadPartialSum` function. What's wrong about each thread incrementing the globalSum (i.e., `globalSum += myArray[i];`) without any synchronization with the other threads?

Part B: Using an editor open the file `sumNoSyncNeeded.c` which contains a similar program, but correctly calculates the array sum using pthreads. In this program thread synchronization is not needed because each thread updates their own index in the threadSums array, i.e., thread 0 updates threadSum[0], etc.

a) After all the threads sum their chuck of the array into their slot of the threadSum array, what does the main function do to compute the overall globalSum?

b) How does the main function "know" when all threads have finished computing their partial sums?

Consider the `threadPartialSum` function that each thread runs. It is defined as:

```
void * threadPartialSum(void * rank) {

  long myRank = (long) rank;
  long i, blockSize;
  long firstIndex, lastIndex;

  blockSize = length / numberOfThreads;
  firstIndex = blockSize * myRank;
  if (myRank == numberOfThreads-1) { // last thread gets the rest
    lastIndex = length-1;
  } else {
    lastIndex = blockSize * (myRank+1) - 1;
  } // end if

  threadSums[myRank] = 0.0;
  for (i=firstIndex; i <= lastIndex; i++) {
    threadSums[myRank] += myArray[i];
  } /* end for (i */

  return NULL;
} // end threadPartialSum
```

a) What range of `rank` parameters is sent to the threads when you use the command-line of: `sumArray 10000 4`?

b) How do threads gain access to other variables (i.e., `length`, `numberOfThreads`, `myArray`, etc.)?

c) Explain the calculation of the `firstIndex` and `lastIndex` variables.

```
firstIndex = blockSize * myRank;
```

```
lastIndex = blockSize * (myRank+1) - 1;
```

d) What is the purpose of the `if`-statement in the calculation of the `lastIndex` variable?

e) Explain how `threadSums` could cause *false sharing* if the threads run on cores with separate caches.

Part C: Using an editor open the file `sumMutex1.c` which contains a similar program which also correctly calculates the array sum using pthreads. In this program threads synchronize their updating of the `globalSum` variable in the `threadPartialSum` function by a mutex controlled critical section:

```
for (i=firstIndex; i <= lastIndex; i++) {
  pthread_mutex_lock(&updateSumLock);
  globalSum += myArray[i];
  pthread_mutex_unlock(&updateSumLock);
} /* end for (i */
```

a) If thread 2 is updating `globalSum` and thread 0 executes the `pthread_mutex_lock` function, what happens to thread 0?

Consider the following alternative code for the `threadPartialSum` critical section:

```
pthread_mutex_lock(&updateSumLock);
for (i=firstIndex; i <= lastIndex; i++) {
  globalSum += myArray[i];
} /* end for (i */
pthread_mutex_unlock(&updateSumLock);
```

b) Would this alternative code calculate the correct `globalSum` value?

c) How does the efficiency of this alternative code compare to the original critical section code? (Explain you answer)

Part D: Using an editor open the file `sumMutex2.c` which contains a similar program which also correctly calculates the array sum using pthreads. In this program threads synchronize their updating of the `globalSum` variable in the `threadPartialSum` function by:

```
void * threadPartialSum(void * rank) {

  double localSum;

  <CODE OMITTED HERE>

  localSum = 0.0;
  for (i=firstIndex; i <= lastIndex; i++) {
    localSum += myArray[i];
  } /* end for (i */

  pthread_mutex_lock(&updateSumLock);
  globalSum += localSum;
  pthread_mutex_unlock(&updateSumLock);

  return NULL;
} // end threadPartialSum
```

a) How many times does each thread need to lock and unlock the critical section?

b) How "long" (# of statements) does each thread need to keep the critical section locked?

c) Unlike the `sumNoSyncNeeded.c` program which had *false sharing* with respect to the `threadSums` array, why does each thread updating of their `localSum` (i.e., `localSum += myArray[i];`) not cause *false sharing*?

Part E: Using an editor open the file `sumMutex3.c` which contains a similar program which also correctly calculates the array sum using pthreads. In this program threads synchronize their updating of the `globalSum` variable in the `threadPartialSum` function by:

```
void * threadPartialSum(void * rank) {

  long myRank = (long) rank;
  long i, stride, firstIndex;
  double localSum;

  stride = numberOfThreads;
  firstIndex = myRank;

  localSum = 0.0;
  for (i=firstIndex; i < length; i = i + stride) {
    localSum += myArray[i];
  } /* end for (i */

  pthread_mutex_lock(&updateSumLock);
  globalSum += localSum;
  pthread_mutex_unlock(&updateSumLock);

  return NULL;
} // end threadPartialSum
```

a) When you use the command-line of: `sumArray 10000 4` what are the first 5 `i`-index values for each thread?

`myRank` value of 0:

`myRank` value of 1:

`myRank` value of 2:

`myRank` value of 3:

b) Why would `sumMutex3.c`'s cyclic division of work among the threads cause poorer cache performance (i.e., more misses) than `sumMutex2.c`'s block division of work?

Submit lab6.zip containing question answers and completed programs on the eLearning system

Learning Objectives:

- Design an efficient data-decomposition (i.e., block vs. cyclic) for a pthread program in C.
- Write correct C program using pthread library commands to initialize (create) pthreads and synchronize their operation.
- Time various sizes and # of threads

Homework #6 Description:

You are to design and write a C program utilizing pthreads to efficiently perform matrix multiplication. To start the homework, download and extract `hw6.zip` from the eLearning system. It contains a copy of the `lab5/multSeqOptions.c` program which you can use as your starting point.

Your program must satisfy the following specifications:

- can handle square matrices A and B
- fill matrices A and B with randomly generated doubles in the range -1.0 and +1.0
- checks the results of your parallel product matrix by comparing it against a sequentially calculated (`matrixMultiplicationAlt`) matrix. Each element of the two products should be within a tolerance of 0.000001
- time both the parallel and sequential matrix multiplications -- second granularity as done in multSeqOptions.c

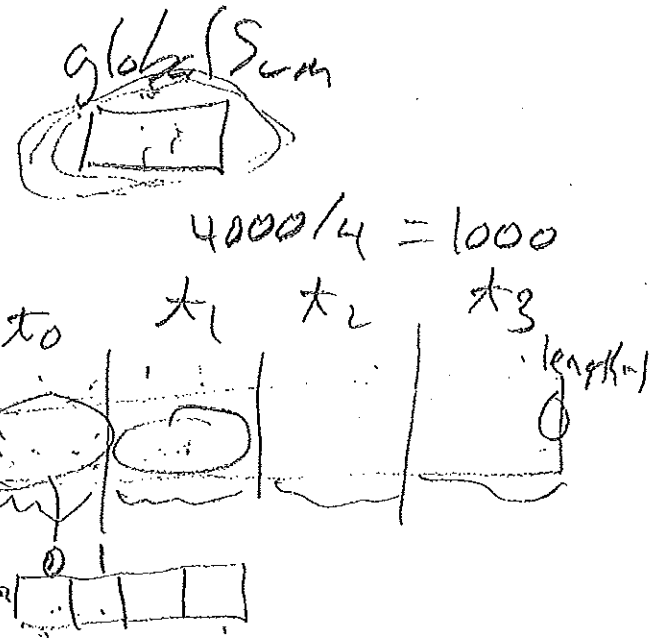
Compile (`gcc -o mmult -O3 mmultHW6.c -lpthread`) and run the program varying the matrix size and number of threads to complete the following table. **You can develop your program using student.cs.uni.edu, but I would like you to complete the timings on fermi1.cs.uni.edu.**

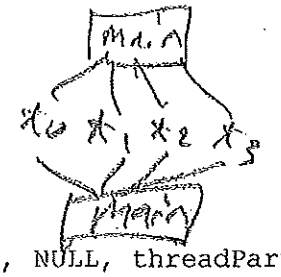
Matrix Sizes	Time of sequential `matrixMultiplicationAlt` function (in seconds)	Time of your parallel matrix multiplication function (in seconds) using a varying number of threads				
		1 thread	2 threads	4 threads	8 threads	16 threads
1500 x 1500						
2000 x 2000						
2500 x 2500						
3000 x 3000						
3500 x 3500						

Submit hw6.zip containing a completed timing table and completed program (mmultHW6.c) on the eLearning system

Week 6 Tuesday Video Notes


```
/*  Programmer:  Mark Fienup
    File:        sumNoSyncNeeded.c
    Compile As:  gcc -o sumArray -O3 sumNoSyncNeeded.c -lpthread
    Run as:  ./sumArray <# of floats to sum> <# of pthreads>
    Description:  A parallel C solution to sum an array of floats
                  using pthreads.  CORRECT solution that needs no
          thread synchronization because each updates their
          own index in the threadSums array.
*/

#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <time.h>
#include <pthread.h>

// Prototypes
void  * threadPartialSum(void * args);

// GLOBAL variables
double globalSum;
double * threadSums;
int numberOfThreads;
long length;
float * myArray;

int main(int argc, char* argv[]) {
  long i;
  pthread_t * threadHandles;
  int errorCode;
  double seqSum;
  long startTime, endTime, seqTime, parallelTime;

  if (argc != 3) {
    printf("Usage: %s <# of floats to sum> <# of threads>\n", argv[0]);
    return(0);
  };

  sscanf(argv[1],"%d",&length);
  sscanf(argv[2],"%d",&numberOfThreads);

  // Generate arrays for threads handles
  threadHandles = (pthread_t *) malloc(numberOfThreads*sizeof(pthread_t));
  threadSums = (double *) malloc(numberOfThreads*sizeof(double));

  // Generate data array
  myArray=(float *) malloc(length*sizeof(float));
  srand(5);

  for (i=0; i < length; i++) {
    myArray[i] = rand() / (float) RAND_MAX;
  } // end for i

  time(&startTime);
  seqSum = 0.0;
  for (i=0; i < length; i++) {
    seqSum += myArray[i];
  } // end for i
```






```
  time(&endTime);
  seqTime = endTime - startTime;

  time(&startTime);

  for (i=0; i < numberOfThreads; i++) {
    if (errorCode = pthread_create(&threadHandles[i], NULL, threadPartialSum,
 (void *)(i) != 0) {
      printf("pthread %d failed to be created with error code %d\n", i, errorC
ode);
    } // end if
  } // end for

  for (i=0; i < numberOfThreads; i++) {
    if (errorCode = pthread_join(threadHandles[i], (void **) NULL) != 0) {
      printf("pthread %d failed to be joined with error code %d\n", i, errorCo
de);
    } // end if
  } // end for

  // Sum up all the threadSums sequentially
  globalSum = 0.0;
  for (i=0; i < numberOfThreads; i++) {
    globalSum += threadSums[i];
  } // end for

  time(&endTime);
  parallelTime = endTime - startTime;

  printf( "Time to sum %ld floats using %d threads  %ld seconds (seq. %ld seco
nds)\n",
      length, numberOfThreads, parallelTime, seqTime);

  printf("Thread's Sum is %lf and seq. sum %lf\n\n", globalSum, seqSum);

  free(myArray);

  return 0;
} /* end main */

void * threadPartialSum(void * rank) {

  long myRank = (long) rank;
  long i, blockSize;
  long firstIndex, lastIndex;

  blockSize = length / numberOfThreads;
  firstIndex = blockSize * myRank;
  if (myRank == numberOfThreads-1) { // last thread gets the rest
    lastIndex = length-1;
  } else {
    lastIndex = blockSize * (myRank+1) - 1;
  } // end if

  threadSums[myRank] = 0.0;
  for (i=firstIndex; i <= lastIndex; i++) {
    threadSums[myRank] += myArray[i];
  } /* end for (i */
```

length=4000

1000

1000 · x 1

myRank	firstIndex	lastIndex
0	0	999
1	1000	1999
2	2000	2999
3	3000	3999

W6T-1

```
  return NULL;
} // end threadPartialSum
```

W6T3

W6T-4

```
/*  Programmer:  Mark Fienup
    File:        sumMutex1.c
    Compile As:  gcc -o sumArray -O3 sumMutex1.c -lpthread
    Run as:  ./sumArray <# of floats to sum> <# of pthreads>
    Description:  A parallel C solution to sum an array of floats
                  using pthreads.  CORRECT solution that uses a mutex
          to update the globalSum.
*/

#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <time.h>
#include <pthread.h>


// Prototypes
void  * threadPartialSum(void * args);

// GLOBAL variables
double globalSum;
int numberOfThreads;
long length;
float * myArray;
pthread_mutex_t updateSumLock;

int main(int argc, char* argv[]) {
  long i;
  pthread_t * threadHandles;
  int errorCode;
  double seqSum;
  long startTime, endTime, seqTime, parallelTime;

  if (argc != 3) {
    printf("Usage: %s <# of floats to sum> <# of threads>\n", argv[0]);
    return(0);
  };

  sscanf(argv[1],"%d",&length);
  sscanf(argv[2],"%d",&numberOfThreads);

  // Generate arrays for threads handles
  threadHandles = (pthread_t *) malloc(numberOfThreads*sizeof(pthread_t));

  // Generate data array
  myArray=(float *) malloc(length*sizeof(float));
  srand(5);

  for (i=0; i < length; i++) {
    myArray[i] = rand() / (float) RAND_MAX;
  } // end for i

  time(&startTime);
  seqSum = 0.0;
  for (i=0; i < length; i++) {
    seqSum += myArray[i];
  } // end for i
  time(&endTime);
```

```
  seqTime = endTime - startTime;

  time(&startTime);

  pthread_mutex_init(&updateSumLock, NULL);
  globalSum = 0.0;

  for (i=0; i < numberOfThreads; i++) {
    if (errorCode = pthread_create(&threadHandles[i], NULL, threadPartialSum,
(void *) i) != 0) {
      printf("pthread %d failed to be created with error code %d\n", i, errorC
ode);
    } // end if
  } // end for

  for (i=0; i < numberOfThreads; i++) {
    if (errorCode = pthread_join(threadHandles[i], (void **) NULL) != 0) {
      printf("pthread %d failed to be joined with error code %d\n", i, errorCo
de);
    } // end if
  } // end for

  time(&endTime);
  parallelTime = endTime - startTime;

  printf( "Time to sum %ld floats using %d threads  %ld seconds (seq. %ld seco
nds)\n", length, numberOfThreads,
      parallelTime, seqTime);

  printf("Thread's Sum is %lf and seq. sum %lf\n\n", globalSum, seqSum);

  free(myArray);

  return 0;
} /* end main */

void * threadPartialSum(void * rank) {

  long myRank = (long) rank;
  long i, blockSize;
  long firstIndex, lastIndex;

  blockSize = length / numberOfThreads;
  firstIndex = blockSize * myRank;
  if (myRank == numberOfThreads-1) { // last thread gets the rest
    lastIndex = length-1;
  } else {
    lastIndex = blockSize * (myRank+1) - 1;
  } // end if

  for (i=firstIndex; i <= lastIndex; i++) {
    pthread_mutex_lock(&updateSumLock);
    globalSum += myArray[i];
    pthread_mutex_unlock(&updateSumLock);
  } /* end for (i */

  return NULL;
} // end threadPartialSum
```

WGT-6

Week 6 ~~Friday~~ Thursday Video, Sect. 4.4-4.6

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + (-1)^n \frac{1}{2n+1} + \cdots\right).$$

The following *serial* code uses this formula:

```
double factor = 1.0;
double sum = 0.0;
for (i = 0; i < n; i++, factor = -factor) {
   sum += factor/(2*i+1);
}
pi = 4.0*sum;
```

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;

   if (my_first_i % 2 == 0)  /* my_first_i is even */
      factor = 1.0;
   else  /* my_first_i is odd */
      factor = -1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
      sum += factor/(2*i+1);
   }
   return NULL;
}  /* Thread_sum */
```

R3 (3+α) R3 (3+β)

load R3, sum
add R3, R3,

sum [3] critical section

Program 4.3: An attempt at a thread function for computing π — Wrong

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;

   if (my_first_i % 2 == 0)
      factor = 1.0;
   else
      factor = -1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
      while (flag != my_rank);
      sum += factor/(2*i+1);
      flag = (flag+1) % thread_count;
   }

   return NULL;
}  /* Thread_sum */
```

flag [0]

0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, ...

Program 4.4: Pthreads global sum with busy-waiting

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor, my_sum = 0.0;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;

   if (my_first_i % 2 == 0)
      factor = 1.0;
   else
      factor = -1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor)
      my_sum += factor/(2*i+1);

   while (flag != my_rank);
   sum += my_sum;
   flag = (flag+1) % thread_count;

   return NULL;
}  /* Thread_sum */
```

Program 4.5: Global sum function with critical section after loop

W6F-2

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;
   double my_sum = 0.0;

   if (my_first_i % 2 == 0)
      factor = 1.0;
   else
      factor = -1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
      my_sum += factor/(2*i+1);
   }
   pthread_mutex_lock(&mutex);
   sum += my_sum;
   pthread_mutex_unlock(&mutex);

   return NULL;
}  /* Thread_sum */
```

false
0 lock
sum

li R4,1
spin: testANDSet (R4), R5, lock
beq R5, 1, spin

Program 4.6: Global sum function that uses a mutex

Table 4.1 Run-Times (in Seconds) of π Programs Using $n = 10^8$ Terms on a System with Two Four-Core Processors

Threads	Busy-Wait	Mutex
1	2.90	2.90
2	1.45	1.45
4	0.73	0.73
8	0.38	0.38
16	0.50	0.38
32	0.80	0.40
64	3.56	0.38

~7.5

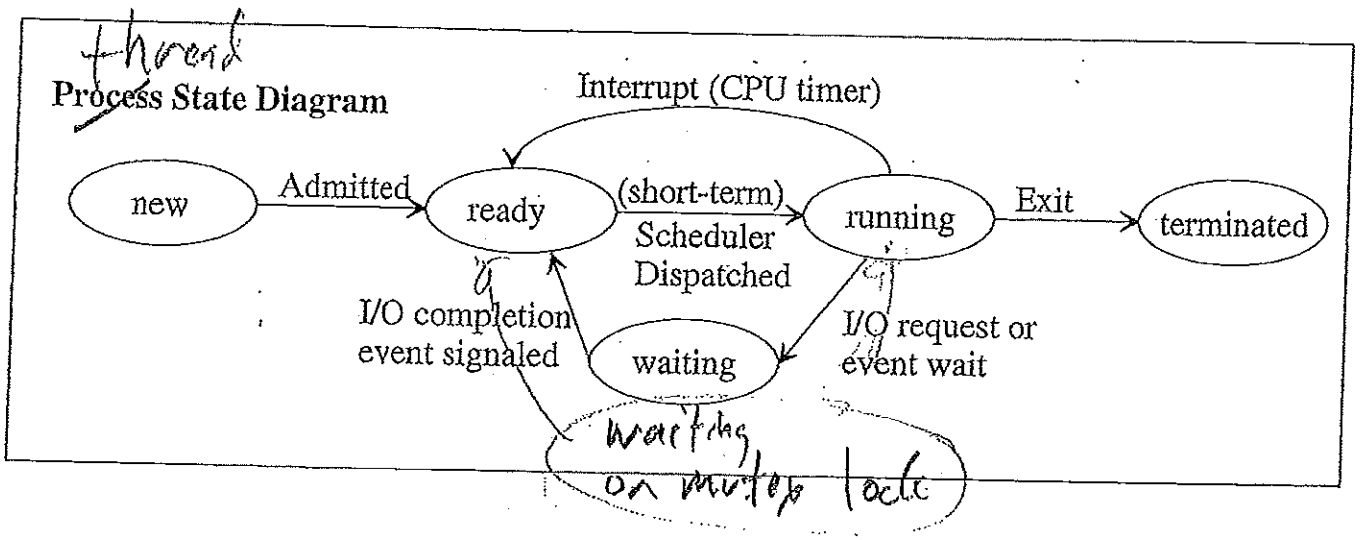
Serial

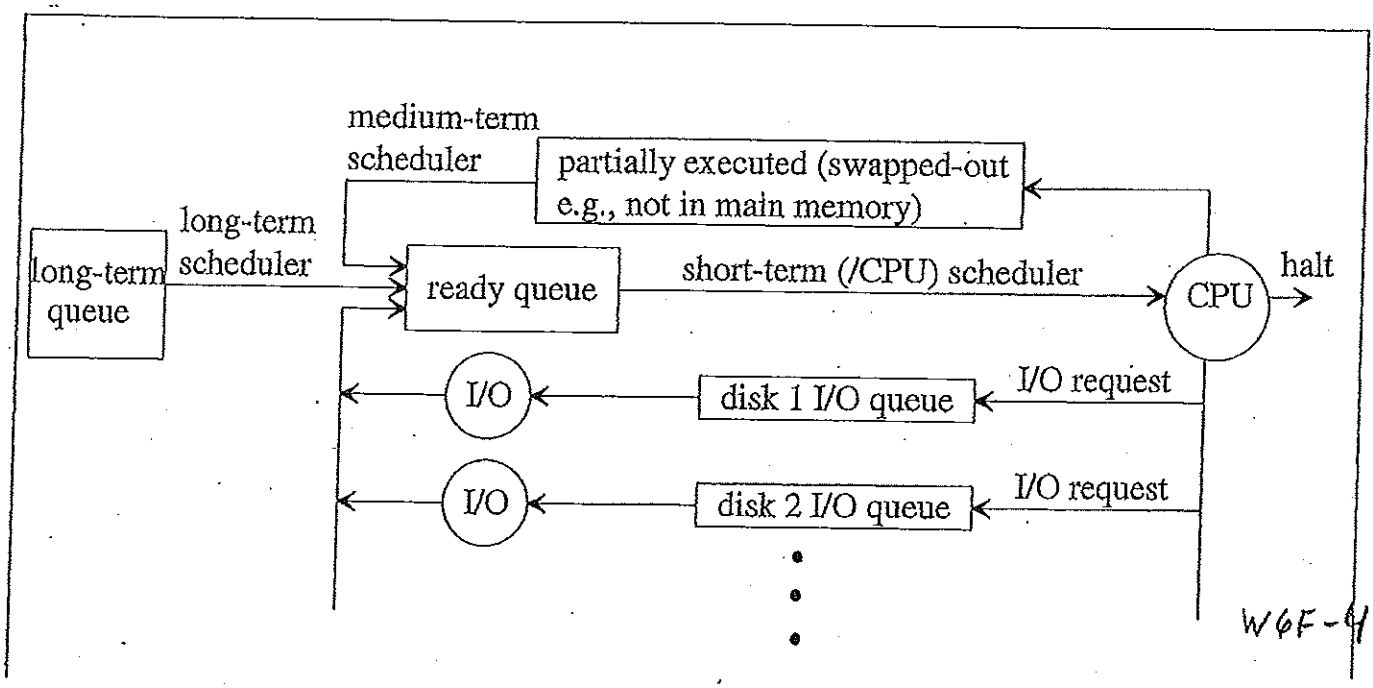
Processor

W6F-3

Table 4.2 Possible Sequence of Events with Busy-Waiting and More Threads than Cores

		Thread				
Time	Flag	0	1	2	3	4
0	0	crit sect	busy-wait	susp	susp	susp
1	1	terminate	crit sect	susp	busy-wait	susp
2	2	—	terminate	susp	busy-wait	busy-wait
⋮	⋮			⋮	⋮	⋮
?	2	—	—	crit sect	susp	busy-wait





W6F-4

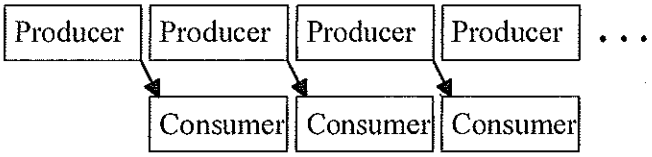
Week 7 Discussion Questions
Chapter 4.7 – 4.8:

1. Consider two threads that are linked by a processing pipeline where:

- a producer thread produces some data ("work"), and
- a consumer thread consumes the data (does the "work")

Without concurrency they could alternate:

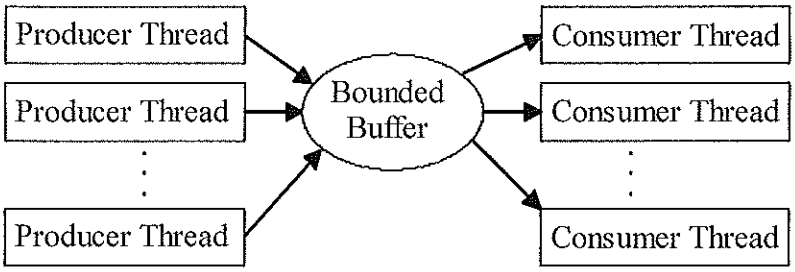
With concurrency they could pipeline:



a) What is the advantage of the pipelined version?

b) In the pipelined version, how does the rate the producer produces data compare to the rate the consumer consumes it?

2. A bounded buffer can be used to store some (limited by some bound) data between multiple producer and multiple consumer threads:



a) What is the advantage of the bounded buffer approach over pipelining the producer data directly to the consumer thread?

b) In the bounded buffer approach when would the producer thread need to block?

c) In the bounded buffer approach when would the consumer thread need to block?

3. Another common synchronization pattern is a *synchronization barrier* where we want all threads to reach the same point in a program before any thread continues (e.g., they all finish phase one of a calculation before they all start phase two). Typically, barrier are initialized with the number of threads for which to wait. As threads arrive at the barrier they issue a "wait" and are blocked. When the last thread arrives at the barrier, all threads are unblocked and allowed to continue execution. This pattern is often used in scientific calculations on large matrices (2D, 3D, etc. arrays).

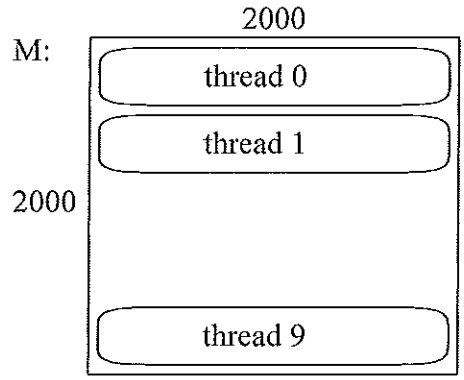
Week 7 Discussion Questions

Sequential code:

for time = 0 to 1000 do
  for row = 0 to 1999 do
    for column = 0 to 1999 do
      M[row][column] = ...

Static allocation of work to threads:

Assume that we assign 10 threads each 200 rows to calculate, but they all need to complete a time interval before any thread can move on to the next time interval.



Using high-level pseudo-code sketch the thread code focusing on where the barrier initialization and "barrier wait" would need to be performed.

4. Consider a barrier implementation with a condition variable (based on code from p. 180 of textbook):

```
/* Shared Global variables */
pthread_mutex_t barrier_lock;    /* mutex for the barrier */
pthread_cond_t all_here;         /* condition variable for barrier */
int count=0;                     /* counter for barrier */
int numberOfThreads;

void barrier(long id) {
  pthread_mutex_lock(&barrier_lock);
  count++;
  // printf("count %d, id %d\n", count, id);
  if (count == numberOfThreads) {
    count = 0;
    pthread_cond_broadcast(&all_here);
  } else {
    while(pthread_cond_wait(&all_here, &barrier_lock) != 0);
  } // end if
  pthread_mutex_unlock(&barrier_lock);
} // end barrier
```

a) Why does the `pthread_cond_wait` call need to be passed the `&barrier_lock`?

b) Why is the `while` –loop around the call to the `pthread_cond_wait` **not** busy-waiting?

Learning Objectives:
- Ability to create threads and pass them structures with complex parameter values.
- Apply condition variables and mutexes to create barriers to synchronize threads.

To start the lab:
- watch the Lab 7 Video on the eLearning system
- download `lab7.zip` from the eLearning system and unzip/extract it locally on your computer
- copy the `lab7` directory to `fermi1.cs.uni.edu` using a secure ftp client (winSCP, FileZilla, scp, etc.)
- log-on to `fermi1.cs.uni.edu` using Putty/ssh

Part A: Using an editor on `fermi1.cs.uni.edu` open the file `maddA.c` which contains a C program that allows the user to enter three integer command-line arguments: the # of rows, the # of columns, and the # of threads. It creates two matrices (i.e., two 2-dimensional arrays) of that size, fills the matrices with random floating-point numbers, and uses the specified number of pthreads to sum the two matrices values. In also calculates the sum of the matrices sequentially for comparison.

Answer the following questions about the `maddA.c` program.
The main function creates the threads in the loop:

```
for (i=0; i < numberOfThreads; i++) {
  if (errorCode = pthread_create(&threadHandles[i], NULL, threadMAdd, &blocksOfWork[i]) != 0) {
    printf("pthread %d failed to be created with error code %d\n", i, errorCode);
  } // end if
} // end for
```

a) How does the ith thread know what rows and columns of the Sum matrix is must calculate?

b) If a thread fails to be created, what would the user observe in the terminal window?

c) How does the main function know when all the threads have completed?

d) In the comparison of the Sum and Sum_seq matrices by the function equal2DArrays, why would a tolerance of 0.0 likely cause the matrices not to match?

The `threadMAdd` function is the code each thread runs. It is defined as:

```
void * threadMAdd(void * arg) {
  BLOCK * block = (BLOCK *) arg;
  int threadId = block->threadId;
  int startRow = block->start_row;
  int endRow = block->end_row;
  int startCol = block->start_col;
  int endCol = block->end_col;
  int r, c;

  for (r = startRow; r <= endRow; r++) {
    for (c = startCol; c <= endCol; c++) {
      Sum[r][c] = A[r][c] + B[r][c];
    } // end for (c...
  } // end for (r...
} // threadMAdd
```

e) Explain each part of the assignment statements:

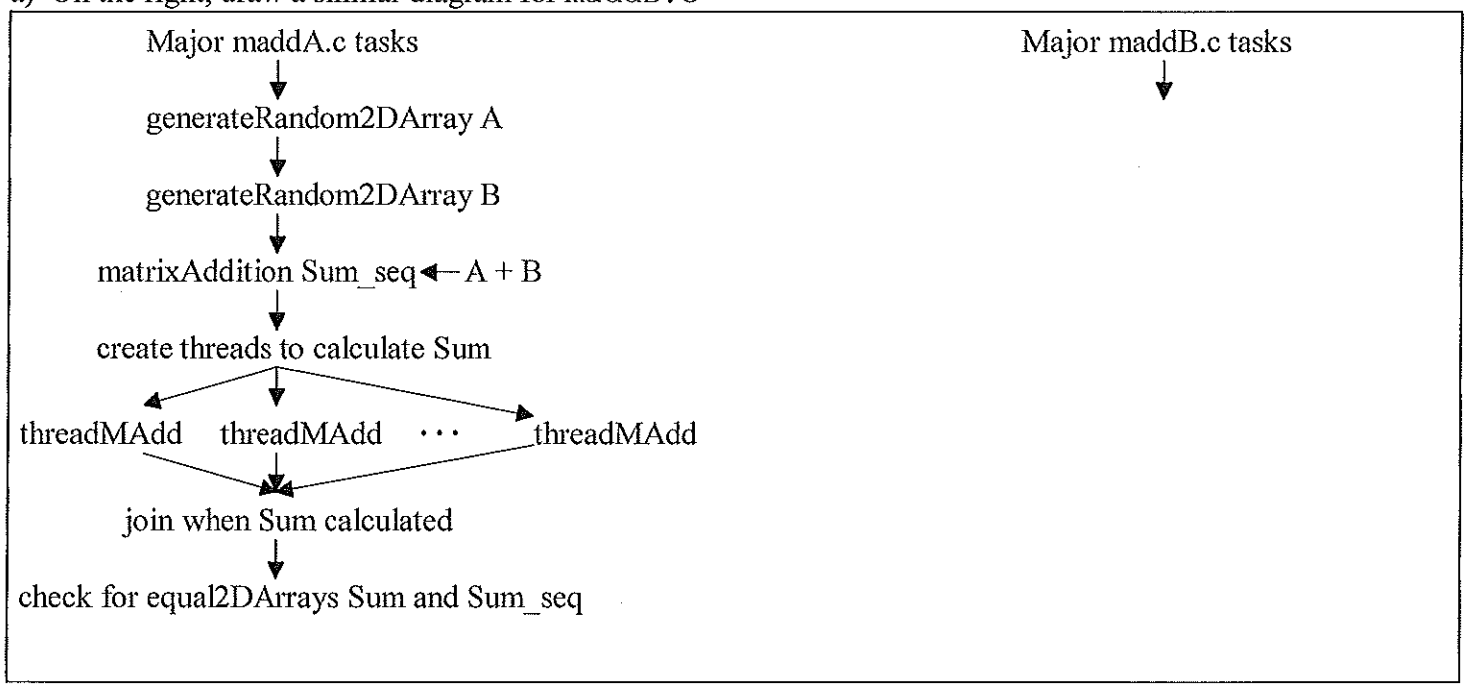
```
BLOCK * block = (BLOCK *) arg;
```

```
int threadId = block->threadId;
```

f) Why are the threads able to run unsynchronized to calculate the `Sum` matrix?

Part B: Open the file `maddB.c` which contains a similar program to `maddA.c`, except it tries to speed up the generation of the random matrices A and B by using multiple threads. The diagram on the left conceptualizes the major tasks of `maddA.c`.

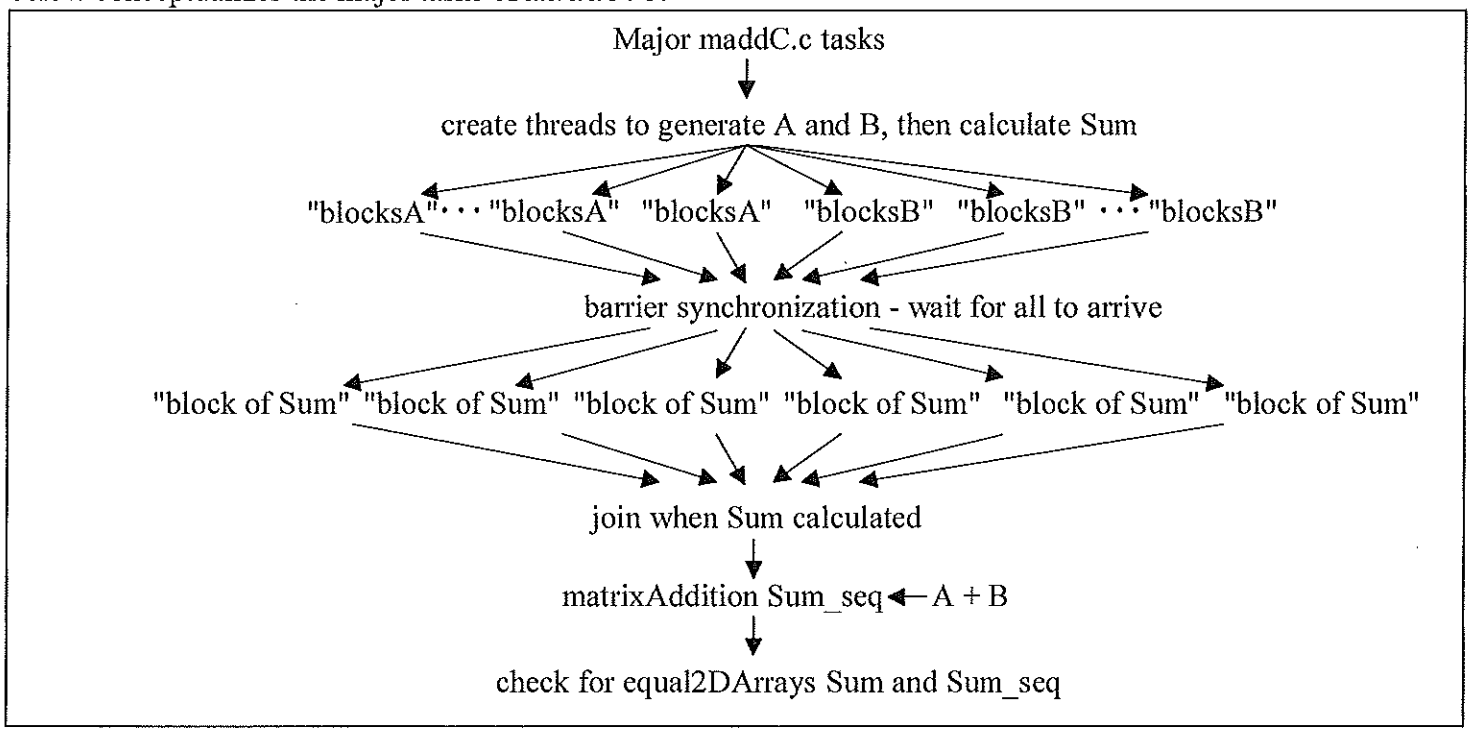
a) On the right, draw a similar diagram for `maddB.c`



b) Program `maddA.c` generations of the random matrices A and B by using the `rand()` function. Explain why program `maddB.c` needed to use the `rand_r()` function. Hints: As a starting point see the web-page at: http://linux.die.net/man/3/rand_r

c) Which of the two programs has more overhead for thread creation and destruction? (Explain your answer)

Part C: Open the file `maddC.c` which tries to speed up the `maddB.c` program by combining the generation of the random matrices A and B with the calculation of the the Sum matrix. by using multiple threads. The diagram below conceptualizes the major tasks of `maddC.c`.



a) Why is a barrier synchronization needed for the correct calculation of the Sum matrix?

The `barrier` function definition is:

```
void barrier(long id) {
  pthread_mutex_lock(&barrier_lock);
  count++;
  //printf("count %d, id %d\n", count, id);
  if (count == numberOfThreads) {
    count = 0;
    pthread_cond_broadcast(&all_here);
  } else {
    while(pthread_cond_wait(&all_here, &barrier_lock) != 0);
  } // end if
  pthread_mutex_unlock(&barrier_lock);
} // end barrier
```

b) What is the role of each of the following global variables in the `barrier` function?

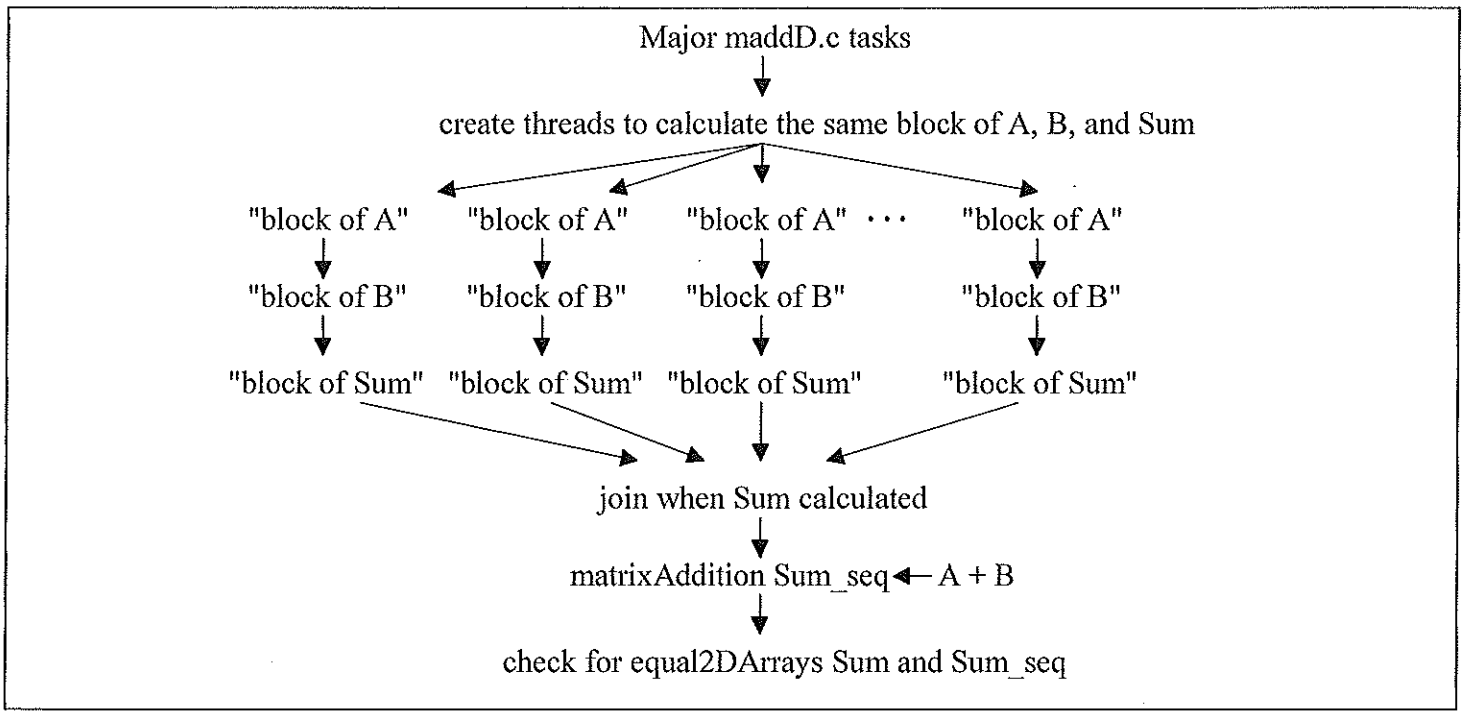
`count`

`barrier_lock`

`all_here`

c) What is the role if the `while`-statement: `while(pthread_cond_wait(&all_here, &barrier_lock) != 0);`?

Part D: Copy the `maddC.c` file to `maddD.c` (i.e., `cp maddC.c maddD.c`). Modify the `maddD.c` program to eliminate the need for a barrier. If each thread is responsible for calculate the same block of A, B, and Sum, then the barrier in not needed.



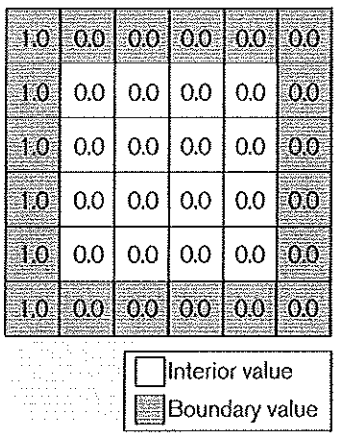
Submit lab7.zip containing question answers and completed programs on the eLearning system

Learning Objectives:

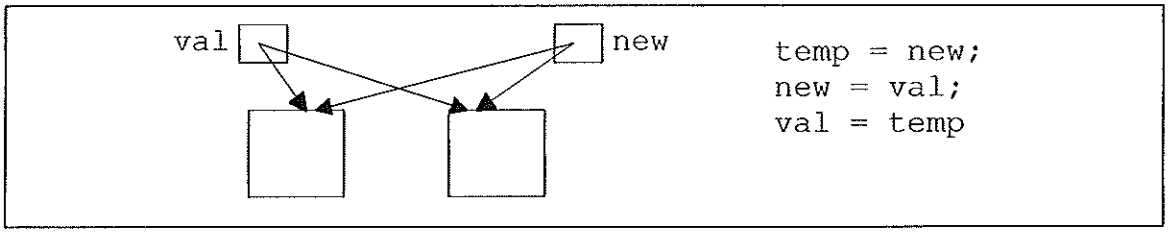
- Design an efficient data-decomposition (i.e., block vs. cyclic) for a pthread program in C.
- Write correct C program using pthread library commands to initialize/create pthreads and synchronize their operation.
- Time various sizes and # of threads

Homework #7 Description: You are to design and write a C program utilizing pthreads to efficiently perform 2D Successive Over-Relaxation (SOR) (often used in 3D form to solve differential equations such as Navier-Stokes equations for fluid flow). To start the homework, download and extract `hw7.zip` from the eLearning system. It contains a copy of a partial `hw7.c` program which you can use as your starting point.

2D SOR Problem: Initially, the 2D-array `val` contains 0.0s everywhere, except for the 1.0s down column 0. On each iteration, SOR updates all **interior** values (i.e., only the white values in the diagram change with the gray boundary values being fixed) by the average of their four nearest neighbors: above, below, to the left, and to the right. Eventually after many iterations the values will stablize. We won't run to complete stablization, but just until the maximum value change across the whole array during an iteration is less than a user specified *threshold* (e.g., 0.0001).



During an iteration use the 2D-array `val` to compute the updated values in a second 2D-array `new`. Before the next iteration, flip-flop and reuse these 2D-arrays. This is actually easy and efficient since `val` and `new` are really pointers to the arrays, so all we really need to do is swap their pointer values.



The starter `hw7.c` program contains sequential 2D SOR code (see next page) for timing comparisons. You'll need to complete the pthread parallelization. The general steps are:

- decide how to decompose the work among threads -- keep in mind load balancing, cache performance
- decide how to synchronize the threads so all threads finish an iteration before any start the next iteration
- decide how to synchronize the threads so all threads stop if the max. change at any spot during an iteration is less than the threshold
- code the `thread_main` function run by all the threads
- test and debug your program

Compile (`gcc -o sor -O3 hw7.c -lpthread -lm`) and run the program on `fermi1.cs.uni.edu` with a threshold of 0.0001varying the matrix size and number of threads to complete the following table.

Interior Matrix Sizes	Time of sequential SOR `sequential2DSOR` function (in seconds)	Time of your parallel 2D SOR function (in seconds) using a varying number of threads				
		1 thread	2 threads	4 threads	8 threads	16 threads
500 x 500						
1000 x 1000						
1500 x 1500						
2000 x 2000						

Submit hw7.zip containing a completed timing table and completed program (HW7.c)

on the eLearning system

```
void sequential2D_SOR() {
  double average, maxDelta, thisDelta;
  double **temp;
  int i, j;

  do {
    maxDelta = 0.0;

    for (i = 1; i <= n; i++) {
      for (j = 1; j <= n; j++) {
          average = (val[i-1][j] + val[i][j+1] + val[i+1][j] + val[i][j-1])/4;
          thisDelta = fabs(average - val[i][j]);
          if (maxDelta < thisDelta) {
            maxDelta = thisDelta;
          } // end if

          new[i][j] = average;    // store into new array

      } // end for j
    } // end for i
    temp = new;            /* prepare for next iteration */
    new = val;
    val = temp;
    // printf("maxDelta = %8.6f\n", maxDelta);
  } while (maxDelta > threshold);  // end do-while
  delta = maxDelta; // sets global delta
} // end sequential2D_SOR
```

Common synchronization patterns between threads

critical section - shared variable(s) updated must be done mutually exclusively. In pthreads, we can use a mutex to synchronize them without "busy-wait". Waiting threads are not using the CPU/core, but on an "unlock" we have no control which gets the woke-up.

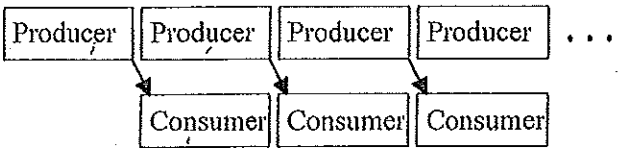
1) Another common synchronization pattern is *bounded buffer* where two threads are linked by a processing pipeline where:

- a producer thread produces some data ("work"), and
- a consumer thread consumes the data (do the "work")

Without concurrency they could alternate:

→ time

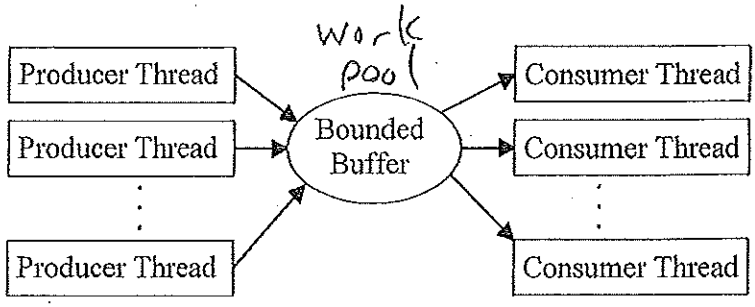
With concurrency they could pipeline:



a) What is the advantage of the pipelined version?

b) In the pipelined version, how does the rate the producer produces data compare to the rate the consumer consumes it?

2) A bounded buffer can be used to store some (limited by some bound) data between the producer and consumer threads:



a) What is the advantage of the bounded buffer approach over pipelining the producer data directly to the consumer thread?

b) In the bounded buffer approach when would the producer thread need to block?

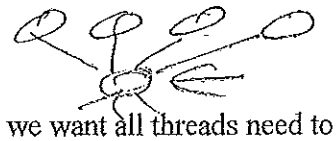
buffers full

wake when consumer removes work

c) In the bounded buffer approach when would the consumer thread need to block?

buffer is empty

producer adds work

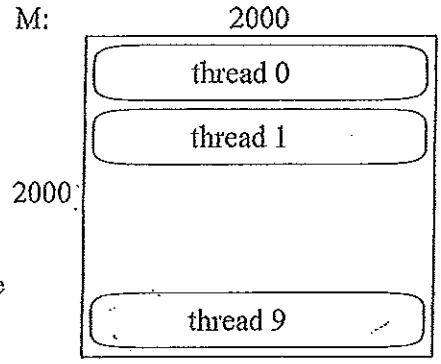


3) Another common synchronization pattern are *synchronization barriers* where we want all threads need to finish some phase of a task before any can continue. This pattern is often used in scientific calculuations on large matrices (2D, 3D, etc. arrays). Barrier are initialized with the number of threads for which to wait. As threads arrive at the barrier and issue a "wait," all threads are blocked until the last thread performs a wait.

Sequential code:

```
for time = 0 to 1000 do
  for row = 0 to 1999 do
      for column = 0 to 1999 do
          M[row, column] = ...
```

Might assign 10 threads each 200 rows to calculate, but they all need to complete a time interval before any thread can move on to the next time interval.



Sketch the multi-thread code focusing on where the barrier initialization and wait would be performed.

Initialize barrier
for time = 0 to 1000
for row = firstR to ...
for column = ~~[illegible]~~
wait until all threads arrive at barrier

4) Readers / Writer lock.

To implement these synchronization patterns we need mechanisms (tools) to allow threads to wait until circumstances are appropriate for it to proceed. Two common mechanisms are: semaphore

I. *condition variables* used with monitors (or mutexes used in a monitor style) have operations

- wait - executed by a thread finding circumstances ("conditions") not to its liking, so it sleeps/blocks until another thread does a signal/notify operation
- signal/notify - unblocks a waiting thread and moves it to the run queue
- broadcast/notify-all - unblocks all waiting threads.

critical section

General pseudocode syntax:

```
lock mutex
if condition is favorable then
        signal thread(s)
else
        unlock mutex and block
        /* when thread is unblocked, mutex is relocked */
end if
unlock mutex
```

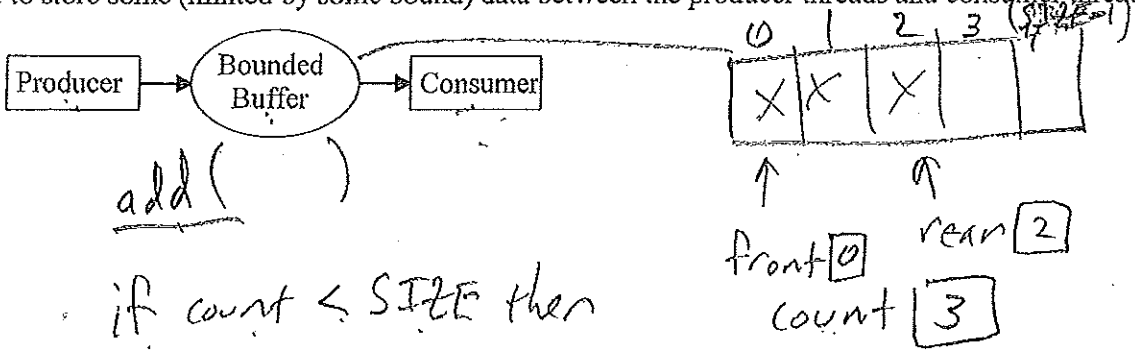
4) Consider a *barrier synchronization* with global integers `threadCount` (# of total threads) and `counter`(count of threads that have reached the barrier. Complete if-statement check for "favorable condition".

```
int threadCount;
int counter = 0;
pthread_mutex_t mutex;
pthread_cond_t condVar;

int main(...) {
    pthread_cond_init(&condVar, NULL);
...

    pthread_cond_destroy(&condVar);
} // end main

void* threadWork(...) {
    ...
    /* Barrier */
    pthread_mutex_lock(&mutex);
    counter++;
    if ( counter >= threadCount ) {
        counter = 0;
        pthread_cond_broadcast(&condVar);
    } else {
        while(pthread_cond_wait(&condVar, &mutex) != 0);
    } // end if
    pthread_mutex_unlock(&mutex);
    ...
} // end threadWork
```

Notes:

- the `pthread_cond_wait` will unlock the mutex parameter and cause the executing thread to block until:
  - `pthread_cond_broadcast` or `pthread_cond_signal` by another thread in which case `pthread_cond_wait` returns a 0
  - other system events ("bugs") have been know to cause threads to be woke-up, but with non-0 return value. Hence the while-loop around the `pthread_cond_wait`

5) Outline how pthread mutex(es) and condition variable(s) can be used to implement a bounded buffer that can be used to store some (limited by some bound) data between the producer threads and consumer threads:



add( )

if count < SIZE then

front 0 rear 2

count 3

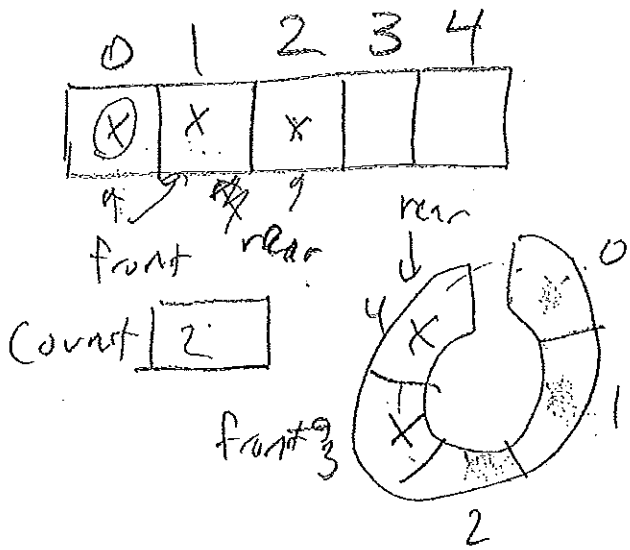
2 cond. variables

non Empty - signal consumer

non Full

Week 7 Thursday Video Notes


```
/*  Programmer:  Mark Fienup
    File:        bounded_buffer.c
    Compiled by: gcc bounded_buffer.c -lpthread -lm
    Description:  Bounded buffer using pthread condition variables to avoid ov
er or underflowing the bounded buffer.
*/
#include <unistd.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <pthread.h>
#include <stdlib.h>

#define SIZE 5
#define MAX_PRODUCER_SLEEP 10
#define MAX_CONSUMER_SLEEP 5
#define MAX_DURATION 100

// Global Bounded Buffer
int buffer[SIZE];
int count = 0;
int front = -1;
int rear = 0;

pthread_mutex_t lock;
pthread_cond_t nonFull;
pthread_cond_t nonEmpty;

void  *producerWork(void *);
void  *consumerWork(void *);
void bufferAdd(int);
int bufferRemove();
```



rear = (rear+1) % SIZE

0 1 2 3 4 0 1 2 3 4 0

```
int main(int argc, char * argv[]) {

  pthread_mutex_init(&lock, NULL);
  pthread_cond_init(&nonFull, NULL);
  pthread_cond_init(&nonEmpty, NULL);

  pthread_t threadHandle[64];

  int i,numberOfThreads;

  if (argc != 2) {
    printf("usage: %s <integer number of producter and consumer threads>\n",
       argv[0]);
    exit(1);
  } // end if

  sscanf(argv[1], "%d", &numberOfThreads);

  srand(5);

  for (i=0; i < numberOfThreads; i++) {
    if (i % 2 == 0) {
      // create producer threads
      pthread_create(&threadHandle[i], NULL, producerWork, &i);
```

```
    } else {
      pthread_create(&threadHandle[i], NULL, consumerWork, &i);
    } // end if
  } // end for

    for (i=0; i < numberOfThreads; i++) {
      pthread_join(threadHandle[i], (void **) NULL);
    } // end for
    printf("After join in main!\n");

} // end main


void  *producerWork(void * args) {

  int threadId;
  int counter = 0;
  unsigned int sleepAmt;

  threadId = *((int *) args);

  while (counter < MAX_DURATION) {
    sleepAmt = rand() % MAX_PRODUCER_SLEEP;
    sleep(sleepAmt);
    printf("Producer try to add when count = %d\n", count);
    bufferAdd(threadId);
    printf("Producer added\n");
    counter++;
  } // end while


} // end producerWork

void  *consumerWork(void * args) {

  int threadId;
  int counter = 0;
  unsigned int sleepAmt;

  threadId = *((int *) args);

  while (counter < MAX_DURATION) {
    sleepAmt = rand() % MAX_CONSUMER_SLEEP;
    sleep(sleepAmt);
    printf("Consumer trying to remove when count = %d\n", count);
    bufferRemove(threadId);
    printf("Consumer removed\n");
    counter++;
  } // end while


} // end consumerWork
```

```
void bufferAdd(int item) {

  pthread_mutex_lock(&lock);
  while(count == SIZE) {
    while(pthread_cond_wait(&nonFull, &lock) != 0);
  } // end while
  if (count == 0) {
    front = 0;
    rear = 0;
  } else {
    rear = (rear + 1) % SIZE;
  } // end if
  buffer[rear] = item;
  count++;
  pthread_cond_signal(&nonEmpty);
  pthread_mutex_unlock(&lock);

} // end bufferAdd

int bufferRemove() {
  int returnValue;

  pthread_mutex_lock(&lock);
  while(count == 0) {
    while(pthread_cond_wait(&nonEmpty, &lock) != 0);
  } // end while
  returnValue = buffer[front];
  front = (front + 1) % SIZE;
  count--;
  pthread_cond_signal(&nonFull);
  pthread_mutex_unlock(&lock);
  return returnValue;

} // end bufferRemove
```

Week 8 Discussion Questions
Chapter 4.9 – 4.11:

1. Another common synchronization pattern is *readers/writers locks* which are similar to mutexs, except threads specify whether they are locking for writing (updating) or only reading the shared data structure.
a) Since writer threads must update the shared data structure mutually exclusively, what concurrency would readers/writers locks provide?

b) Consider the scenario where one (or more) reader thread(s) hold the lock and there are waiting writer thread(s) when a new reader thread "arrives." Two possibilities are:

- Allow the new reader to immediately start reading
- Make the new reader wait until after the waiting writers

What are the advantages and disadvantages of each approach?

c) Many systems including POSIX provide an implementation of readers/writers locks (`pthread_rwlock_init`, `pthread_rwlock_rdlock`, `pthread_rwlock_wrlock`, `pthread_rwlock_unlock`). However, the choice of prioritizing a new reader vs. waiting writer(s) is open to system implementation. What are the advantages and disadvantages of not specifying this choice?

d) POSIX has specialized forms of readers/writers locks for files: `fcntl` procedure (complex) and simpler `flock` interface. Why are files likely choice for applying readers/writers locks toward?

2. In Lab 6 Part B, we summed a 1-D array of doubles using a global `threadSums` array where each thread used its rank (thread id) as an index into the `threadSums` array. The `threadPartialSum` function that each thread runs is defined as:

```
void * threadPartialSum(void * rank) {

  long myRank = (long) rank;
  long i, blockSize;
  long firstIndex, lastIndex;

  blockSize = length / numberOfThreads;
  firstIndex = blockSize * myRank;
  if (myRank == numberOfThreads-1) { // last thread gets the rest
    lastIndex = length-1;
  } else {
    lastIndex = blockSize * (myRank+1) - 1;
  } // end if

  threadSums[myRank] = 0.0;
  for (i=firstIndex; i <= lastIndex; i++) {
    threadSums[myRank] += myArray[i];
  } /* end for (i */

  return NULL;
} // end threadPartialSum
```

a) Explain how threadSums could cause *false sharing* if the threads run on cores with separate caches.

Week 8 Discussion Questions

b) How would the following modification reduce *false sharing* among the threads?

```
void * threadPartialSum(void * rank) {

  double localSum;

  <CODE OMITTED HERE>

  localSum = 0.0;
  for (i=firstIndex; i <= lastIndex; i++) {
    localSum += myArray[i];
  } /* end for (i */

  threadSums[myRank] = localSum;

  return NULL;
} // end threadPartialSum
```

3. What is meant by *thread-safe*?

4. The threadGenerate2DBlock function from lab7/maddB.c uses rand_r instead of rand to generate a random number. Why does rand_r have a parameter and rand does not? (see http://linux.die.net/man/3/rand_r)

```
/*******************************************************************
* Function threadGenerate2DBlock - each thread is passed a BLOCK structure
* define the block of an array to randomly generate.
* NOTE:  rand_r is used to generate random #'s because it is "thread safe"
*******************************************************************/
void * threadGenerate2DBlock(void * arg) {
  BLOCK * block = (BLOCK *) arg;
  int threadId = block->threadId;
  int startRow = block->start_row;
  int endRow = block->end_row;
  int startCol = block->start_col;
  int endCol = block->end_col;
  double ** array = block->array;
  double min = block->min;
  double max = block->max;
  int r, c;
  double range, div;
  int seed = (threadId+1)*1000;

  for (r = startRow; r <= endRow; r++) {
    for (c = startCol; c <= endCol; c++) {
      range = max - min;
      div = RAND_MAX / range;
      array[r][c] = min + (rand_r(&seed) / div);
    } // end for (c...
  } // end for (r...
} // end threadGenerate2DBlock
```

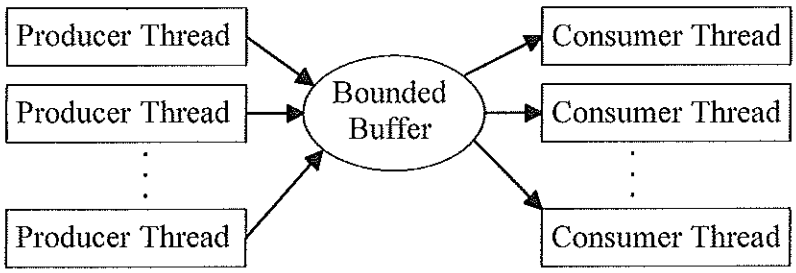
Learning Objectives:

- Ability to create threads and pass them structures with complex parameter values.
- Apply the synchronization pattern of Producer threads and Consumer threads with a bounded buffer of work to solve a programming problem.

To start the lab:

- watch the Lab 8 Video on the eLearning system
- download `lab8.zip` from the eLearning system and unzip/extract it locally on your computer
- copy the `lab8` directory to `fermi.cs.uni.edu` using a secure ftp client (winSCP, FileZilla, scp, etc.)
- log-on to `fermi1.cs.uni.edu` using Putty/ssh

Part A: Using an editor on `fermi1.cs.uni.edu` open the file `lab8/maddE.c` which contains a C program that allows the user to enter four integer command-line arguments: the # of rows, the # of columns, the # of Producer threads, and the # of Consumer threads. As in lab 7, `maddE.c` creates two matrices (i.e., two 2-dimensional arrays) of that size, fills the matrices with random floating-point numbers, and sums the two matrices. However, `maddE.c` uses a collection of Producer threads that generate corresponding single rows of both matrices and puts them is a bounded buffer for Consumer threads to remove and sum.



Answer the following questions about the `maddE.c` program.

a) How does a Producer thread know what row of A and B need to be generated?

b) What type of synchronization is performed among the Producer threads in determining what row of A and B need to be generated?

c) What information does a Producer thread put into the bounded buffer?

d) What granularity of work does a Producer thread put into the bounded buffer?

e) Describe in English how the Producer code would be modified to increase the granularity of work?

f) Describe in English how the Producer code (and other parts of the program) would be modified to decrease the granularity of work?

g) How and when does a Producer thread get out of its infinite loop (i.e., `while (TRUE)`)?

h) Consumer threads never get out of their infinite loops. How is the `main` function prevented from calculating the Sum_seq array until the Consumers finish the Sum array calculation?

Part B: Compile (`gcc -O3 -o madd maddE.c -lpthread -lm`) and repeatedly run on 20000 x 20000 varying the number of producer and consumer threads (`./madd 20000 20000 1 1`, `./madd 20000 20000 1 2`, etc.) to complete this table:

Number of Producer Threads	Time of your parallel maddE program (in seconds) using a varying number of Consumer threads				
	1 thread	2 threads	4 threads	8 threads	16 threads
1					
2					
4					
8					
16					

a) What ratio of Producer threads : Consumer threads seems to be optimal?

b) Why is one kind of thread slower than the other?

Part C: Answer the following questions about the `bufferAdd` and `bufferRemove` functions of the `maddE.c` program.

a) When a Producer thread calls `bufferAdd` and the `buffer` is full (i.e., `count == SIZE`), it gets suspended and waits at the condition variable `notFull`. How does it get woken up?

b) What is the purpose of **each** of the nested while-loops?

```
while(count == SIZE) {
.   while(pthread_cond_wait(&nonFull, &lock) != 0);
} // end while
```

- `while(count == SIZE):`

- `while(pthread_cond_wait(&nonFull, &lock) != 0);`

c) Both the `bufferAdd` and `bufferRemove` functions start with `pthread_mutex_lock(&lock);` and end with `pthread_mutex_unlock(&lock);` which makes them a single critical section (i.e., guarantees that only a single thread (Producer or Consumer) is modifying the shared variables: `buffer`, `count`, `front`, and `rear`. However, suppose the buffer is full and Producer is suspended and waits at the condition variable `notFull` with is inside the critical section. How does a Consumer thread get past the `pthread_mutex_lock(&lock);` to remove at item from the buffer?

Part D: Copy the `maddE.c` file to `maddF.c` (i.e., `cp maddE.c maddF.c`). Modify the `maddF.c` program to allow a Producer to generate a blockOfWork that's more than one row. Add a new comand-line parameter used to fill a global variable `blockSize`. (My thought was a block of rows, but to can do a block of rows and columns if you want.)

Compile (`gcc -O3 -o madd maddF.c -lpthread -lm`) and repeatedly run your program on 20000 x 20000 arrays. Varying the number of producer and consumer threads (`./madd 20000 20000 1 1 8`, `./madd 20000 20000 1 2 8`, etc.) to complete this table with a `blockSize` of 8:

Number of Producer Threads	Time of your parallel maddF program (in seconds) using a varying number of Consumer threads with a `blockSize` of 8				
	1 thread	2 threads	4 threads	8 threads	16 threads
1					
2					
4					
8					
16					

Submit lab8.zip containing question answers and completed programs on the eLearning system

Learning Objectives:

- Design an efficient Producer/Consumer configuration of a pthread program in C that uses a bounded buffer.
- Write correct C program using two different types of pthreads Producer/Consumer with a bounded buffer of work.
- Time various sizes and # of threads

Homework #8 Description:

You are to design and write a C program utilizing two different types of pthreads Producers and Consumers with a bounded buffer of work. The purpose of the program is to process a sequence Chroma key / Green-screen frames (see http://en.wikipedia.org/wiki/Chroma_key) that we can use to create a video of the "Loch Ness monster." You will be given:

- 99 pictures (nessie001.ppm, ..., nessie099.ppm) showing the "Loch Ness monster" moving across a green-screen background (The .ppm format is used here for its simplicity -- **http://www.fileformat.info/format/pbm/egff.htm**)
- a picture of "Loch Ness" called swans.ppm that's to be used as the new background picture replacing the green-screen background in the "nessie" pictures to generate 99 new pictures: frame001.ppm, ..., frame099.ppm
- a sequential version of the program `seqChromakey.c` which takes about 18.5 seconds (on fermi1.cs.uni.edu) to produce < 5 seconds of video (minimum video frames per second, FPS, is about 20 FPS) . Clearly too slow for "live" green-screen streaming like you might see in a weather forecast on the local news.

Parallel Program Requirements:

Two types of pthreads (Producers and Consumers) should work together using a bounded buffer of work where:

- a Producer thread's job is to read the next "nessie" picture from disk and add it to the bounded buffer of PICTURE structures waiting to be processed
- a Consumer thread's job is to remove a "nessie" PICTURE from the bounded buffer, process it by doing the green-screen background replacement, and write the new "frame" picture to disk

Hints:

- You should look at the simple `bounded_buffer.c` program from lab 8. The `bufferAdd` and `bufferRemove` functions can be modified to work with a global buffer that's an array of PICTURE structures.
- Producer threads might share an integer global variable `nextFrameNumber` that they access mutually exclusively to get the next frame of "nessie" pictures needing to be read from disk.
- Consumer threads might share an integer global variable `consumedCount` that they increment after processing and writing a new frame. The comsumer thread processing the last picture can exit the program.
- Both the Producer and Consumer threads can reuse the functions from the `seqChromakey.c` program: `allocate2DArray`, `free2DArray`, `generateFileName`, `readPicture`, `writePicture`, and `chromakey`.

Compile (`gcc -o hw8 -O3 hw8.c -lpthread -lm`) and run your program on fermi1.cs.uni.edu varying the number of producer and consumer threads. Use the `time` command when executing the programs to get the "`real`" time is seconds, e.g., `time ./hw8 3 2`

```
real    0m4.489s
user    0m19.230s
sys     0m0.727s
```

This shows that on my solution program with 3 producer threads and 2 consumer threads took 4.489 seconds.

Complete the following table to determine the "optimal" number of producer and consumer threads for this problem.

Number of Producer Threads	Time of your parallel hw8 program (in seconds) using a varying number of Consumer threads				
	1 thread	2 threads	4 threads	8 threads	16 threads
1					
2					
4					
8					

16					

Submit hw8.zip containing a completed timing table and completed program (hw8.c) on the eLearning system
DO NOT INCLUDE ANY OF THE .PPM FILES IN YOUR hw8.zip FILE!!!

Comp. Arch. Week 8 ~~Monday~~ Tuesday

update shared variable

R/W Locks and Deadlock

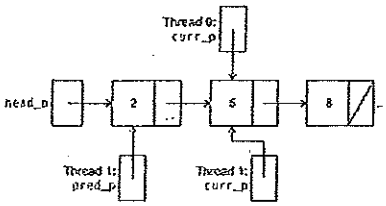
Another common synchronization pattern is *readers/writers locks* which are similar to mutexs, except threads specify whether they are locking for writing (updating) or only reading the shared data structure.

1) Shared sorted linked list textbook example with three operations Member (search), Delete, and Insert.

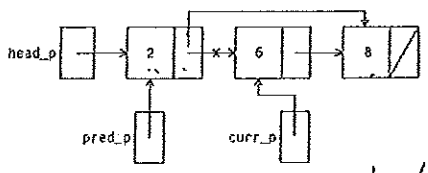
a) Which operation(s) need mutex exclusion? Delete, Insert

b) Which operation(s) can occur simultaneously? Member

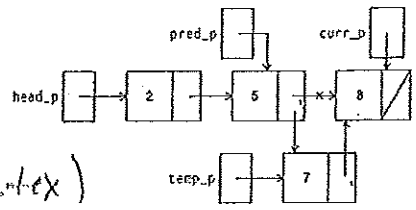
Member



Delete



Insert



lock(mutex)
Delete
unlock(mutex)

Textbooks discusses three solutions:

- Threads put mutex lock and unlock around their calls to list operations
- Lock individual nodes by putting mutex in each node. ("fine-grained" approach)

```
int  Member(int value) {
   struct list_node_s* temp_p;

   pthread_mutex_lock(&head_p_mutex);
   temp_p = head_p;
   while (temp_p != NULL && temp_p->data < value) {
      if (temp_p->next != NULL)
         pthread_mutex_lock(&(temp_p->next->mutex));
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      pthread_mutex_unlock(&(temp_p->mutex));
      temp_p = temp_p->next;
   }

   if (temp_p == NULL || temp_p->data > value) {
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      if (temp_p != NULL)
         pthread_mutex_unlock(&(temp_p->mutex));
      return 0;
   } else {
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      pthread_mutex_unlock(&(temp_p->mutex));
      return 1;
   }
}  /* Member */
```

```
struct list_node_s {
   int data;
   struct list_node_s* next;
   pthread_mutex_t mutex;
}
```

```
pthread_rwlock_rdlock(&rwlock);
Member(value);
pthread_rwlock_unlock(&rwlock);
. . .
pthread_rwlock_wrlock(&rwlock);
Insert(value);
pthread_rwlock_unlock(&rwlock);
. . .
pthread_rwlock_wrlock(&rwlock);
Delete(value);
pthread_rwlock_unlock(&rwlock);
```

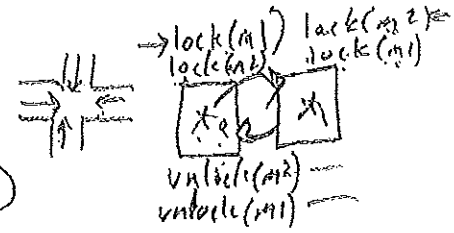
- Read-Write Locks to allow multiple readers/Member operations

Implementation	Number of Threads 1	2	4	8
Read-Write Locks	0.213	0.123	0.098	0.115
One Mutex for Entire List	0.211	0.450	0.385	0.457
One Mutex per Node	1.680	5.700	3.450	2.700

100,000 ops/thread
99.9% Member
0.05% Insert
0.05% Delete

Implementation	Number of Threads 1	2	4	8
Read-Write Locks	2.48	4.97	4.69	4.71
One Mutex for Entire List	2.50	5.13	5.04	5.11
One Mutex per Node	12.00	29.60	17.00	12.00

100,000 ops/thread
80% Member
10% Insert
10% Delete



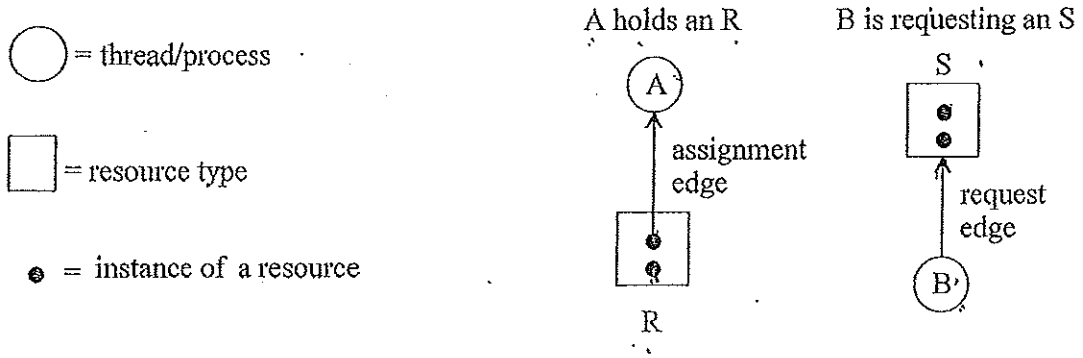
Deadlock

Necessary Conditions for Deadlock (Coffman, et al 1971): ALL MUST HOLD!

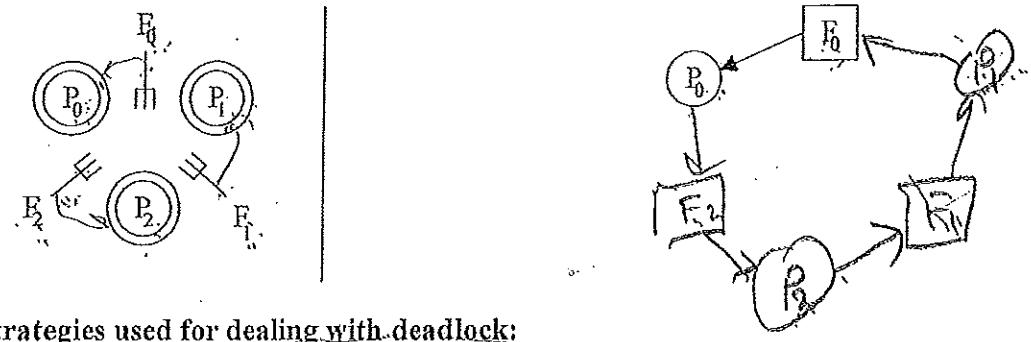
- mutual exclusion: resource(s) used in mutually exclusive fashion
- hold and wait: threads/processes currently holding granted resources can request new resources
- no preemption: resources are released voluntarily by the thread/process holding it after it has completed its task
- circular wait: a circular chain of two or more threads/processes that are each waiting for a resource held by the next member of the chain.

A mutex is a single-instance resource drawn as just a box in the text.

Resource-Allocation Graph (Holt '72) - (a directed graph model)



a) Consider the Dining Philosophers problem below. Complete the resource-allocation graph showing deadlock if all the philosophers are holding their left forks.



Strategies used for dealing with deadlock:

1) Ignore the problem - "Ostrich algorithm". Is it a rare occurrence with minor consequences?
2) Detection and recovery - Periodically run an algorithm that checks for cycles in a resource-allocation graph. Rollback or kill a thread/process to break the cycle.
3) Deadlock Prevention - OS rules that prevent one of the 4 necessary conditions for deadlock
4) Deadlock Avoidance - dynamically avoid deadlock by careful resource allocation - make sure that there is some order that threads/processes can finish execution without deadlock.

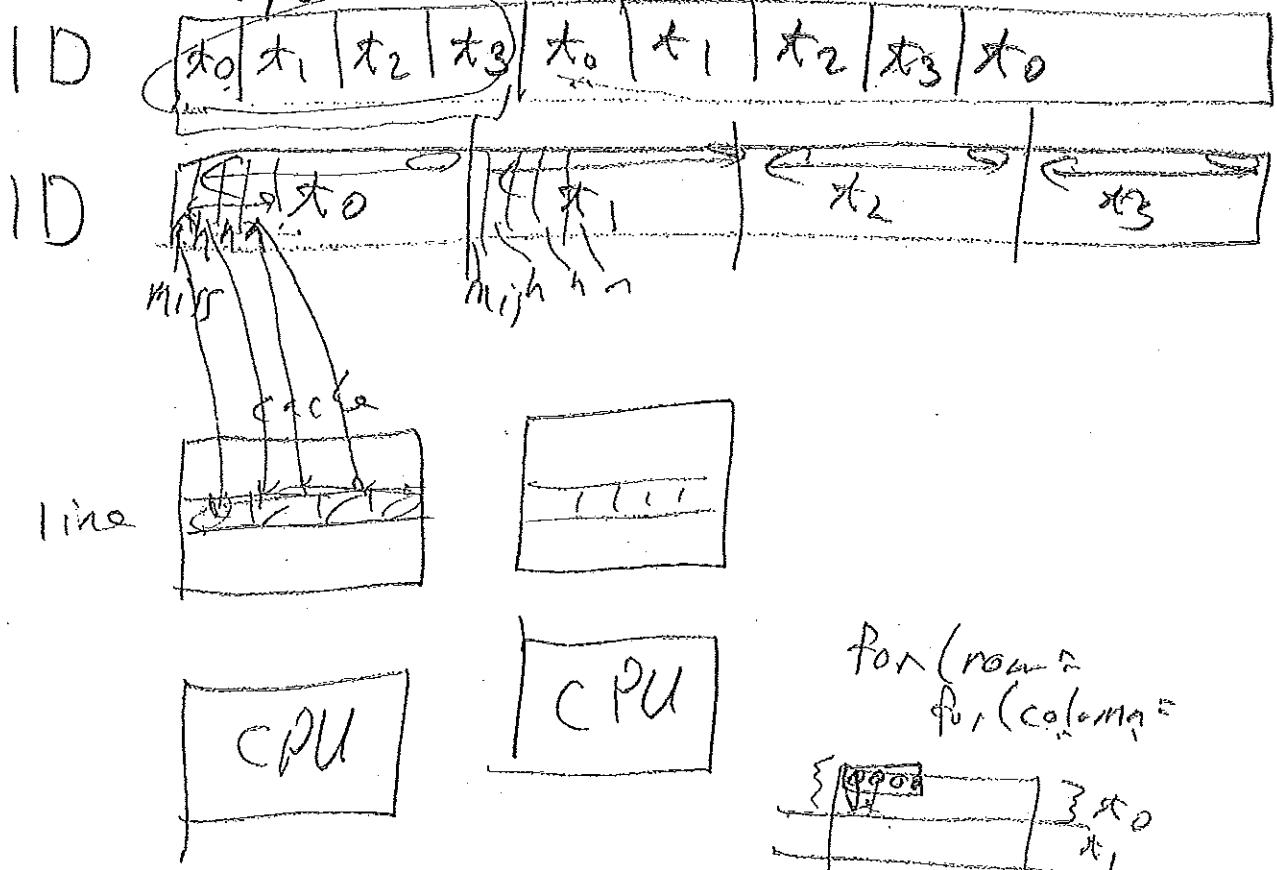
One appoach to *deadlock prevention is through resource ordering* - resources are ordered (e.g., numbered artificially) and we require threads/processes to request needed resources in ascending order.

a) Consider your resource-allocation graph for the above Dining Philosophers problem. If forks must be acquired in ascending order, how is the above deadlock prevented?

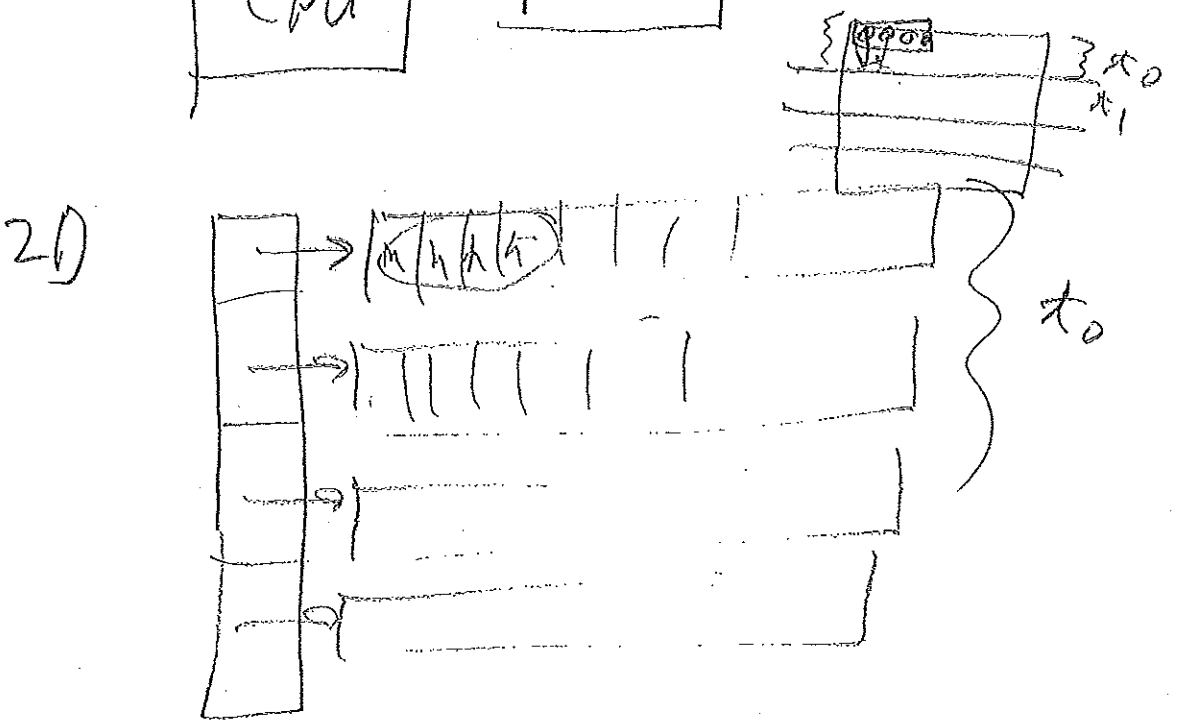
Week 8 ~~Friday~~ Thursday Sect. 4.10 + 4.11

Cache

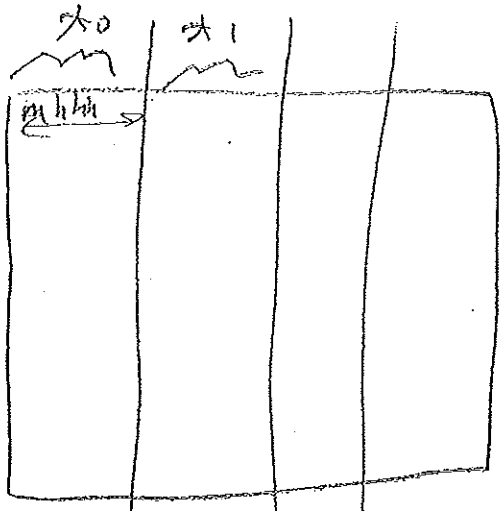
cyclic decomposition



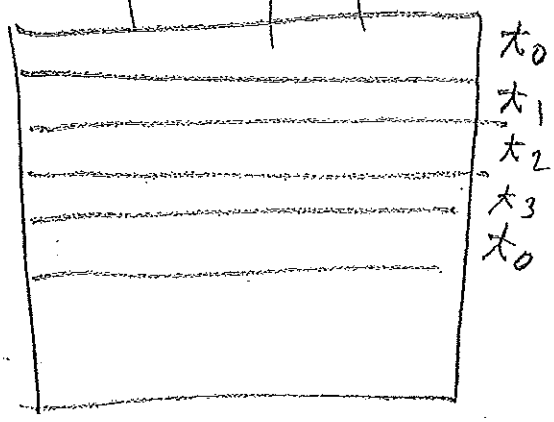
for (row =
for (column =

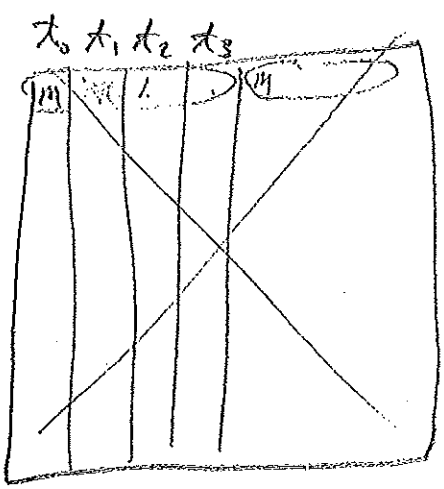


2D

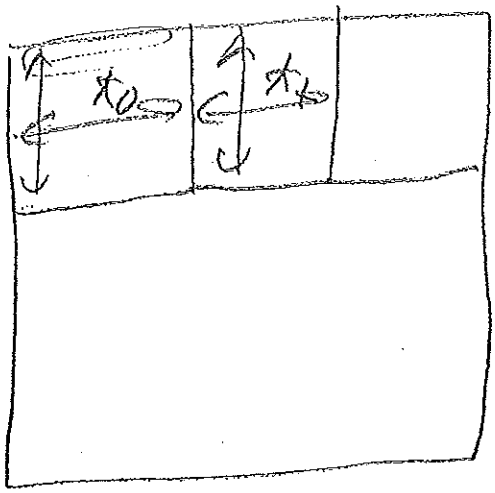


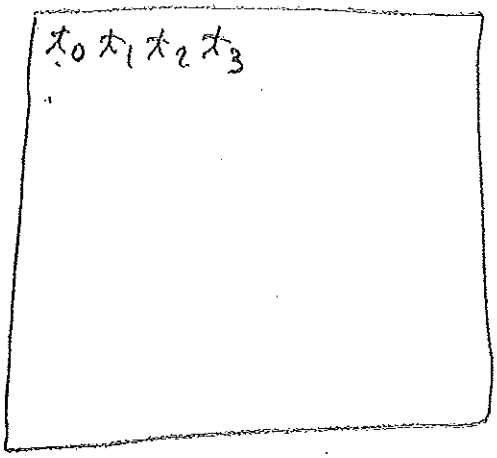
2D



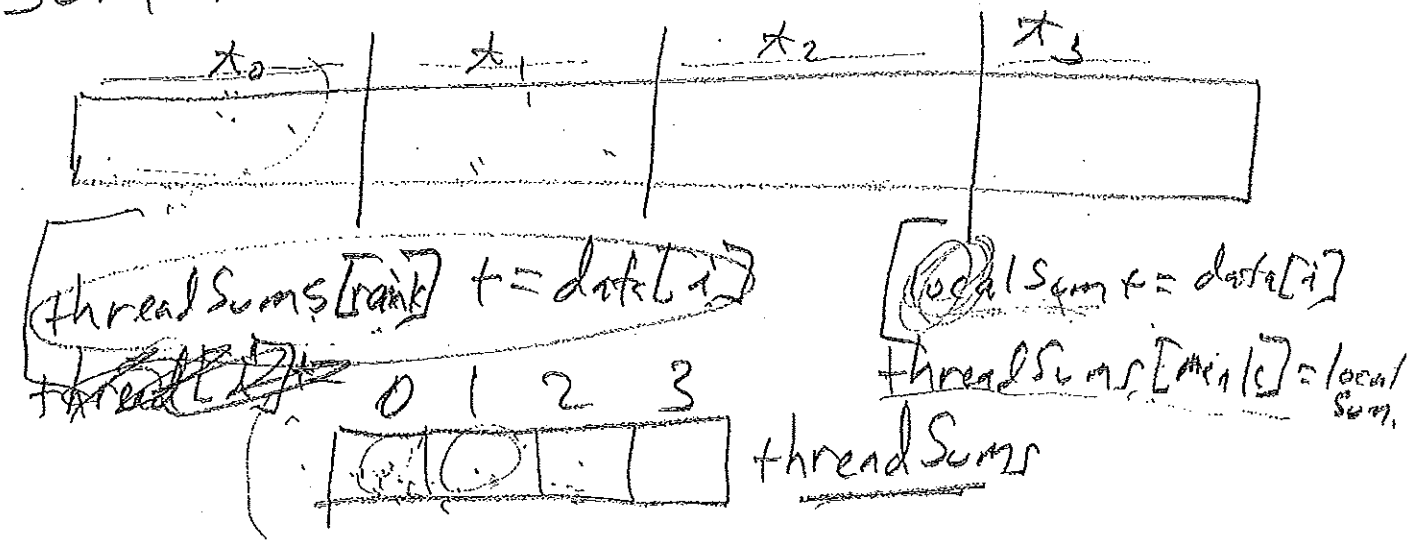


2D

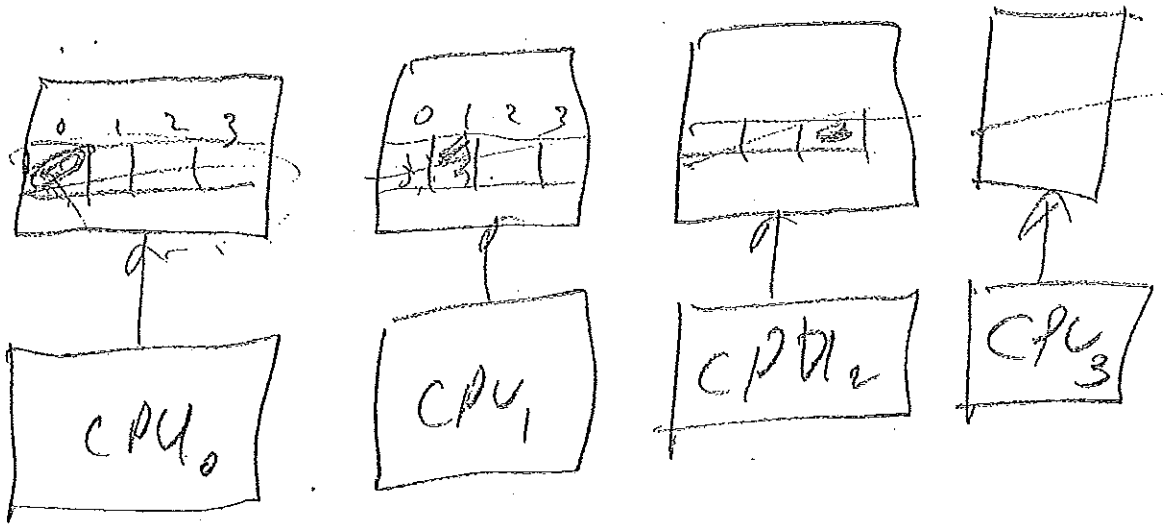




SUM 1D



false sharing



4.11 Thread Safety

~~rand()~~ State, remember last value

rand_r()

Week 9 Discussion Questions
Chapter 6.

Answer the following textbook problems:

6.1

6.17

6.20 a, b, and c **without any implementation/coding**

6.1. In each iteration of the serial n-body solver, we first compute the total force on each particle, and then we compute the position and velocity of each particle. Would it be possible to reorganize the calculations so that in each iteration we did all of the calculations for each particle before proceeding to the next particle? That is, could we use the following pseudocode?

```
for each timestep
   for each particle {
      Compute total force on particle;
      Find position and velocity of particle;
      Print position and velocity of particle;
   }
```

If so, what other modifications would we need to make to the solver? If not, why not?

6.17. **a.** Use Figure 6.10 to determine the maximum number of records that would be on the stack at any one time in solving a four-city TSP. (*Hint*: Look at the stack after branching as far as possible to the left).

b. Draw the tree structure that would be generated in solving a five-city TSP.

c. Determine the maximum number of records that would be on the stack at any one time during a search of this tree.

d. Use your answers to the preceding parts to determine a formula for the maximum number of records that would be on the stack at any one time in solving an n-city TSP.

6.20. Suppose the stack on process/thread A contains k tours.

a. Perhaps the simplest strategy for implementing stack splitting in TSP is to pop $k/2$ tours from A's existing stack and push them onto the new stack. Explain why this is unlikely to be a good strategy.

b. Another simple strategy is to split the stack on the basis of the cost of the partial tours on the stack. The least-cost partial tour goes to A. The second cheapest tour goes to `new_stack`. The third cheapest goes to A, and so on. Is this likely to be a good strategy? Explain your answer.

c. A variation on the strategy outlined in the preceding problem is to use average cost per edge. In average cost per edge, the partial tours on A's stack are ordered according to their cost divided by the number of edges in the partial tour. Then the tours are assigned in round-robin fashion to the stacks, that is, the cheapest cost per edge to A, the next cheapest cost per edge to `new_stack`, and so on. Is this likely to be a good strategy? Explain your answer.

Implement the three strategies outlined here in one of the dynamic load-balancing codes. How do these strategies compare to each other and the strategy outlined in the text? How did you collect your data?

Learning Objectives:

- Analyze larger pthread programs for key design decisions, and sources of parallel overhead.
- Estimate the run-time of larger problems from timings of smaller instances using big-oh notation.
- Contrast the performance improvements of parallelization vs. algorithmic performance improvements.

To start the lab:

- read section 6.1 in the textbook
- watch the Lab 9 Video on the eLearning system
- download `lab9.zip` from the eLearning system and unzip/extract it locally on your computer
- copy the `lab9` directory to `fermi.cs.uni.edu` using a secure ftp client (winSCP, FileZilla, scp, etc.)
- log-on to `fermi1.cs.uni.edu` using Putty/ssh

Part A: Using an editor on `fermi1.cs.uni.edu` open the file `lab9/nbody_basic.c` which contains a sequential C program for the "basic" n-body problem (i.e., "basic" means that it computes the whole 2D of force calculations without making use of its symmetry $f_{ij} = -f_{ji}$).

Answer the following questions about the `nbody_basic.c` program.

a) How is the "current state" of the system (i.e., all the particles' mass, position (x, y coordinates), and velocity (x, y components)) stored in the `main` function? (Describe the data structures used)

Recall (p. 273 of the textbook) the whole 2D of force calculations, where row i are the forces on partical i by the all particles::

$$\begin{bmatrix} 0 & \mathbf{f}_{01} & \mathbf{f}_{02} & \cdots & \mathbf{f}_{0,n-1} \\ -\mathbf{f}_{01} & 0 & \mathbf{f}_{12} & \cdots & \mathbf{f}_{1,n-1} \\ -\mathbf{f}_{02} & -\mathbf{f}_{12} & 0 & \cdots & \mathbf{f}_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -\mathbf{f}_{0,n-1} & -\mathbf{f}_{1,n-1} & -\mathbf{f}_{2,n-1} & \cdots & 0 \end{bmatrix}$$

b) Why are the diagonal values 0?

c) Why is there symmetry (i.e., $f_{ij} = -f_{ji}$) in the calculations?

d) In the `main` function how can the above force calculation results be stored in the 1-dimensional array `forces`?

e) Compile and run both sequential versions ("basic" and "reduced") of the n-body programs with randomly generated particle data and varying the number of particles. Compile by:

- `gcc -g -Wall -o nbody_basic nbody_basic.c -lm`
- `gcc -g -Wall -o nbody_red nbody_red.c -lm`

Run as (500 particles, 1000 timesteps, 0.05 timestep size, output results after 1000 timesteps, "g" for randomly generated particle data):

- `./nbody_basic 500 1000 0.05 1000 g`

Vary the program name and number of particles to complete the following timing table:

Command line parameters	Basic version of sequential program: `./nbody_basic`	Reduced version of sequential program: `./nbody_basic`
`./nbody_xyz 500 1000 0.05 1000 g`		
`./nbody_xyz 1000 1000 0.05 1000 g`		
`./nbody_xyz 1500 1000 0.05 1000 g`		
`./nbody_xyz 2000 1000 0.05 1000 g`		

f) Explain why the "reduced" version is faster than the "basic" version.

Part B: Using an editor on `fermi1.cs.uni.edu` open the file `lab9/pth_nbody_basic.c` which contains a parallel pthread C program for the "basic" n-body problem.
Answer the following questions about the `pth_nbody_basic.c` program.

a) When pthreads start running (i.e., function `Thread_work`), the first thing they do is call the `Loop_schedule` which "returns"/sets values for `first`, `last`, and `incr`. Assume `thread_count` = 4, n = 1000 and BLOCK, what would `Loop_schedule` "return" for `first`, `last`, and `incr` for each `my_rank` value?

`my_rank`	`first` value "returned"	`last` value "returned"	`incr` value "returned"
0			
1			
2			
3			

b) How are the values of `first`, `last`, and `incr` used in the inner-for-loops of the `Thread_work` function to BLOCK partition the work among threads?

c) Assume `thread_count` = 4, n = 1000 and **CYCLIC**, what would `Loop_schedule` "return" for `first`, `last`, and `incr` for each `my_rank` value?

`my_rank`	`first` value "returned"	`last` value "returned"	`incr` value "returned"
0			
1			
2			
3			

d) How are the values of `first`, `last`, and `incr` used in the inner-for-loops of the `Thread_work` function to `CYCLIC` partition the work among threads?

Compile `pth_nbody_basic.c` as is without modification (`BLOCK` partition the work among threads) by:
`gcc -g -Wall -o pth_nbody_basic pth_nbody_basic.c -lm -lpthread`
e) Complete the middle column in the below table by varying the number of threads.
Edit `pth_nbody_basic.c` to `CYCLIC` partition the work among threads, re-compile, and complete the right column in the below table by varying the number of threads.

Command line parameters	BLOCK partition the work among threads	CYCLIC partition the work among threads
`./pth_nbody_basic 1 2000 1000 0.05 1000 g`		
`./pth_nbody_basic 2 2000 1000 0.05 1000 g`		
`./pth_nbody_basic 4 2000 1000 0.05 1000 g`		
`./pth_nbody_basic 8 2000 1000 0.05 1000 g`		

f) Explain why `BLOCK` partitioning the work among threads is faster than `CYCLIC` partitioning.

Part C: Using an editor on `fermi1.cs.uni.edu` open the file `lab9/pth_nbody_red.c` which contains a parallel pthread C program for the "reduced" n-body problem (i.e., "reduced" means that it avoids computes the whole 2D of forces by making use of its symmetry $f_{ij} = -f_{ji}$). Answer the following questions about the `pth_nbody_red.c` program.

When pthreads start running (i.e., function `Thread_work`), the first thing they do is call the function `Loop_schedule` **twice**:

- first with `BLOCK` which "returns" for `bfirst`, `blast`, and `bincr`
- second with `CYCLIC` which "returns" for `cfirst`, `clast`, and `cincr`

a) How are the values of `bfirst`, `blast`, and `bincr` used in the inner-for-loops of the `Thread_work` function?

b) How are the values of `cfirst`, `clast`, and `cincr` used in the inner-for-loops of the `Thread_work` function?

c) Explain why using both types of partitioning of work among threads is fastest for the "reduced" version.

Compile `pth_nbody_red.c` as is without modification by:

```
gcc -g -Wall -o pth_nbody_red pth_nbody_red.c -lm -lpthread
```

d) Complete the middle column in the below table by varying the number of particles.

Command line parameters	Time in seconds
`./pth_nbody_red 1 2000 1000 0.05 1000 g`	
`./pth_nbody_red 2 2000 1000 0.05 1000 g`	
`./pth_nbody_red 4 2000 1000 0.05 1000 g`	
`./pth_nbody_red 8 2000 1000 0.05 1000 g`	

e) Explain why `pth_nbody_red.c` is faster than `pth_nbody_basic.c` with BLOCK partitioning of work.

Submit lab9.zip containing question answers on the eLearning system

Learning Objectives:

- Contrast the performance improvements of parallelization vs. algorithmic performance improvements.
- Perform maintanence on larger pthread program to improve run-time.

To start the homework:

- read section 6.2 in the textbook dealing with the pthread solution of the traveling sales person (TSP) problem
- watch the TSP Video on the eLearning system
- download `hw9.zip` from the eLearning system and unzip/extract it locally on your computer
- copy the `hw9` directory to `fermi.cs.uni.edu` using a secure ftp client (winSCP, FileZilla, scp, etc.)
- log-on to `fermi1.cs.uni.edu` using Putty/ssh

The `hw9` directory contains several TSP programs:

- tsp_iter1.c and tsp_iter2.c - sequential solutions to the TSP problem that do not use recursion from the textbook
- tsp_rec_with_pruning.c - my sequential solution to the TSP problem that uses recursion, but does more pruning of the search-tree. (as described in the TSP Video) It's *promising* function calculates a bound on the best solution possible from a partial tour by summing the minimum edges leaving vertices not already on the partial tour.
- pth_tsp_stat.c - the textbook's parallel pthread solution that statically allocates work to ptheads
- pth_tsp_dyn.c - the textbook's parallel pthread solution that dynamically allocates work to ptheads
- graph4.dat, graph5.dat, and graph17.dat - data files containing digraphs: graph4.dat is the 4-city graph from Figure 6.9, graph5.dat is the 5-city graph from the TSP Video, and graph17.dat contains a larger problem for testing.

Some timing results on `fermi1.cs.uni.edu` using graph17.dat and compiler optimization -O3 are:

Program	Notes on command-line parameters	Time is seconds
./tsp_iter1 graph17.dat		3697.2 seconds (1 hr, 1 min., 37.2 sec)
./pth_tsp_dyn 8 graph17.dat 8	8 threads and min. split size of 8	603.7 seconds
./tsp_rec_with_pruning graph17.dat		68.4 seconds

Homework #9 Description:

Notice that the `tsp_rec_with_pruning.c` program is substantially faster than even the `pth_tsp_dyn.c` program using 8 threads. The `tsp_rec_with_pruning.c` is able to prune away much of the search-tree because it's *promising* function calculates a bound on the best solution possible from a partial tour by summing the minimum edges leaving vertices not already on the partial tour. All of the textbook programs use a much simpler `Feasible` function.

Your task is to incorporate the better promising function calculation from `tsp_rec_with_pruning.c` into the `tsp_iter1.c` and `pth_tsp_dyn.c` programs.

Complete table below by timing results on `fermi1.cs.uni.edu` using graph17.dat and compiler optimization -O3 are:

Program	Notes on command-line parameters	Time is seconds
./tsp_iter1 graph17.dat		
./pth_tsp_dyn 2 graph17.dat 8	2 threads and min. split size of 8	
./pth_tsp_dyn 4 graph17.dat 8	4 threads and min. split size of 8	
./pth_tsp_dyn 8 graph17.dat 8	8 threads and min. split size of 8	
./pth_tsp_dyn 16 graph17.dat 8	16 threads and min. split size of 8	

Submit hw9.zip containing a completed timing table and completed program (hw9.c) on the eLearning system

Comp. Arch. Week 9 ~~Monday~~ Tuesday Video

Scalable Algorithmic Techniques - focuses on data parallelism instead of task parallelism since task parallelism does scale well with # of processors, P (i.e., the number of tasks determines the parallelism -- however, each task can often have data parallelism)

General steps for designing parallel programs ("Foster's methodology"):
1. Partitioning: Divide computation and the data operated on into small tasks. Here the focus is identifying tasks that can be executed in parallel. (We might have true (RAW) data dependencies to consider.)
2. Communication: Determine what communication is needed to be carried out among the tasks identifed in step 1
3. Agglomeration or aggregation: Combine tasks and communications identified in the previous steps into larger tasks. For example, if task A must be executed before task B can be executed, it may make sense to aggregate them into a single composite task.
4. Mapping: Assign the composite tasks identified in the previous step to processes/threads. This should be done so that communication is minimized, and each process/thread get roughly the same amount of work

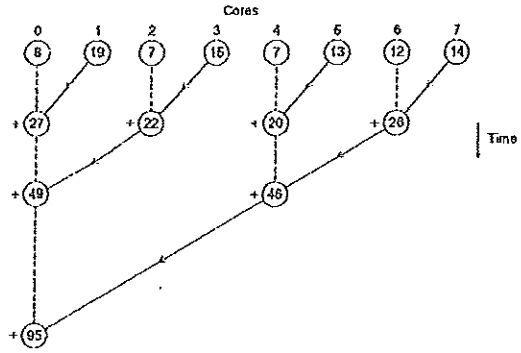
Guiding principle: *Parallel programs are more scalable when they emphasize blocks of computation -- typically the larger the block the better -- that minimize the inter-thread dependencies.*

Example: applies this guiding principle to array summation via tree-reduction where # processors P << n (array size).

Approach (1)	Approach (2)
• Create n/2 logically threads evenly across the P processors, so each processor had n/2P threads. • Have the logical threads perform the binary-tree reduction	• Create P threads -- one for each processor • Have each thread sum its n/P values in a local sum • Have the P threads perform the tree-reduction on the P local sums (see below diagram)

a) Why is approach 2 better?

Approach (2) for tree-reduction, where:

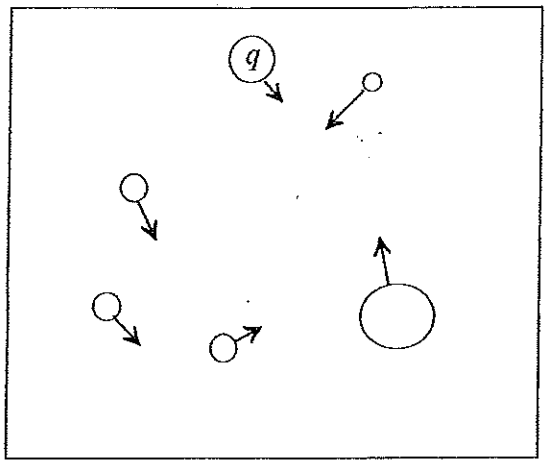


Dynamic Allocation of Data and Work: it might be hard or impossible to statically assign an even workload for several reasons:
- server processes client requests - requests determine the amount of work
- dynamic work created on the fly as the computation proceeds
- size of static data does not reflect amount of computation (i.e., hard to load balance)

Work Queue Solution - data structure for dynamically assigning work to threads or processes. A thread/process *producing* extra work puts it into the work queue. An idle thread/process remove (*consumes*) work from the queue to keep busy.

Types of work queues might depend on the problem being solved. Some options:
- FIFO queue - add new items to rear and remove items from the front
- LIFO stack - add new items to the top of the stack and remove items from the top (e.g., depth-first search)
- randomized queue - remove items randomly from queue
- priority queue - each item has a priority associated with it. Remove the item with highest priority (e.g, best-first search)

Chapter 6 has a couple larger "real-world" examples to demonstrate parallel program development. The first one is the n-body problem where we calculate the movement of n-bodies (e.g., n objects/planets in space, or n particles in a contain) over time. To be concrete, the book considers the motion of planets or stars in a 2D space.



a) What initial information would we need to know about each planet to calculate their motion over time?

$6.673 \times 10^{-11} N \cdot \frac{m}{Kg^2}$

b) How do planets "interact" to effect their motions?

$$F_{ij} = G\frac{m_i m_j}{d^2}$$

Serial code:

```
Get input data;
for each timestep {
   if (timestep output) Print positions and velocities of particles;
   for each particle q
      Compute total force on q;
   for each particle q
      Compute position and velocity of q;
}
Print positions and velocities of particles;
```

The for each particle q:
Computer total force on q"
code needs to perform individual force calculations:
Nodes:

- row 0 are the forces on particle 0 by other particles, etc.
- matrix is "symmetric", except opposite forces are negated (two versions: *basic* and *reduced* utilitizing symmetry)

$$\begin{bmatrix} 0 & f_{01} & f_{02} & \cdots & f_{0,n-1} \\ -f_{01} & 0 & f_{12} & \cdots & f_{1,n-1} \\ -f_{02} & -f_{12} & 0 & \cdots & f_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -f_{0,n-1} & -f_{1,n-1} & -f_{2,n-1} & \cdots & 0 \end{bmatrix}$$

If we want to parallelize the calculation by applying the "guiding principle," then how might we map calculations to threads?

```
/* Function:  Thread_work
 * Purpose:   Execute individual thread's contribution
 *            to finding the positions and velocities
 *             of the particles.
 * In arg:    rank:   thread's rank (0, 1, . . . ,
thread_count-1)
 * Globals:   thread_count (in):  */
void* Thread_work(void* rank) {
   long my_rank = (long) rank;
   int step;    /* Current step      */
   int part;    /* Current particle  */
   double t;    /* Current Time      */
   int first;   /* My first particle */
   int last;    /* My last particle  */
   int incr;    /* Loop increment    */

   Loop_schedule(my_rank, thread_count, n, BLOCK,
                 &first, &last, &incr);
   for (step = 1; step <= n_steps; step++) {
      t = step*delta_t;
      /* Particle n-1 will have all forces computed
       * after call to Compute_force(n-2, . . .) */
      for (part = first; part < last; part += incr)
         Compute_force(part);
      Barrier();
      for (part = first; part < last; part += incr)
         Update_part(part);
      Barrier();
#     ifndef NO_OUTPUT
      if (step % output_freq == 0 && my_rank == 0) {
         Output_state(t);
      }
#     endif
   }  /* for step */

   return NULL;
}  /* Thread_work */
```

```
/*-----------------------------------------------------------
 * Function:  Loop_sched
 * Purpose:   Return the parameters for a block or a cyclic
schedule
 *            for a for loop
 * In args:
 *    my_rank:       rank of calling thread
 *    thread_count:  number of threads
 *    n:             number of loop iterations
 *    sched:         schedule:  BLOCK or CYCLIC
 * Out args:
 *    first_p:       pointer to first loop index
 *    last_p:        pointer to value greater than last index
 *    incr_p:        loop increment
 */
void Loop_schedule(int my_rank, int thread_count, int n, int sched,
                   int* first_p, int* last_p, int* incr_p) {
   if (sched == CYCLIC) {
      *first_p = my_rank;
      *last_p = n;
      *incr_p = thread_count;
   } else {  /* sched == BLOCK */
      int quotient = n/thread_count;
      int remainder = n % thread_count;
      int my_iters;
      *incr_p = 1;
      if (my_rank < remainder) {
         my_iters = quotient + 1;
         *first_p = my_rank*my_iters;
      } else {
         my_iters = quotient;
         *first_p = my_rank*my_iters + remainder;
      }
      *last_p = *first_p + my_iters;
   }

}  /* Loop_schedule */
```

```
/* Function:  Compute_force
 * Purpose:   Compute the total force on particle part.  This
 *            version does 'not' exploit the symmetry.
 * In arg:  part: the particle on which we're computing the total force
 * Global vars:
 *    curr (in):  current state of the system:  curr[i] stores the mass,
 *       position and velocity of the ith particle
 *    n (in):     number of particles
 *    forces (out): forces[i] stores the total force on the ith particle
*/
void Compute_force(int part) {
   int k;
   double mg;
   vect_t f_part_k;
   double len, len_3, fact;
   forces[part][X] = forces[part][Y] = 0.0;
   for (k = 0; k < n; k++) {
      if (k != part) {
         /* Compute force on part due to k */
         f_part_k[X] = curr[part].s[X] - curr[k].s[X];
         f_part_k[Y] = curr[part].s[Y] - curr[k].s[Y];
         len = sqrt(f_part_k[X]*f_part_k[X] + f_part_k[Y]*f_part_k[Y]);
         len_3 = len*len*len;
         mg = -G*curr[part].m*curr[k].m;
         fact = mg/len_3;
         f_part_k[X] *= fact;
         f_part_k[Y] *= fact;

         /* Add force in to total forces */
         forces[part][X] += f_part_k[X];
         forces[part][Y] += f_part_k[Y];
      }
   }
}  /* Compute_force */
```

```
/* Function:  Update_part
 * Purpose:   Update the velocity and position for particle part
 * In arg:       part:    the particle we're updating
 * Global vars:  forces (in):   forces[i] stores the total force on
 *                               the ith particle
 *               n (in):        number of particles
 *               curr (in/out): curr[i] stores the mass, position and
 *                              velocity of the ith particle */
void Update_part(int part) {
   double fact = delta_t/curr[part].m;

   curr[part].s[X] += delta_t * curr[part].v[X];
   curr[part].s[Y] += delta_t * curr[part].v[Y];
   curr[part].v[X] += fact * forces[part][X];
   curr[part].v[Y] += fact * forces[part][Y];
}  /* Update_part */
```

```
#define DIM 2  /* Two-dimensional system */
#define X 0    /* x-coordinate subscript */
#define Y 1    /* y-coordinate subscript */

const int BLOCK = 0; /*Block partition of loop iterations*/
const int CYCLIC = 1; /*Cyclic partition of loop iter.s */

typedef double vect_t[DIM]; /*Vector type for position,etc*/

struct particle_s {
   double m;  /* Mass     */
   vect_t s;  /* Position */
   vect_t v;  /* Velocity */
};
```

Comp. Arch. Week 9 ~~Friday~~ Thursday Video

Ch 6: *Traveling Salesperson Problem* (TSP)

Handling "Hard" Problems: For many optimization problems (e.g., *NP-Complete* problems: TSP, knapsack, job-scheduling), the best known algorithms have run-time's that grow exponentially, $O(2^n)$. Thus, you could wait centuries for the solution of all but the smallest problems!

Ways to handle these "hard" problems:

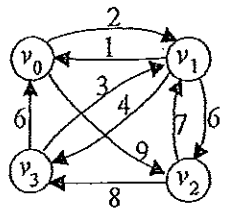
- Find the best (or a good) solution "quickly" to avoid considering the vast majority of the 2^n worse solutions, e.g, Backtracking and Branch-and-Bound
- See if a restricted version of the problem meets your needed that might have a tractable (polynomial, e.g., $O(n^3)$) solution. e.g., Fractional Knapsack problem, TSP problem satisfying the triangle inequality
- Use an approximation algorithm to find a good, but not necessarily optimal solution

Backtracking general idea:

- Search the "*state-space tree*" using depth-first search to find a suboptimal solution quickly
- Use the best solution found so far to prune partial solutions that are not "promising,", i.e., cannot lead to a better solution than one already found.

The goal is to prune enough of the state-space tree (exponential is size) that the optimal solution can be found in a reasonable amount of time. However, in the worst case, the algorithm is still exponential.

Ch 6: *Traveling Salesperson Problem* (TSP) -- Find an optimal (i.e., minimum length) tour when at least one tour exists. A *tour* (or *Hamiltonian circuit*) is a path from a vertex back to itself that passes through each of the other vertices exactly once. (Since a tour visits every vertice, it does not matter where you start, so we will start at v_0.)



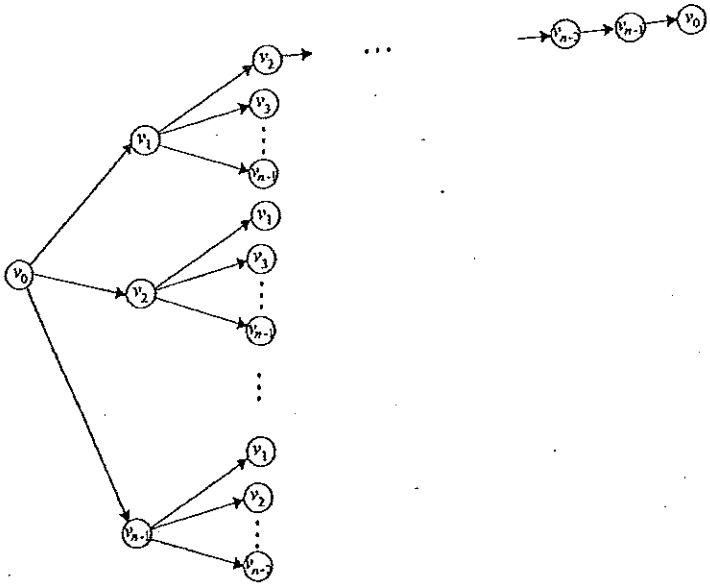
0. What are the length of the following tours?

a) $[v_0, v_1, v_2, v_3, v_0]$

b) $[v_0, v_2, v_1, v_3, v_0]$

c) List another tour starting at v_0 and its length.

d) For a graph with "n" vertices ($v_0, v_1, v_2, \ldots, v_{n-1}$), one possible approach to solving TSP would be to brute-force generate all possible tours to find the minimum length tour. "Complete" the following decision tree to determine the number of possible tours.



To speed the backtracking algorithm, we can prune unpromising branches. The general recursive backtracking algorithm for optimization problems (e.g., TSP, knapsack, job-scheduling) looks something like:

```
Backtrack( recursionTreeNode p ) {

    treeNode c;
    for each child c of p do                          # each c represents a possible choice
        if promising(c) then                          # c is "promising" if it could lead to a better solution
            if c is a solution that's better than best then  # check if this is the best solution found so far
                best = c                              # remember the best solution
            else
                Backtrack(c)                          # follow a branch down the tree
            end if
        end if
    end for
} // end Backtrack
```

General Notes about Backtracking sequentially:

- The depth-first nature of backtracking only stores information about the current branch being explored so the memory usage is "low"
- Each node of the state-space (recursive-call) tree maintains the state of a partial solution. In general the partial solution state consists of potentially large arrays that change little between parent and child. To avoid having multiple copies of these arrays, a single "global" state is maintained which is updated before we go down to the child (via a recursive call) and undone when we backtrack to the parent.
- We could use the concept of backtracking without recursion by using a stack to maintain a collection of unexplored choices. Thus, we would simulate the run-time stack to drive the backtracking algorithm.

1. Consider customizing the above Backtrack template for the TSP problem.

a) What would the "for each child c" loop iterate over? (What problem instance information is needed?)

b) What state information is needed at each node?

c) What criteria can be used to determine if a child node (c) is NOT promising?

d) What information is needed by our promising function?

2. To parallelize the Backtracking "tree" search, we want to eliminate the recursion and replace the run-time stack by our own stack. The TSP algorithm might looks something like:

```
startState = Create partialTour with only v0
push(startState)
while stack is not empty do
   currentPartialTour = pop( )
   if currentPartialTour contains n cities then
      if currentPartialTour distance < bestTour distance then
          bestTour = currentPartialTour
   else
      for city not already on currentPartialTour do
          if promising( currentPartialTour, city) then
              push( new lengthened partial tour with city added )
          end if
      end for
   end if
   free( currentPartialTour)
end while
```

Each stack entry contains complete information about a partial tour: an array of cities visited and the distance of the partial tour.

a) What information would we need to maintain for each partial tour?

b) If we want to parallelize TSP with pthreads, how might we statically allocate the work to each thread?

c) If we want to parallelize TSP with pthreads, how might we dynamically allocate the work to each thread?

3. With pthreads we can statically allocate the work (pth_tsp_stat.c) to each thread by doing a breadth-first search (BFS) of tree until we have enough tree nodes for all the threads. Each thread can run its own search on its subtree with all threads updating a common best tour. The BFS of a tree uses a queue of tree nodes. Draw the queue for the graph shown.

4. With pthreads we can dynamically allocate the tree search (pth_tsp_dyn.c) by allowing a thread which runs out of work to obtain work from another thread. A thread with work can split its stack and give work to the thread which ran out. A thread running out of work can wait on a condition variable, until another thread can split its work and then signal the waiting thread. How many threads will be waiting for work when the last thread runs out of work?

5. How can a thread with work split its stack into two roughly equal stacks of work?

tsp_iter1.c which uses a stack to avoid recursion but has a simple pruning function.

```
Best tour:
0 1 2 9 7 5 3 6 4 14 8 15 16 11 10 13 12 0
Cost = 87
Elapsed time = 1.287522e+04 seconds
```

3 hours, 34 minutes and 35 seconds

pth_tsp_dyn.c with 8 threads on ITT 335 i7 computers splitting the work with a parameter 8:

```
Best tour: 0 9 2 1 7 5 3 6 4 14 8 15 16 11 10 13 12 0

Cost = 87
Elapsed time = 2.418689e+03 seconds
```

Speedup of 5.3 with 8 pthreads.

My recursive DFS Backtracking using the sum of the minimum edges out of the remaining cities to bound the solution for pruning.

```
<116 pandora:~/cs2420/codeExamples/tsp_pthread >g++ -O3 -o tsp_backtrack_pruning tsp.cpp
<117 pandora:~/cs2420/codeExamples/tsp_pthread >tsp_backtrack_pruning < bri7.dat
The adjacency matrix is:
     0       3       5      48      48       8       8       5       5       3       3 1000000       3       5       8       8       5
     3       0       3      48      48       8       8       5       5 1000000 1000000       3 1000000       3       8       8       5
     5       3       0      72      72      48      48      24      24       3       3       5       3 1000000      48      48      24
    48      48      74       0 1000000       6       6      12      12      48      48      48      48      74       6       6      12
    48      48      74 1000000       0       6       6      12      12      48      48      48      48      74       6       6      12
     8       8      50       6       6       0 1000000       8       8       8       8       8       8      50 1000000 1000000       8
     8       8      50       6       6 1000000       0       8       8       8       8       8       8      50 1000000 1000000       8
     5       5      26      12      12       8       8       0 1000000       5       5       5       5      26       8       8 1000000
     5       5      26      12      12       8       8 1000000       0       5       5       5       5      26       8       8 1000000
     3 1000000       3      48      48       8       8       5       5       0 1000000       3 1000000       3       8       8       5
     3 1000000       3      48      48       8       8       5       5 1000000       0       3 1000000       3       8       8       5
1000000       3       5      48      48       8       8       5       5       3       3       0       3       5       8       8       5
     3 1000000       3      48      48       8       8       5       5 1000000 1000000       3       0       3       8       8       5
     5       3 1000000      72      72      48      48      24      24       3       3       5       3       0      48      48      24
     8       8      50       6       6 1000000 1000000       8       8       8       8       8       8      50       0 1000000       8
     8       8      50       6       6 1000000 1000000       8       8       8       8       8       8      50 1000000       0       8
     5       5      26      12      12       8       8 1000000 1000000       5       5       5       5      26       8       8       0
Execution time: 88.8106 seconds.
Best tour length is 87
The best tour is:  0 1 2 9 7 5 3 6 4 14 8 15 16 11 10 13 12 0
<118 pandora:~/cs2420/codeExamples/tsp_pthread >
```

Unit 3 Quiz Fall 2016

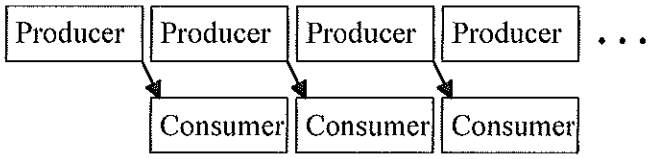
Name:_____

1) Consider two kinds of threads that are linked such that:

- a producer thread produces some data ("work"), and
- a consumer thread consumes the data (does the "work")

Without concurrency they could alternate:

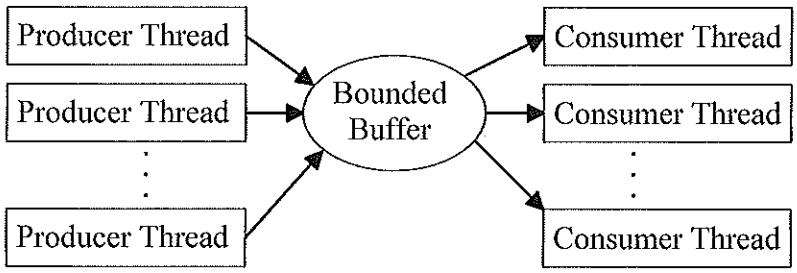
With concurrency they could pipeline:



a) What is the advantage of the pipelined version?

b) In the pipelined version, how does the rate the producer produces data compare to the rate the consumer consumes it?

2) A bounded buffer can be used to store some (limited by some bound) data between multiple producer and multiple consumer threads:



a) What is the advantage of the bounded buffer approach over pipelining producer-data directly to the consumer thread?

b) In the bounded buffer approach when would the producer thread need to block?

c) If we had a lot of producer and consumer threads, then the bounded buffer might become a bottleneck. How might this bottleneck be alleviated? (briefly describe your approach)

2) Answer the following questions about the `bufferAdd` and `bufferRemove` functions from lab 8.

```
// Global Bounded Buffer
WORK buffer[SIZE];
int count = 0;
int front = -1;
int rear = 0;
pthread_mutex_t lock;
pthread_cond_t nonFull;
pthread_cond_t nonEmpty;
```

```
void bufferAdd(WORK item) {

  pthread_mutex_lock(&lock);
  while(count == SIZE) {
    while(pthread_cond_wait(&nonFull,&lock)!= 0);
  } // end while
  if (count == 0) {
    front = 0;
    rear = 0;
  } else {
    rear = (rear + 1) % SIZE;
  } // end if
  buffer[rear] = item;
  count++;
  pthread_cond_signal(&nonEmpty);
  pthread_mutex_unlock(&lock);

} // end bufferAdd
```

```
WORK bufferRemove() {
  WORK returnValue;

  pthread_mutex_lock(&lock);
  while(count == 0) {
    while(pthread_cond_wait(&nonEmpty,&lock) != 0);
  } // end while
  returnValue = buffer[front];
  front = (front + 1) % SIZE;
  count--;
  pthread_cond_signal(&nonFull);
  pthread_mutex_unlock(&lock);
  return returnValue;
}  // end bufferRemove
```

a) When a Producer thread calls `bufferAdd` and the `buffer` is full, it gets suspended and waits at the condition variable `notFull`. How does it get woken up?

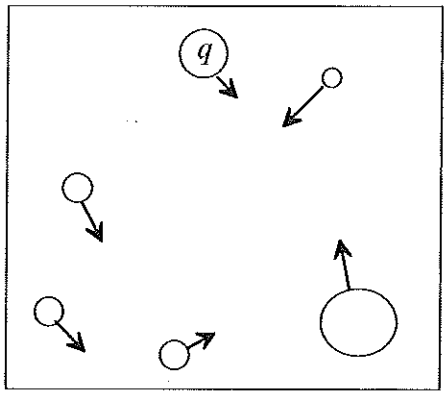
b) What is the purpose of **each** of the while-loops in the `bufferAdd`?

- `while(count == SIZE):`

- `while(pthread_cond_wait(&nonFull, &lock) != 0);`

c) Both the `bufferAdd` and `bufferRemove` functions start with `pthread_mutex_lock(&lock);` and end with `pthread_mutex_unlock(&lock);` which makes them a single critical section (i.e., guarantees that only a single Producer or Consumer thread is modifying the shared variables: `buffer`, `count`, `front`, and `rear`). However, suppose the buffer is full and a Producer is suspended and waits at the condition variable `notFull` with is inside the critical section. How does a Consumer thread get past the `pthread_mutex_lock(&lock);` to remove an item from the buffer?

3) The n-body problem simulates the motion of "planets/particles" in a 2D space.



The serial code:

```
Get input data:
for each timestep {
   if (timestep output) Print positions and velocities of particles;
   for each particle q
      Compute total force on q;
   for each particle q
      Compute position and velocity of q;
}
Print positions and velocities of particles;
```

The for each particle q:
Computer total force on q"
code needs to perform individual force calculations:
Here, row i are the forces on particle i by all other particles, i.e. $\mathbf{f}_{ij}$ is the force of particle j on particle i.

$$\begin{bmatrix} 0 & \mathbf{f}_{01} & \mathbf{f}_{02} & \cdots & \mathbf{f}_{0,n-1} \\ -\mathbf{f}_{01} & 0 & \mathbf{f}_{12} & \cdots & \mathbf{f}_{1,n-1} \\ -\mathbf{f}_{02} & -\mathbf{f}_{12} & 0 & \cdots & \mathbf{f}_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -\mathbf{f}_{0,n-1} & -\mathbf{f}_{1,n-1} & -\mathbf{f}_{2,n-1} & \cdots & 0 \end{bmatrix}$$

a) Except for the negative-sign, why is this "matrix" symmetric around the diagonal?

Recall the two textbook versions of the n-body program:

- *basic* - which did not utilize the matix symmetry and calculated both $\mathbf{f}_{ij}$ and $\mathbf{f}_{ji}$
- *reduced* - which only calculated the forces above the diagonal.

b) When assigning force calculations of particles to threads in the *reduced* version, why is a cyclic distribution better than a block distribution? (For example: 4 threads with n = 1024 would have a cyclic distribution for thread 0 being assigned particles 0, 4, 8, 12, ... 1020, and a block distribution for thread 0 being assigned particles 0 - 255.)