Programming Shared-memory Machines

Some slides adapted from Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar ``Introduction to Parallel Computing'', Addison Wesley, 2003.

Running example

$$sum = \Sigma_{i=1}^{n} f(i)$$

- Algorithmic parallelism:
  - map: function evaluations *f(i)* can be done in parallel
  - reduce: if addition is associative, *f(i)* values can be summed in parallel in O(log(n)) steps
    - we will not worry about exploiting this parallelism
- How do we exploit this algorithmic parallelism using shared-memory programming?
- We will use Pthreads and OpenMP to illustrate concepts
- Pthreads: POSIX threads
- OpenMP:
  - Higher-level API than Pthreads
  - OpenMP programs are often compiled to Pthreads code

Overview

- Thread Basics
- The POSIX Thread API
- Synchronization primitives in Pthreads
  - join
  - locks and try-locks
  - barriers
- Implementing synchronization primitives using atomic instructions
- Deadlocks and how to avoid them

Threads

- Software analog of cores
  - Each thread has its own PC, SP, registers etc.
  - All threads share heap and globals
- Runtime system handles mapping of threads to cores (scheduling)
  - If there are more threads than cores, runtime system will time-slice threads on cores
  - HPC applications: usually #threads = #cores
    - portability: number of threads is usually a runtime parameter
- Threads have names (opaque handles)
  - used to assign different work to different threads
  - used by one thread to refer to another thread

Thread Basics: Creation and Termination

- Program begins execution with main thread
- Creating threads:

```
#include <pthread.h>
int pthread_create (
   pthread_t *thread_handle,
   const pthread_attr_t *attribute,
   void * (*thread_function)(void *),
   void *arg);
```

- Thread is created and it starts to execute thread_function with parameter arg
- Thread handle: opaque name for thread
- Type (void *) is C notation for “raw address” (that is, can point to anything)

Terminating threads

- Thread terminated when:
    - o it returns from its starting routine, or
    - o it makes a call to pthread_exit()
- Main thread
    - – exits with pthread_exit(): other threads will continue to execute
    - – Otherwise: other threads automatically terminated
- Cleanup:
    - – pthread_exit() routine does not close files
    - – any files opened inside the thread will remain open after the thread is terminated.

Example

```
#include <pthread.h>
 #include <stdio.h>
 #include <stdlib.h>
 #define NUM_THREADS 5
 pthread_t handles[NUM_THREADS]; //store opaque handles for threads
 int shortNames[NUM_THREADS];//store short names for threads

 void *PrintHello(void *threadIdPtr) {
   int shortId = * (int *)threadIdPtr;
   printf("\n%d: Hello World!\n", shortId);
   pthread_exit(NULL);
  }

 int main(int argc, char *argv[]) {
   for(int t=0;t<NUM_THREADS;t++){
     printf("Creating thread %d\n", t);
     shortNames[t] = t;
     int rc = pthread_create(&handles[t], NULL, PrintHello, &shortNames[t]);
     if (rc){ printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
            }
   }
   pthread_exit(NULL);
 }
```

Output

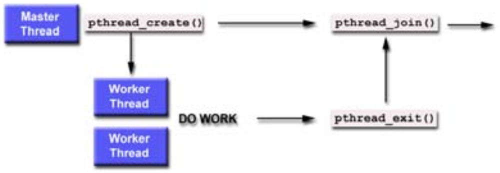
```
Creating thread 0
Creating thread 1

0: Hello World!

1: Hello World!
Creating thread 2
Creating thread 3

2: Hello World!

3: Hello World!
Creating thread 4

4: Hello World!
```

Synchronization

- In most programs, threads need to coordinate their execution to ensure correctness
- Examples:
  - Join: block a given thread until some other thread terminates
  - Critical section: region of code that must be executed by at most one thread at a time
  - Barrier: all threads must reach this point in code before any thread can move ahead

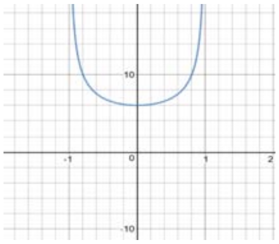
Join

pthread_join (threadid,status)



- The pthread_join() function blocks the calling thread until the specified thread terminates.
- The programmer can obtain the target thread's termination return status if it was specified in the target thread's call to pthread_exit().

Using joins in sum example



$$f(x) = \frac{6}{\sqrt{1-x^2}}$$

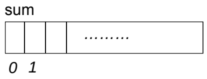
- Estimate value of π using numerical integration $\int_0^{1/2} f(x)dx = \pi$
- Divide interval [0,1/2) into steps of equal size h and compute

$$\sum_{i=0}^{\frac{1}{2h}-1} f(i*h)*h$$

Structure of code

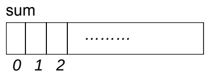
- Main thread creates P worker threads
  - numbered 0,1,2,…P-1
- Thread t
  - computes sum of values for i = t, t+P,t+2P,…
  - writes value into sum[i] where sum is a global array
- Main thread joins with each worker thread and reads its contribution from sum array
- Main thread prints answer after joining with all worker threads

$$\sum_{i=0}^{\frac{1}{2h}-1} f(i*h)*h$$



Code

```
#include <pthread.h>
#include <stdlib.h>
#include <math.h>
#include <stdio.h>

#define MAX_THREADS 512

pthread_t handles[MAX_THREADS];
int shortNames[MAX_THREADS];

void *compute_pi (void *);

int numPoints;
int numThreads;
double step;
double sum[MAX_THREADS];

double f(double x) {
  return (6.0/sqrt(1-x*x));
}
```



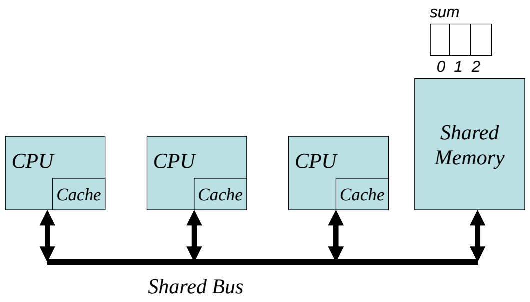
thread i will return its local sum in sum[i]

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

  pthread_attr_t attr;
  pthread_attr_init (&attr);

  double pi = 0.0;
  numPoints = 100000000;
  step = 0.5/numPoints;
  numThreads = atoi(argv[1]); //number of threads is an input

  //create threads and initialize sum array
  for (int i=0; i< numThreads; i++) {
    sum[i] = 0.0;
    shortNames[i] = i;
    pthread_create(& handles[i],&attr,compute_pi,& shortNames[i]);
  }

  //join with threads and add their contributions from sum array
  for (int i=0; i< numThreads; i++) {
    pthread_join(handles[i], NULL);
    pi += sum[i];
  }
  printf("%f\n", pi);
  return 0;

}
```

```
void *compute_pi (void *threadIdPtr) {
  int myId = *(int *)threadIdPtr;

 double mySum =0.0;
 for (int i = myId; i < numPoints; i+=numThreads) {
     double x = step * ((double) i);  // next x
     mySum = mySum + step*f(x);  // Add to local sum
  }
  sum[myId] = mySum; //write to global sum array
}
```

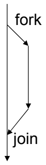
- What happens if loop writes directly to sum[myId] instead of accumulating locally in mySum? See next slide.

False-sharing



Remarks

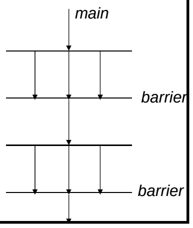
- Style of computing shown in Example 2 is sometimes called fork-join parallelism



- This style of parallel execution in which threads only synchronize at the end is quite rare
- Usually, threads need to synchronize during their execution

Other styles of synchronization

- Mutual exclusion
  - Shared “resource” such as variable or device
  - Only one thread at a time can access resource or update variable
  - Critical section: portion of code that should be executed by only thread at a time
- Barrier
  - Program executes in phases
  - All threads must complete a phase before any thread can execute next phase
  - Insert barrier synchronization between phases



Need for Mutual Exclusion

- Consider variant of sum program
  - Global variable: globalSum
  - Each thread adds its contributions directly to globalSum

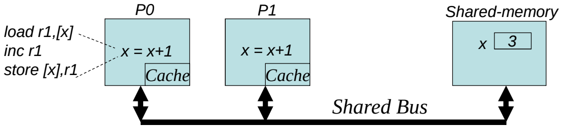
```
for (int i = myId; i < numPoints; i+=numThreads) {
      double x = step * ((double) i);  // next x
      globalSum = globalSum + step*f(x);  // Add to global sum
  }
```

- Data-race: ☹
  - Read and write to globalSum by a thread may be interleaved with reads and writes to globalSum by other threads
  - Example:
    - globalSum value is 0.5
    - Thread 0 has a contribution of 0.1 and thread 1 has a contribution of 0.2
    - Final value after both additions should be 0.8
    - However here is one sequence of possible operations
      - Thread 0 reads value 0.5
      - Thread 1 reads value 0.5
      - Thread 0 adds its contribution and writes 0.6
      - Thread 1 adds its contribution and writes 0.7

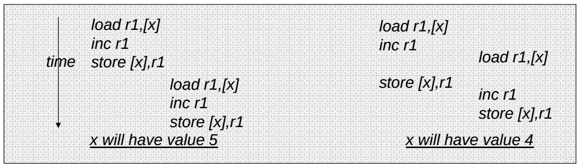
Mutual exclusion

- Basic problem: read/modify/write
  - Shared variable x
  - Two threads want to
    - *read* value of x
    - compute a *new value for x*
    - *write* new value to x
  - Unless you are careful, you get a *data-race*
    - final value can depend on
      - how code is compiled
      - scheduling of threads
      - result may not be what you expect

Coherent caches do not solve data-race problem

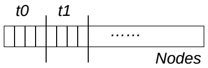


- Final value can be 4 or 5 depending on scheduling of instructions



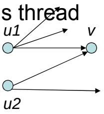
Another example

- Bellman-Ford SSSP
  - assume 16 nodes and 4 threads
  - Work assignment
    - Thread 0 handles nodes 0..3
    - Thread 1 handles nodes 4..7
    - Thread 2 handles nodes 8..11
    - Thread 3 handles nodes 12..15



- Pseudo-code for thread function

```
//compute startNode and endNode for this thread
for node u = startNode to endNode
  for each edge (u,v)
     if (d(u)+length(u,v) < d(v))
       d(v) = d(u)+length (u,v);
```



- What happens if two threads wants to update label of same node v?
  - data-race can cause incorrect results

Solution (I)

- Architecture provides atomic instructions
  - Small collection of read/modify/write instructions operating on ints, doubles, etc.
  - Execute as though all other threads were suspended during execution of atomic instruction
  - Example:
    - swap(addr, reg) //swap value in memory at address addr with value in register reg
- Easy to modify MESI protocol to implement atomic instructions
  - Like write except that line is locked down in cache until instruction completes
  - No other core can steal line until instruction completes

Solution (II):Using atomic operations in code

- Intrinsics: use atomic instructions in code
  - can be tricky to use correctly
- Locks (mutex, spin-lock)
  - programming abstraction implemented by thread libraries
  - locks can be acquired and released by threads
  - implementation uses atomic instructions to guarantee that only thread can acquire lock at a time
  - to enter critical section, thread must acquire lock
  - release lock when exiting critical section
- Let us study how locks can be used to implement critical sections and then dig down to see how they can be implemented using atomic instructions

Mutex in Pthreads

- The Pthreads API provides the following functions for handling mutex-locks:
  - Lock creation

```
int pthread_mutex_init (
   pthread_mutex_t *mutex_lock,
   const pthread_mutexattr_t *lock_attr);
```

  - Acquiring lock

```
int pthread_mutex_lock (
   pthread_mutex_t *mutex_lock);
```

  - Releasing lock

```
int pthread_mutex_unlock (
   pthread_mutex_t *mutex_lock);
```

Using locks

- Lock is implemented by
  - variable with two states: *available* or *not_available*
  - queue that can hold ids of threads waiting for the lock
- Lock acquire:
  - If state of lock is *available*, its state is changed to *not_available*, and control returns to application program
  - If state of lock is *not_available*, thread-id is queued up at the lock, and control returns to application program only when lock is acquired by that thread
  - Key invariant: once a thread tries to acquire lock, control returns to thread only after lock has been awarded to that thread
- Lock release:
  - next thread in queue is informed it has acquired lock, and it can proceed
- “Fairness”: any thread that wants to acquire a lock can succeed ultimately even if other threads want to acquire the lock an unbounded number of times

Correct Mutual Exclusion

- We can now write our sum example as follows:

```
 double globalSum = 0.0;
pthread_mutex_t globalSum_lock;
...
main() {
    ....
    pthread_mutex_init(&globalSum_lock, NULL);
    ....
}
void *compute_pi(void *s) {
    ....
  for (int i = myId; i < numPoints; i+=numThreads) {
     double x = step * ((double) i);  // next x
     double value = step*f(x);
     pthread_mutex_lock(&globalSum_lock);
     globalSum = globalSum + value;  // Add to globalSum
     pthread_mutex_unlock(&globalSum_lock);
   }
```

critical section

Critical sections

- For performance, it is important to keep critical sections as small as possible
- While one thread is within critical section, all others threads that want to enter the critical section are blocked
- It is up to the programmer to ensure that locks are used correctly to protect variables in critical sections

```
Thread A     Thread B     Thread C
 lock(l)      lock(l)
   x:= ..x..    x:= ..x..    x: = …x
unlock(l)    unlock(l)
```

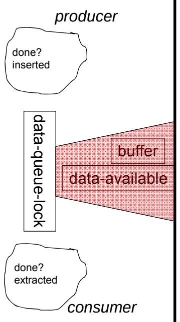
This program may fail to execute correctly because programmer forgot to use locks in Thread C

Producer-Consumer Using Locks

- Two threads
  - Producer: produces data
  - Consumer: consumes data
- Shared buffer is used to communicate data from producer to consumer
  - Buffer can contain one data value (in this example)
  - Flag is associated with buffer to indicate buffer has valid data
- Consumer must not read data from buffer unless there is valid data
- Producer must not overwrite data in buffer before it is read by consumer

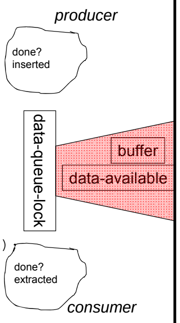
Producer-Consumer Using Locks

```
pthread_mutex_t data_queue_lock;
int data_available; //1 if buffer is full
...
main() {
     ....
     data_available = 0;
     pthread_mutex_init(&data_queue_lock, NULL);
     ....
}
void *producer(void *producer_thread_data) {
     ....
     while (!done()) {
          create_data(&my_data);
          inserted = 0;
          while (inserted == 0) {
               pthread_mutex_lock(&data_queue_lock);
               if (data_available == 0) {
                    insert_data(my_data);
                    data_available = 1;
                    inserted = 1;
               }
               pthread_mutex_unlock(&data_queue_lock);
          }
     }
}
```



Producer-Consumer Using Locks

```
void *consumer(void *consumer_thread_data) {
     int extracted;
     struct data my_data;
     /* local data structure declarations */
     while (!done()) {
          extracted = 0;
          while (extracted == 0) {
               pthread_mutex_lock(&data_queue_lock);
               if (data_available == 1) {
                    extract_data(&my_data);
                    data_available = 0;
                    extracted = 1;
               }
               pthread_mutex_unlock(&data_queue_lock);
          }
          process_data(my_data);
     }
}
```



Types of Mutexes

- Pthreads supports three types of mutexes - normal, recursive, and error-check.
- A normal mutex deadlocks if a thread that already has a lock tries a second lock on it.
- A recursive mutex allows a single thread to lock a mutex as many times as it wants. It simply increments a count on the number of locks. A lock is relinquished by a thread when the count becomes zero.
- An error check mutex reports an error when a thread with a lock tries to lock it again (as opposed to deadlocking in the first case, or granting the lock, as in the second case).
- The type of the mutex can be set in the attributes object before it is passed at time of initialization.

Spin locks/trylocks

- Another kind of lock: trylock.

```
int pthread_mutex_trylock (
      pthread_mutex_t *mutex_lock);
```

- If lock is available, acquire it; otherwise, return a "busy" error code (EBUSY)
- Faster than `pthread_mutex_lock` on typical systems when there is no contention since it does not have to deal with queues associated with locks

Using locks

```
/* Finding k matches in a list */
void *find_entries(void *start_pointer) {
    /* This is the thread function */
    struct database_record *next_record;
    int count;
    current_pointer = start_pointer;
    do {
        next_record = find_next_entry(current_pointer);
        count = output_record(next_record);
    } while (count < requested_number_of_records);
}
int output_record(struct database_record *record_ptr) {
    int count;
    pthread_mutex_lock(&output_count_lock);
    output_count ++;
    count = output_count;
    pthread_mutex_unlock(&output_count_lock);
    if (count <= requested_number_of_records)
        print_record(record_ptr);
    return (count);
}
```

Using spin-locks

```
/* rewritten output_record function */
int output_record(struct database_record
   *record_ptr) {
    int count;
    int lock_status;
    lock_status=pthread_mutex_trylock(&output_count_lock);
    if (lock_status == EBUSY) {
        insert_into_local_list(record_ptr);
        return(0);
    }
    else {
        count = output_count;
        output_count += number_on_local_list + 1;
        pthread_mutex_unlock(&output_count_lock);
        print_records(record_ptr, local_list,
            requested_number_of_records - count);
        return(count + number_on_local_list + 1);
    }
}
```

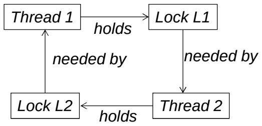
Problems with locks

- Locks are most dangerous when a thread needs to acquire multiple locks before releasing locks
- Two main problems:
  - deadlock
  - livelock
- Deadlock:
  - Threads A and B need locks L1 and l2
  - Thread A acquires L1 and wants L2
  - Thread B acquires L2 and wants L1
  - In general, there will be a cycle of threads in which each thread holds some locks and is waiting for locks held by other threads in the cycle
- Livelock:
  - may arise in some solutions to deadlock

Deadlock

- Code snippet shows example of possible deadlock
- Subtle point:
  - deadlock may happen in some executions and not in others!
- "Deadly embrace": Dijkstra
- How do we ensure deadlocks cannot occur?

```
Thread 1:        Thread 2:
…                …
lock(L1);        lock(L2);
lock(L2);        lock(L1);
….               …
```



Deadlock: four conditions

- Mutual exclusion:
  - thread has exclusive control over resource it acquires
- Hold-and-wait:
  - thread does not release resource it holds if it is waiting for another resource
- No pre-emption:
  - No external agency forces a thread to release resources if thread is waiting for another resource
- Circular wait:
  - There is a cycle of threads such that each thread holds one or more resources needed by the next thread in the cycle

You prevent deadlocks by ensuring that one or more of these conditions cannot arise in your program.

Prevent circular wait

- Assign a logical total order to locks
  - (eg) name them L1,L2,L3,…
- Ensure that threads will never try to acquire a lower numbered lock while holding a higher numbered lock
  - (eg) if thread owns L3, it can try to acquire L4, L5, L6,… but it cannot try to acquire locks L1 or L2 (unless it already owns them and locks are re-entrant)
- Useful software engineering principle when you have control over the entire code base and you know what locks are required where
- However
  - easy to make mistakes
  - tension with encapsulation:
    - requires detailed knowledge of entire code base

Prevent hold-and-wait

- Try to acquire all locks atomically
- One implementation:
  - single global lock to get permission to acquire locks you need
- Problem:
  - not scalable
  - conflicts with modularity and encapsulation
- You might encounter a hidden version of this problem if thread has to enter the kernel to perform some function like storage allocation
  - kernel lock is like the global-lock in our example

```
…
lock(global-lock);
lock(l1);
lock(l2);
unlock(global-lock);
…
```

Self-preemption

- Coding discipline:
  - Use only try-locks
  - If a thread cannot acquire a lock while it is holding other locks, it releases all locks it holds and tries again
  - Variation: OS or some other agency steps in and preempts a thread
- Problems:
  - Encapsulation
  - Livelock: threads can keep on acquiring and releasing locks without making progress because no thread ever gets all the locks it needs
  - One solution to livelock: (Ethernet) backoff: thread does not retry until some randomly chosen amount of time has passed

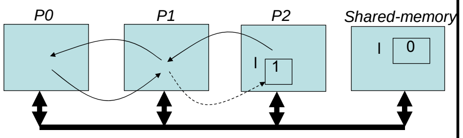
```
loop:
//start of lock acquires
….
if (trylock(Lj) == EBUSY) {
//unlock all locks you hold
goto loop;
}
….
endloop:

//compute with resources
//release locks
```

Implementing locks using atomic instruction

- Atomic swap(addr,reg)
  - swap contents of address and register atomically
- Spin-lock using swap
  - location lock has 0/1 for unlocked/locked
  - lock code:
    - load 1 into register rx;
    - swap(lock,rx);
    - test rx:
      - if rx is 1, you don't have lock so try again
      - if rx is 0, you have lock and no one else can have it till you unlock
  - unlock
    - store 0 into lock;
- Problem:
  - swap must invalidate line in all caches even when lock acquire is not successful
  - if there are a lot of threads waiting for lock, busy-waiting will create a lot of bus traffic

Busy-waiting and bus traffic

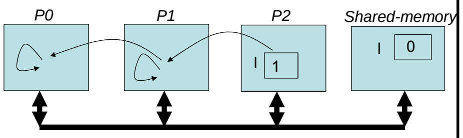
```
…..
     mov edx,1
acq: swap [l], edx
     test edx, edx
     jnz acq
…..
```



- Busy-waiting creates a lot of bus traffic
- Sequence of actions
  - all threads do exchg
  - P2 wins and gets lock
  - P0 and P1 keep doing swap operations, invalidating line in other caches
  - P2 releases lock by writing 0 to lock
  - ….
- Solution: test-and-test-and-set
  - keep doing ordinary reads until lock is 0
  - then go into acq loop and see if you can get lock
  - if you fail, jump back to read loop

Better spin-locks

```
…..
      mov edx,1
spin: mov eax, [l]
      test eax, eax
      jnz spin
      swap [l], edx
      test edx, edx
      jnz spin
…..
```



- Inner spin loop does not create bus traffic since all spinning threads spin on their local caches
- When P2 unlocks, line is invalidated from P0 and P1

Compare-and-swap (CAS)

- Another atomic instruction: compare-and-swap (cas)
  - cas addr, old-value, new-value
  - check if addr contains old-value
  - if so, update it to new-value and return SUCCESS; otherwise return FAIL.
- Consider Bellman-Ford
  - relaxation of edge (u,v)

```
tN = dist(u) + w(u,v);
acquire lock on v; //this uses swap
if (tN < dist(v)) dist(v) = tN;
release lock on v;
```

- Bellman-Ford (II):
  - Relaxation of edge (u,v)

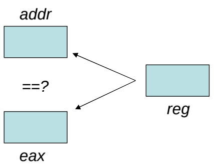
```
repeat {
  tO = dist(v); //read old value
  tN = dist(u)+w(u,v); //compute new value
  if (tN < tO)
    done = cas(dist(v),tO,tN); //write if dist(v) still contains tO
  else
    done = SUCCESS;
until (done==SUCCESS)}
```

- Advantages:
  - Separate locations for locks not needed
  - Smaller critical section
  - Fewer writes
- CAS operation was first introduced in IBM System 370

CAS in x86

- x86 is a 2-address ISA but CAS requires three operands
  - Solution: eax is implicit operand
- cmpxchg addr, reg;

```
if (contents(addr) == eax))
   {addr = reg;
    zero-flag = 1;
   }
  else
   {eax = reg;
    zero-flag = 0;
   }
```



- "lock cmpxchg addr,reg"
  - instruction is executed atomically

Spinlock example in x86

```
global main

extern printf
extern pthread_create
extern pthread_exit
extern pthread_join

section .data
        align 4
        sLock:              dd 0        ; The lock, values are:
                                        ; 0         unlocked
                                        ; 1         locked
        tID1:               dd 0
        tID2:               dd 0
        fmtStr1:     db "In thread %d with ID: %02x", 0x0A, 0
        fmtStr2:     db "Result %d", 0x0A, 0

section .bss
        align 4
        result:             resd 1
```

```
section .text
        main:                                   ; Using main since we are using gcc to link

                                                ; Call pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                                                ;                   void *(*start_routine) (void *), void *arg);

        push        dword 0                     ; Arg Four: argument pointer
        push        thread1                     ; Arg Three: Address of routine
        push        dword 0                     ; Arg Two: Attributes
        push        tID1                        ; Arg One: pointer to the thread ID
        call        pthread_create

        push        dword 0                     ; Arg Four: argument pointer
        push        thread2                     ; Arg Three: Address of routine
        push        dword 0                     ; Arg Two: Attributes
        push        tID2                        ; Arg One: pointer to the thread ID
        call        pthread_create

                                                ; Call int pthread_join(pthread_t thread, void **retval) ;
                                                ;
        push        dword 0                     ; Arg Two: retval
        push        dword [tID1]  ; Arg One: Thread ID to wait on
        call        pthread_join
        push        dword 0                     ; Arg Two: retval
        push        dword [tID2]  ; Arg One: Thread ID to wait on
        call        pthread_join

        push        dword [result]
        push        dword fmtStr2
        call        printf
        add         esp, 8                      ; Pop stack 2 times 4 bytes

        call exit
```

```
thread1:
        pause
        push    dword [tID1]
        push    dword 1
        push    dword fmtStr1
        call    printf
        add     esp, 12             ; Pop stack 3 times 4 bytes

        call    spinLock

        mov     [result], dword 1
        call    spinUnlock

        push    dword 0             ; Arg one: retval
        call    pthread_exit

thread2:
        pause
        push    dword [tID2]
        push    dword 2
        push    dword fmtStr1
        call    printf
        add     esp, 12             ; Pop stack 3 times 4 bytes

        call    spinLock

        mov     [result], dword 2
        call    spinUnlock

        push    dword 0             ; Arg one: retval
        call    pthread_exit
```

```
spinLock:
        push    ebp
        mov     ebp, esp
        mov     edx, 1                          ; Value to set sLock to
spin:   mov     eax, [sLock]  ; Check sLock
        test    eax, eax      ; If it was zero, maybe we have the lock
        jnz     spin                            ; If not try again
        ;
        ; Attempt atomic compare and exchange:
        ; if (sLock == eax):
        ;       sLock         <- edx
        ;       zero flag     <- 1
        ; else:
        ;       eax           <- edx
        ;       zero flag     <- 0
        ;
        ; If sLock is still zero then it will have the same value as eax and
        ; sLock will be set to edx which is one and therefore we aquire the
        ; lock. If the lock was acquire between the first test and the
        ; cmpxchg then eax will not be zero and we will spin again.
        ;
        lock    cmpxchg [sLock], edx    ;eax is implicit operand
        test    eax, eax
        jnz     spin
        pop     ebp
        ret
spinUnlock:
        push    ebp
        mov     ebp, esp
        mov     eax, 0
        xchg    eax, [sLock]
        pop     ebp
        ret
```

```
exit:
                                        ;
                                        ; Call exit(3) syscall
                                        ;void exit(int status)
                                        ;
        mov     ebx, 0                  ; Arg one: the status
        mov     eax, 1                  ; Syscall number:
        int     0x80
```

Barriers

- Pthreads barrier type
  - pthread_barrier_t   varBarrier;
  - basically a struct
    - int total: initialized to # of threads to wait for
    - int count: tracks how many threads have reached barrier
    - mutex
- Initialize barrier
  - int pthread_barrier_init (&varBarrier,NULL,total);
- Waiting at barrier
  - int pthread_barrier_wait (&varBarrier);

Implementation of barriers

- Implemented using an atomic counter
  - Initialized to number of threads that need to arrive at barrier
  - Thread that arrives at barrier
    - decrements counter atomically
    - checks if it is the last one to arrive at barrier (counter = 0) and if so, informs other waiting threads that they can move past barrier
  - Small subtlety when barrier is within a loop

Controlling Thread and Synchronization Attributes

- The Pthreads API allows a programmer to change the default attributes of entities using *attributes objects*.
- An attributes object is a data-structure that describes entity (thread, mutex, condition variable) properties.
- Once these properties are set, the attributes object can be passed to the method initializing the entity.
- Enhances modularity, readability, and ease of modification.

Attributes Objects for Threads

- Use `pthread_attr_init` to create an attributes object.
- Individual properties associated with the attributes object can be changed using the following functions:
  `pthread_attr_setdetachstate`,
  `pthread_attr_setguardsize_np`,
  `pthread_attr_setstacksize`,
  `pthread_attr_setinheritsched`,
  `pthread_attr_setschedpolicy`, and
  `pthread_attr_setschedparam`

Attributes Objects for Mutexes

- Initialize the attrributes object using function: `pthread_mutexattr_init`.
- The function `pthread_mutexattr_settype_np` can be used for setting the type of mutex specified by the mutex attributes object.

```
pthread_mutexattr_settype_np (
pthread_mutexattr_t *attr,
int type);
```

- Here, `type` specifies the type of the mutex and can take one of:
  - `PTHREAD_MUTEX_NORMAL_NP`
  - `PTHREAD_MUTEX_RECURSIVE_NP`
  - `PTHREAD_MUTEX_ERRORCHECK_NP`