GPU Programming

Dr. Farshad Khunjush

Department of Computer Science and Engineering
Shiraz University
Fall 2024

Concurrency & Critical-Sections

Farshad Khunjush

Some slides come from
Professor Henri Casanova
@ http://navet.ics.hawaii.edu/~casanova/
& Professor Saman Amarasinghe (MIT)
@ http://groups.csail.mit.edu/cag/ps3/

Example

- Consider two threads that both increment a variable x
  - Thread #1: ...; increment x; ...
  - Thread #2: ...; increment x; ...
- If you think of this in some low-level code, like assembly or byte code, the codes of the two threads are:

Thread #1	Thread #2
...	...
Load x into Register R	Load x into Register S
R = R + 1	S = S + 1
Store R into x	Store S into x
...	...

Example (Cont'd)

- Problem: Threads can be context-switched at will by the O/S / JVM
- In principle: one can have an arbitrary interleaving of instructions
- Example:

Thread #1

```
...
Load x into
   Register R
R = R + 1
Store R into x
...
```

Interleaving

```
...
Load x into Register R
Load x into Register S
S = S + 1
Store S into x
R = R + 1
Store R into x
...
```

Thread #2

```
...
Load x into Register
   S
S = S + 1
Store S into x
...
```

Resulting computation: x +=1 as opposed to x +=2!

Likely Interleaving?

- The error in the previous slide is called “lost update”
- On a single-proc/single-core computer, with false concurrency, the odds that bad interleaving happens could be low
- On a multi-proc/multi-core system, i.e., when we have true concurrency, bad interleaving is much more likely

Race Condition

- The behavior of our example is non-deterministic
  - The end value of variable x could be either 1 or 2
- There is no way to know in advance what the result will be as it depends on
  - The architecture
  - The O/S, JVM
  - The load and state of the computer
- This lost update problem is an example of a race condition
  - The final result depends on the interleaving of the threads' instructions
  - Threads are "racing" to "get there first" and one cannot tell in advance which thread will win

Atomicity and mutual exclusion

- What we need is a mechanism that makes the updating of shared variable x atomic
  - Atomic: Whenever the update is initiated, we are guaranteed that it will go uninterrupted/undisturbed by other updates
- One can implement atomic updates to variable x by enforcing mutual exclusion
  - If one thread is updating variable x then **NO** other thread can initiate an update of variable x
- This is a great idea, but how can we specify this in a program? by critical sections
- A critical section is a section of code in which only one thread is allowed at a time
- This is the most common and simplest form of synchronization for multi-threaded programs

Critical Sections (CS)

- One would like to write code that looks like this:

```
while(true) {
  enter_CS
  x++
  leave_CS
}
```

- We would like to have the following properties
  - Mutual exclusion: only one thread can be inside the CS
  - No deadlocks: one of the competing threads enters the CS
  - No unnecessary delays: a thread enters the CS immediately if no other thread is competing for it
  - Eventual entry: a thread that tries to enter the critical CS will enter it at some point
- We will see that these come from:
  - the way the CS is implemented by the language+system
  - the way in which one writes concurrent applications

Critical Sections with Locks

- The concept of a critical section is binary
  - Either 0 thread is in the critical section
  - Or 1 thread is in the critical section
- Therefore, the critical section can be "controlled" with a boolean variable
- This variable is called a **lock**

```
while(true) {
    try to acquire lock  // wait if can't and keep trying
                X++
      release lock
}
```

- Just like going to a washroom in an airplane
  - While the lock is "red" wait
  - Then go in and set the lock to "red"
  - Then set the lock to "green" and leave

Locks

- Different languages have different ways to declare/use locks
- Let's see the use of locks on several examples using a C-like syntax
  - Declaration: lock_t lock
  - Locking: lock(&lock)
  - Unlocking: unlock(&lock)

Locks for Data Structures

- A classical use of locks is to protect updates of linked data structures
- Example: Queue and threads
  - Consider a program that maintains a queue (of ints >0)
  - Thread #1 (Producer) adds elements to the queue
  - Thread #2 (Consumer) removes elements from the queue

Thread #1 (Producer)

```
int x;
while(1) {
  x = generate();
  insert(list,x);
}
```

Thread #2 (Consumer)

```
int x;
while(1) {
  x = remove(list);
  process(x);
}
```

Queue Implementation

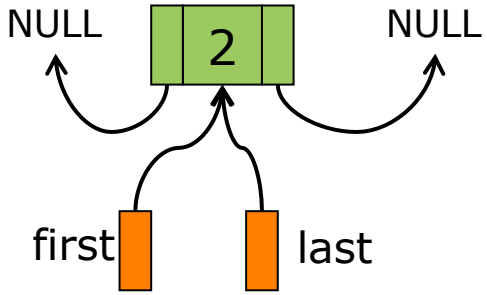
```
void insert (queue_t q, int x) {
    queue_item_t *item = (queue_item_t)
    calloc(1,sizeof(queue_item_t));
    item->value = x;
    item->next = q->first;
    if (item->next)
      item->next->prev = item;
    q->first = item;
    if (! q->last)  q->last = item;
}
```

Queue Implementation (Cont'd)

```
int remove (queue_t q) {
    queue_item_t *item;
    int x;

    if (! q->last) return -1;
    x = q->last->value;
    item = q->last->prev;
    free(q->last);
    if (item) {
      item->next = NULL;
    }
    q->last = item;
    if (q->last == NULL) {
      q->first = NULL;
    }
    return x;
}
```

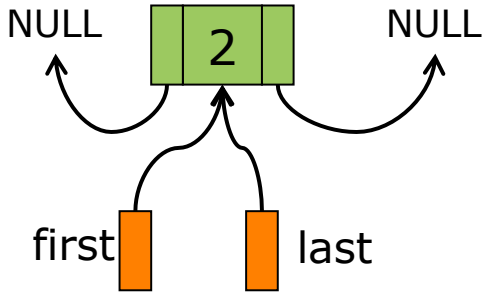
What bad thing could happen?

- Consider the following linked list



What bad thing could happen?
(Cont'd)

- Consider the following linked list

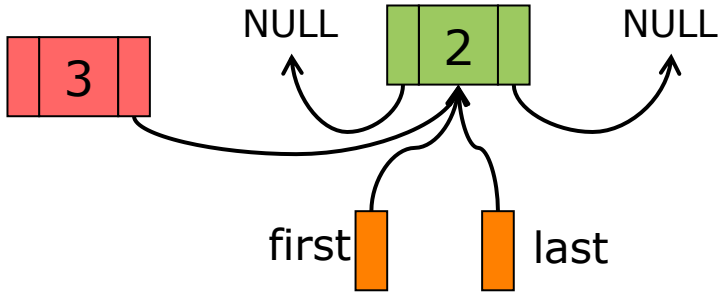


- The Producer calls insert(3)

```
queue_item_t *item = calloc(...)
item->value = x;
item->next = q->first;
if (item->next)
    item->next->prev = item;
q->first = item;
if (! q->last)
    q->last = item;
```

What bad thing could happen?
(Cont'd)

- Consider the following linked list



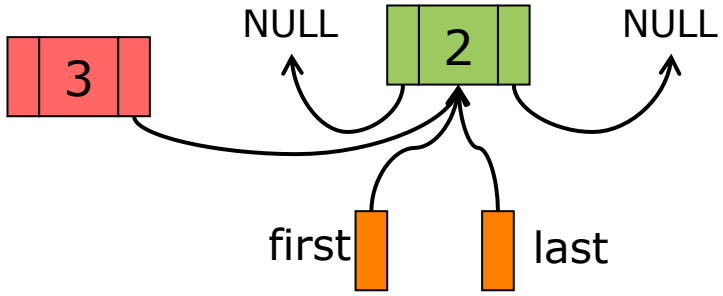
- The Producer calls insert(3)

```
queue_item_t *item = calloc(...)
item->value = x;
item->next = q->first;
if (item->next)
    item->next->prev = item;
q->first = item;
if (! q->last)
    q->last = item;
```

context switch

What bad thing could happen?
(Cont'd)

- Consider the following linked list



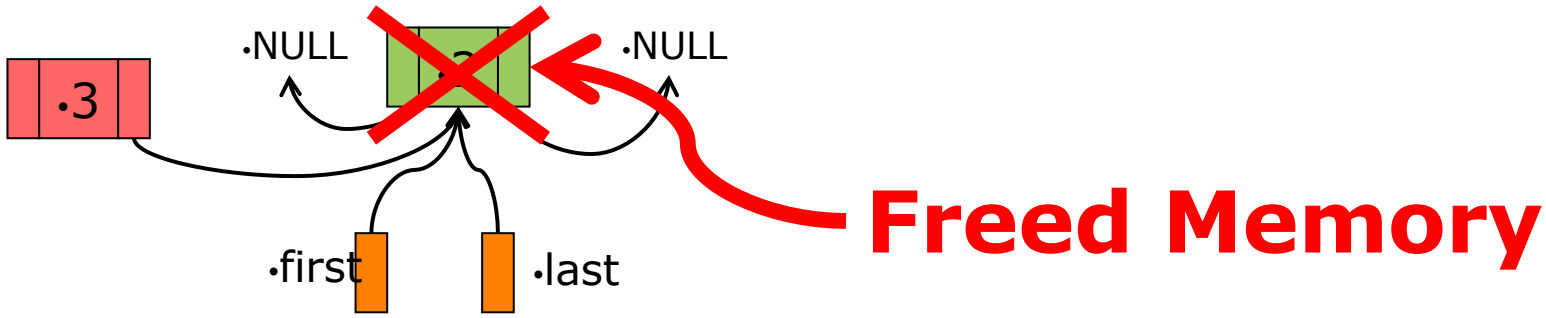
- The Consumer calls remove

```
...
item = q->last->prev;   // returns NULL
free(q->last);
if (item) {
...
```

context switch

What bad thing could happen?
(Cont'd)

- Consider the following linked list



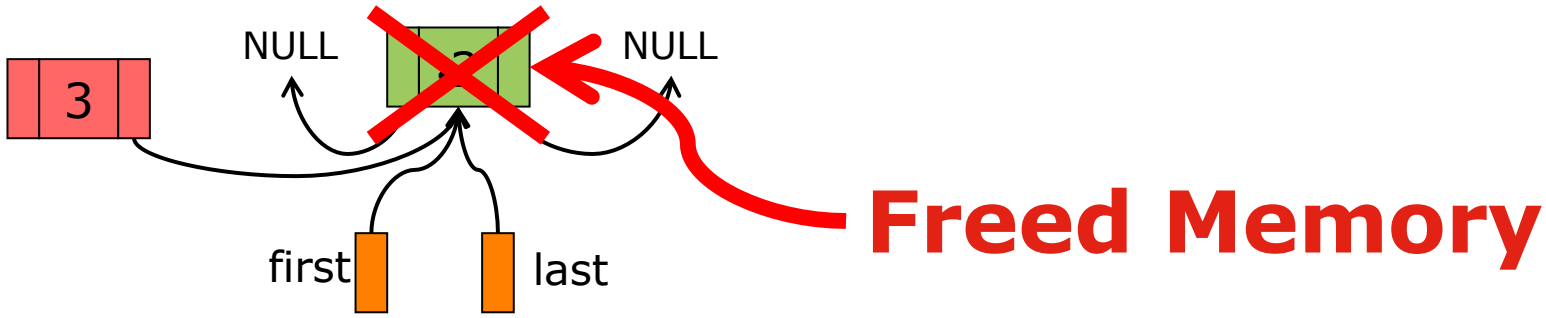
- The Consumer calls remove

```
...
item = q->last->prev;   // returns NULL
free(q->last);
if (item) {
. . .
```

•context
•switch

What bad thing could happen?

- Consider the following linked list



- The Producer resumes

```
queue_item_t *item = calloc(...)
item->value = x;
item->next = q->first;
if (item->next)
    item->next->prev = item;
q->first = item;
if (! q->last)
    q->last = item;
```

Freed Memory Access

So what?

- In this example, the producer updates memory that has been de-allocated
- In Java we would get an exception once in a while
- C doesn't zero out or track freed memory and we would get a segmentation fault once in a while
  - A third thread could have done a malloc and be given the memory that has been de-allocated
  - Then the producer could modify the memory used by that third thread
  - This could cause a bug in that third thread that could be very difficult to track
  - Basically, if you have threads and you get unexplained segmentation faults, you may have a race condition
    - Even if the segmentation fault occurs in a part of the code that has nothing to do with the relevant part of the code!
- Let's use locks and fix it

Simple Solution

```
lock_t lock;  // global variable
```

```
void producer() {
  int x;
  while(1) {
    x = generate();
    lock(lock);
    insert(list,x);
    unlock(lock);
  }
}
```

```
void consumer() {
  int x;
  while(1) {
    lock(lock);
    x = remove(list);
    unlock(lock);
    process(x);
  }
}
```

Simple Solution

- Important: we use a single lock that is referenced/used by both threads
- The solution is simple: place the lock around all calls that manipulate the queue
  - Sometimes determining what calls and code segments modify a structure requires some thought
- The critical section is then the whole queue implementation
- This is the typical strategy when using a non-thread-safe implementation of the queue abstract data type
- To produce a thread-safe implementation, one needs to create critical sections in the queue implementation

Thread Safe Queue

```
void insert (queue_t q, int x) {
    lock(q.lock);   // each queue has its lock
    queue_item_t *item = (queue_item_t)
    calloc(1,sizeof(queue_item_t));
    item->value = x;
    item->next = q->first;
    if (item->next)
      item->next->prev = item;
    q->first = item;
    if (! q->last)  q->last = item;
    unlock(q.lock);
}
```

Thread-Safe Queue

```
int remove (queue_t q) {
    queue_item_t *item;
    int x;
    lock(q.lock);
    if (! q->last) return -1;
    x = q->last->value;
    item = q->last->prev;
    free(q->last);
    if (item) {  item->next = NULL;  }
    q->last = item;
    if (q->last == NULL) {
      q->first = NULL;
    }
    unlock(q.lock);
    return x;
}
```

Implementing Lock

- At this point we have some idea of how to use lock() and unlock() to create critical sections and ensure safe concurrency
- Question: how does one implement lock???
  - Granted, you will probably not need to as languages/systems provide them
  - But it's interesting to have some idea of how things work
  - And it will be our first attempts at reasoning about concurrency
- There are two kinds of lock implementations
  - software solutions
  - hardware solutions

Software Spin Locks: v0

- A Spin Lock is simply a boolean variable
- unlock()
  - simply set the variable to 0
- lock()
  - check whether the variable is equal to 0
  - if it is equal to 1 check again
  - if it is equal to 0, set it to 1 and enter and continue
- Simple implementation?

```
void lock(int *lock) {
  while (*lock) yield();  //spin
  *lock = 1;
}
```

```
void unlock(int *lock) {
  *lock = 0;
}
```

- Note the use of yield(), which is probably better but not mandatory in a system that implements time-slicing
- What's wrong with this implementation?

Software Spin Locks: v0

```
void lock(int *lock) {
  while (*lock) yield(); // spin
  *lock = 1;
}
```

- Code for lock() in assembly pseudo-code

```
spin: LD R1, <lock>
      BNEZ R1, spin
      SDI #1, <lock>
```

Thread #1

```
spin:   LD R1, <lock>
```

Thread #2

Software Spin Locks: v0

```
void lock(int *lock) {
  while (*lock) yield(); // spin
  *lock = 1;
}
```

- Code for lock() in assembly pseudo-code

```
spin: LD R1, <lock>
      BNEZ R1, spin
      SDI #1, <lock>
```

Thread #1

```
spin:   LD R1, <lock>
```

Thread #2

```
spin:   LD R1, <lock>
        BNEZ R1, spin
        SDI #1, <lock>
```

Software Spin Locks: v0

```
void lock(int *lock) {
  while (*lock) yield(); // spin
  *lock = 1;
}
```

- Code for lock() in assembly pseudo-code

```
spin: LD R1, <lock>
      BNEZ R1, spin
      SDI #1, <lock>
```

Thread #1

```
spin:  LD R1, <lock>
       BNEZ R1, spin
       SDI #1, <lock>
```

Thread #2

```
spin:  LD R1, <lock>
       BNEZ R1, spin
       SDI #1, <lock>
```

- Both threads are now in the critical section!!

Software Spin Lock

- There is a race condition in the lock() function on the boolean lock variable itself!
  - Adding another lock on the lock would only push the problem down one level, and so on...
- One possible solution would be to use a “turn-based” system
  - A variable alternates between 0 and 1
  - A value of 0 indicates that Thread #1 should get access to the critical section
  - A value of 1 indicates that Thread #2 should get access to the critical section
  - Initially the value is (arbitrarily) set to 0
- Let’s look at the code

Software Spin Lock: v1 (lock=turn)

```
void lock(int *turn, int me) {
  while (*turn != me) yield();  // spin
}
```

```
void unlock(int *turn, int me)  {
  other = 1 - me;
  *turn = other;
}
```

- Thread #1 calls the functions passing 0 as an argument, and thread #2 calls the functions passing 1 as an argument
- The code above solves the problem of the previous implementation
  - the two threads cannot enter the critical section because only a single thread can have turn to enter the CS
- What is the problem?

Software Spin Lock: v1

```
void lock(int *turn, int me) {
  while (*turn != me) yield();  // spin
}
```

```
void unlock(int *turn, int me)  {
  other = 1 - me;
  *turn = other;}
```

- The problem is starvation
- Consider the following sequence of locks and unlocks:

  Thread #1: lock(0);

  Thread #1: unlock(0);

  Thread #1: lock(0); // blocks!
- Thread #1 is blocked until Thread #2 goes into the critical section
- Threads have to alternate in the critical section
  - Because it's turn-based
- This goes against the principle of "no unnecessary delays"

Software Spin Lock: v2

```
void lock(lock_t lock, int me) {
  other = 1 – me;
  while (lock.occupied[other] == true) yield();
  lock.occupied[me] = true;
}
```

```
void unlock(lock_t lock, int me) {
  lock.occupied[me] = false;
}
```

- The idea here is to use two variables inside the lock:

```
typedef struct {
  boolean occupied[2];
} lock_t;
```

  - Initialize at {false, false}
- This way, we avoid the alternating problem of the previous implementation
- Is it correct?

Software Spin Lock: v2

```
void lock(lock_t lock, int me) {
  other = 1 –me;
  while (lock.occupied[other] == true)
          yield();
  lock.occupied[me] = true;}
```

```
void unlock(lock_t lock, int me) {
  lock.occupied[me] = false;
}
```

- The idea here is to use two variables inside the lock:

```
typedef struct {
  boolean occupied[2];
} lock_t;
```

  - Initialize at {false, false}
- This way, we avoid the alternating problem of the previous implementation
- Is it correct?
- Nope:
  - The two threads enter lock() “at the same time”
  - They both see the other’s flag set to false and proceed
  - We now have two threads in the critical section!

Software Spin Lock: v3

```
void lock(lock_t lock, int me) {
  other = 1 - me;
  lock.occupied[me] = true;
  while (lock.occupied[other] == true)
          yield();
}
```

```
void unlock(lock_t lock, int me) {
  lock.occupied[me] = false;
}
```

- To avoid the problem from before we swapped the two statements in function lock()
  - There is no interleaving of the executions that can lead to both threads entering the critical section simultaneously

```
lock.occupied[0] = true;                    lock.occupied[1] = true;
while(lock.occupied[1] == true) yield();    while(lock.occupied[0] == true) yield();
```

- But now we have a new problem

Software Spin Lock: v3

```
void lock(lock_t lock, int me) {
  other = 1 - me;
  lock.occupied[me] = true;
  while (lock.occupied[other] == true) yield();
}
```

```
void unlock(lock_t lock, int me) {
  lock.occupied[me] = false;
}
```

- To avoid the problem from before we swapped the two statements in function lock()
  - There is no interleaving of the executions that can lead to both threads entering the critical section simultaneously

```
lock.occupied[0] = true;                    lock.occupied[1] = true;
while(lock.occupied[1] == true) yield();    while(lock.occupied[0] == true) yield();
```

- But now we have a new problem: deadlock!
  - Both threads set their variable to true
  - Then they both spin forever
- Again, unlikely but possible, esp. with true concurrency

Software Spin Lock: v4

```
void lock(lock_t lock, int me) {
  other = 1 - me;
  lock.occupied[me] = true;
  while (lock.occupied[other] == true) {
    lock.occupied[me] = false;
    yield();
    lock.occupied[me] = true;
  } }
```

```
void unlock(lock_t lock, int me) {
  lock.occupied[me] = false;
}
```

- The idea here is to fix the problem from v3 by having threads back off when they realize they're both entering the function at the same time
  - If the other's flag is set to true, I set mine to false, let the other run for a while, and set mine to true again and check on the other's flag
- There is STILL a problem here!

Software Spin Lock: v4

```
void lock(lock_t lock, int me) {
  other = 1 - me;
  lock.occupied[me] = true;
  while (lock.occupied[other] == true) {
    lock.occupied[me] = false;
    yield();
   lock.occupies[me] = true;
 }}
```

```
void unlock(lock_t lock, int me) {
  lock.occupied[me] = false;
}
```

- The problem is livelock!
  - A kind of deadlock in which threads are in an infinite (or very long) sequence of blocking and unblocking
- Threads could be in locked step
  - They both set their flags to true
  - They both set their flags to false
  - They both set their flags to true
  - . . .
- With false concurrency, this is virtually impossible
- With true concurrency, the livelock could last a long time

Software Spin Lock: v5

```
void lock(lock_t lock, int me) {
  other = 1 – me;
  lock.occupied[me] = true;
  while (lock.occupied[other] == true) {
    if (lock.turn != me) {
       lock.occupied[me] = false;
       while (lock.turn != me) yield();
       lock.occupied[me] = true;
    }}}
```

```
void unlock(lock_t lock, int me) {
  lock.occupied[me] = false;
  lock.turn = other;
}
```

- We add a "turn" variable to the lock structure

```
typedef struct {
  boolean occupied[2];
  int turn;
}
```

- The threads take turns backing off!
- This is a very good solution [Dekker, 1960's]

Software Spin Lock: v6

- In 1981 Peterson came up with a complete *and simpler* solution:

```
typedef struct {
  boolean occupied[2];
  int last;
} lock_t;
```

- The last field tracks which thread last tried to enter the CS
- This is the thread that is delayed if both threads compete

```
void lock(lock_t lock, int me) {
  other = 1 - me;
  lock.occupied[me] = true;
  lock.last = me;
  while (lock.occupied[other] == true
                    && lock.last == me)
    yield();
}
```

```
void unlock(lock_t lock, int me) {
  lock.occupied[me] = false;
}
```

Exponential Back off

- One problem with spin locks is that they consume CPU cycles
  - A thread is in an infinite loop trying to acquire the lock
- A fix (a “hack” really) is to have threads back off for exponentially increasing (but bounded) periods of time
  - Reduces responsiveness
  - But saves CPU cycles which may benefit the thread in the critical section
- No matter what, it’s always a good idea to have very short critical sections so that threads spend very little time in lock()

Software Locks: Conclusion

- It turns out that having a good solution required some thought
- Thanks to Peterson we have one
  - Note that formally proving that it is a correct solution is not easy
  - Just know that detecting race conditions, deadlocks and starvations by looking at the code is NP-hard
- But what about more than two threads?
- Turns out things get much more complicated
- Example: the bakery algorithm (by Lamport)
  - Analogous to a bakery with a machine dispensing tickets to customers
  - Cleverly designed to avoid all the problems we have seen with v1, v2, v3 and v4
  - Accommodates an arbitrary number of threads

Hardware Solutions

- The software solutions are interesting
  - Especially because the same principles and reasoning applies when writing concurrent applications that use locks
- But they could be time/memory consuming
- As usual, a hardware solution can solve many of the problems of a software solution in a way that's simpler and faster
  - At least simple for software developers
- There are two possible options
  - Disabling interrupts
  - Atomic instructions

Disabling Interrupts

- This is extremely simple: to avoid badly timed context switches, just disable context switches!
- Context switches are implemented via interrupts
- All systems have instructions to disable and enable interrupts
  - But typically these instructions are reserved for the O/S running in kernel mode
- It could be very dangerous
  - e.g., disable interrupts and go into an infinite loop in the critical section
- Therefore, this is not really an acceptable solution
- Furthermore, it wouldn't work on a multi-CPU or multi-core architecture
  - Interrupts are local to a processor
- So although attractive because of simplicity it can only be used in the code of the O/S

Atomic instructions

- Let's look at our first naive implementation

```
void lock(int *lock) {
  while (*lock) yield(); // spin
  *lock = 0;
}
```

- The assembly was

```
spin:  LD R1, <lock>
       BNEZ R1, spin
       SDI #0, <lock>
```

- Therefore, between the *loading*, the *testing* and the *setting* the value may have changed
- If we had an atomic "test and set" instruction, we could be sure that the if is done correctly

Test&Set Instruction

- Most processors provide atomic instructions that do multiple things at once
- Example: T&S R1, 0(R2)
  - Equivalent to:
    - Load memory cell 0(R2) into R1
    - If R1 is 0 (FALSE), store 1 (TRUE) into memory cell 0(R2)
  - Can be implemented by locking the memory bus so that no other memory access can occur in between the load, test, and store
- One can then write the assembly for lock():

```
Lock: T&S  R1, <lock>
      BNZ  R1, Lock
      RET
```

Conclusion

- Race conditions are common bugs in concurrent programs
  - non-deterministic and thus difficult to detect
- To prevent race conditions one needs locks
  - can be implemented in software or in hardware
- New issues arise
  - Deadlocks
  - Livelocks
  - Starvation
- We discussed these issues in the context of implementing locks themselves

Pthreads: POSIX Threads

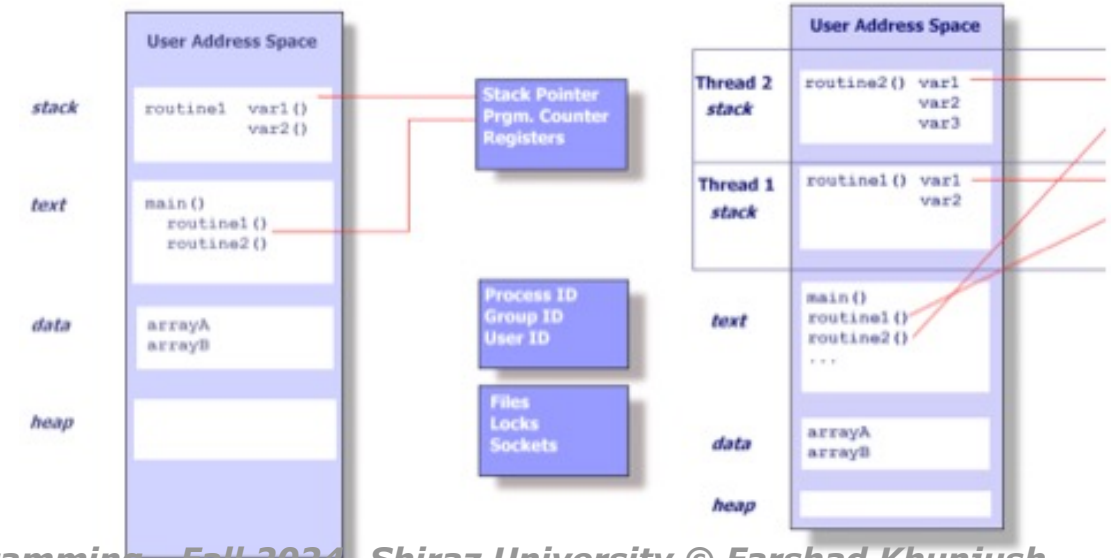
- Pthreads is a standard set of C library functions for multithreaded programming
  - IEEE Portable Operating System Interface, POSIX, section 1003.1 standard, 1995
- Pthread Library (60+ functions)
  - Thread management: create, exit, detach, join, . . .
  - Thread cancellation
  - Mutex locks: init, destroy, lock, unlock, . . .
  - Condition variables: init, destroy, wait, timed wait, . . .
  - . . .
- Programs must include the file **`pthread.h`**
- Programs must be linked with the pthread library (**`-lpthread`**)

Processes & Threads

- A process is created by the operating system.
- Processes contain information about program resources and program execution state, including:
  - Process ID, process group ID, user ID, and group ID
  - Environment
  - Working directory.
  - Program instructions
  - Registers
  - Stack
  - Heap
  - File descriptors
  - Signal actions
  - Shared libraries
  - Inter-process communication tools (such as message queues, pipes, semaphores, or shared memory).

Processes & Threads

- A thread is a light-weight process
  - A thread has a program counter, a stack, a set of registers, and a set of pending and blocked signals
  - All threads in the same process share the virtual address space and the resources



UNIX PROCESS THREADS WITHIN A UNIX

Pthreads: POSIX Threads

- Pthreads is a standard set of C library functions for multithreaded programming
- IEEE Portable Operating System Interface, POSIX, section 1003.1 standard, 1995
- Pthread Library (60+ functions)
  - Thread management: create, exit, detach, join, . . .
  - Thread cancellation
  - Mutex locks: init, destroy, lock, unlock, . . .
  - Condition variables: init, destroy, wait, timed wait, . . .
  - . . .
- Programs must include the file **`pthread.h`**
- Programs must be linked with the pthread library (**`-lpthread`**)

Why pthread?

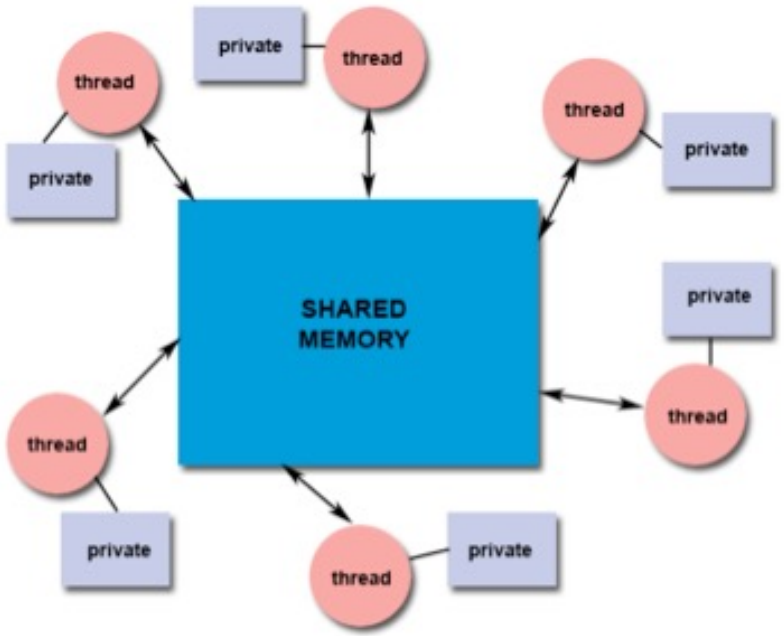
- The primary motivation for using Pthread is to realize potential program performance gains
  - Overlapping CPU work with I/O
  - Asynchronous event handling: tasks which service events of indeterminate frequency
- A thread can be created with much less operating system overhead
- All threads within a process share the same address space

When pthreads?

- Programs having the following characteristics may be well suited for pthreads:
  - Work that can be executed, or data that can be operated on, by multiple tasks simultaneously
  - Block for potentially long I/O waits
  - Must respond to asynchronous events
  - Some work is more important than other work (priority interrupts)

Shared Memory Model

- Shared Memory Model:
  - All threads have access to the same global, shared memory
  - Threads also have their own private data
  - Programmers are responsible for synchronizing access (protecting) globally shared data.



Pthreads API

- The subroutines which comprise the Pthreads API can be informally grouped into three major classes:
  - Thread management: The first class of functions works directly on threads - creating, detaching, joining, etc.
  - Mutexes: The second class of functions deals with synchronization, called a "mutex", which is an abbreviation for "mutual exclusion". Mutex functions provide for creating, destroying, locking and unlocking mutexes.
  - Condition variables: The third class of functions addresses communications between threads that share a mutex. They are based upon programmer specified conditions. This class includes functions to create, destroy, wait and signal based upon specified variable values.

Pthreads Naming Convention

- Types: pthread[_object]_t
- Functions: pthread[_object]_action
- Constants/Macros: PTHREAD_PURPOSE
- Examples:
  - pthread_t: the type of a thread
  - pthread_create(): creates a thread
  - pthread_mutex_t: the type of a mutex lock
  - pthread_mutex_lock(): lock a mutex
  - PTHREAD_CREATE_DETACHED

pthread_create()

```
int pthread_create (      pthread_t *thread,
                          pthread_attr_t *attr,
                          void * (*start_routine) (void *),
                          void *arg);
```

- Returns 0 to indicate success, otherwise returns error code
- **`thread`**: output argument for the id of the new thread
- **`attr`**: input argument that specifies the attributes of the thread to be created (NULL = default attributes)
- **`start_routine:`** function to use as the start of the new thread
  - must have prototype: void * foo(void*)
- **`arg:`** argument to pass to the new thread routine
  - If the thread routine requires multiple arguments, they must be passed bundled up in an array or a structure. NULL may be used if no argument is to be passed.

- Question: After a thread has been created, how do you know when it will be scheduled to run by the operating system?

pthread_create() (Hello World!)

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   int tid;
   tid = (int) threadid;
   printf("Hello World! It's me, thread #%d\n", tid);
   pthread_exit(NULL);
}

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

pthread_create() (Hello World!)
(Cont'd)

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   int tid;
   tid = (int)threadid;
   printf("Hello World! It's me, thread #%d!\n", tid);
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc, t;
   for (t=0; t<NUM_THREADS; t++){
      printf("In main: creating thread %d\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *) t); // &t Correct??
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }
   pthread_exit(NULL); // Why??
}
```

pthread_create() example (Cont'd)

- Want to create a thread to compute the sum of the elements of an array

```
void *do_work(void *arg);
```

- Needs three arguments
  - the array, its size, where to store the sum
  - we need to bundle them in a structure

```
struct arguments {
        double *array;
        int size;
        double *sum;
}
```

pthread_create() example (Cont'd)

```
int main(int argc, char *argv) {
  double array[100];
  double sum;
  pthread_t worker_thread;
  struct arguments *arg;

  arg = (struct arguments *)calloc(1,
                                   sizeof(struct arguments));
  arg->array = array;
  arg->size=100;
  arg->sum = &sum;

  if (pthread_create(&worker_thread, NULL,
                     do_work, (void *) arg)) {
    fprintf(stderr,"Error while creating thread\n");
    exit(1);
  }
  …
}
```

pthread_create() example (Cont'd)

```
void *do_work(void *arg) {
 struct arguments *argument;
 int i, size;
 double *array;
 double *sum;

 argument = (struct arguments*) arg;

 size = argument->size;
 array = argument->array;
 sum = argument->sum;

 *sum = 0;
 for (i=0;i<size;i++)
   *sum += array[i];

 return NULL;
}
```

Comments about the example

- The "main thread" continues its normal execution after creating the "child thread"
- IMPORTANT: If the main thread terminates, then all threads are killed!
  - We will see that there is a join() function
- Of course, memory is shared by the parent and the child (the array, the location of the sum)
  - nothing prevents the parent from doing something to it while the child is still executing → may lead to a wrong computation
  - we will see that Pthreads provide locking mechanisms
- The bundling and unbundling of arguments is a bit tedious

Memory Management of Arguements

- The parent thread allocates memory for the arguments
- Warning #1: you don't want to free that memory before the child thread has a chance to read it
  - That would be a race condition
- Warning #2: if you create multiple threads you should to be careful that there is no sharing of arguments, or that the sharing is safe
  - Safest way: have a separate arg structure for each thread

pthread_exit()

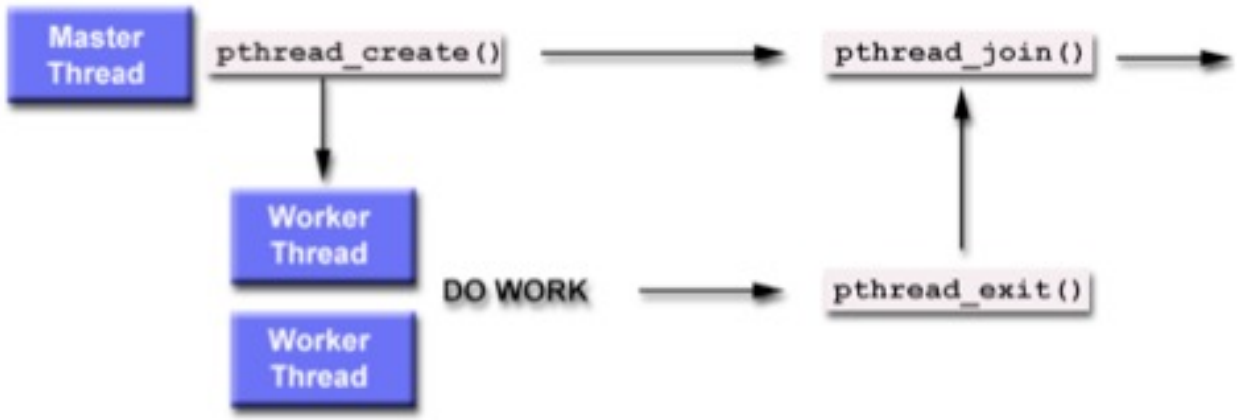
- Terminates the calling thread

```
void pthread_exit(void *retval);
```

  - The return value is made available to another thread calling a **`pthread_join()`** (see next slide)

pthread_join()

- "Joining" is one way to accomplish synchronization between threads. For example:



- The pthread_join() subroutine blocks the calling thread until the specified thread terminates

pthread_join() (Cont'd)

- Causes the calling thread to wait for another thread to terminate

```
int pthread_join(pthread_t thread,
    void **value_ptr);
```

  - **`thread`**: input parameter, id of the thread to wait on
  - **`value_ptr`**: output parameter, value given to **`pthread_exit()`** by the terminating thread (which happens to always be a **`void *`**)
  - returns 0 to indicate success, error code otherwise
  - multiple simultaneous calls for the same thread are not allowed

pthread_join() (Cont'd)

- Thread Attributes
  - One of the parameters to pthread_create() is a thread attribute
  - In all our previous examples we have set it to NULL
  - But it can be very useful and provides a simple way to set options:
    - Initialize an attribute
    - Set its value with some Pthread API call
    - Pass it to Pthread API functions like pthread_create()

Pthread Attributes

- Initialized the thread attribute object to the default values

```
int pthread_attr_init(
        pthread_attr_t *attr);
```

  - Return 0 to indicate success, error code otherwise
  - **attr**: pointer to a thread attribute

pthread_join() (Cont'd)

- Joinable or Not?
  - When a thread is created, one of its attributes defines whether it is joinable or detached. Only threads that are created as joinable can be joined (Not detached).
  - To explicitly create a thread as joinable or detached, the attr argument in the pthread_create() routine is used. The typical 4 step process is:
    - Declare a pthread attribute variable of the pthread_attr_t data type
    - Initialize the attribute variable with pthread_attr_init()
    - Set the attribute detached status with pthread_attr_setdetachstate()
    - When done, free library resources used by the attribute with pthread_attr_destroy()

pthread_join() (Cont'd)

- pthread_attr_setdetachstate()
  - Sets the detach state attribute

```
int pthread_attr_setdetachstate(
                pthread_attr_t *attr,
                int detachstate);
```

  - returns 0 to indicate success, error code otherwise
  - attr: input parameter, thread attribute
  - detachstate: can be either
    - PTHREAD_CREATE_DETACHED
    - PTHREAD_CREATE_JOINABLE (default)

pthread_join() (Cont'd)

- Detached threads have all resources freed when they terminate
- Joinable threads have state information about the thread kept even after they finish
  - To allow for a thread to join a finished thread
- So, if you know that you will not need to join a thread, create it in a detached state so that you save resources

pthread_join() (Cont'd)

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS             3

void *BusyWork(void *null)
{
  int i;
  double result=0.0;
  for (i=0; i<1000000; i++)
  {
    result = result + (double)random();
  }
  printf("result = %e\n",result);
  pthread_exit((void *) 0);
}

int main (int argc, char *argv[])
{
  pthread_t thread[NUM_THREADS];
  pthread_attr_t attr;
  int rc, t;
  void *status;

 /* Initialize and set thread detached attribute */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr,
      PTHREAD_CREATE_JOINABLE);

  for(t=0; t<NUM_THREADS; t++)
  {
    printf("Creating thread %d\n", t);
    rc = pthread_create(&thread[t], &attr,
      BusyWork, NULL);
    if (rc)
    {
      printf("ERROR; return code from
            pthread_create() is %d\n", rc);
      exit(-1);
    }
  }

  /* Free attribute and wait for the other threads */
  pthread_attr_destroy(&attr);

  for(t=0; t<NUM_THREADS; t++)
  {
    rc = pthread_join(thread[t], &status);
    if (rc)
    {
      printf("ERROR; return code from
            pthread_join() is %d\n", rc);
      exit(-1);
    }
    printf("Completed join with thread %d status=
      %ld\n",t, (long)status);}
  pthread_exit(NULL);
}
```

pthread_join() Warning

- This is a common “bug” that first-time pthread programmers encounter
- Without the call to pthread_join() the previous program may end immediately, with the main thread reaching the end of main() and exiting, thus killing all other threads perhaps even before they have had a chance to execute

pthread_kill()

- Causes the termination of a thread

```
int pthread_kill(
                    pthread_t thread,
                    int sig);
```

  - **thread**: input parameter, id of the thread to terminate
  - **sig:** signal number
  - returns 0 to indicate success, error code otherwise

pthread_self()

- Returns the thread identifier for the calling thread
  - At any point in its instruction stream a thread can figure out which thread it is
  - Convenient to be able to write code that says: “If you’re thread 1 do this, otherwise do that”
  - However, the thread identifier is an opaque object (just a pthread_t value)
  - you must use pthread_equal() to test equality

```
pthread_t pthread_self(void);
int pthread_equal(pthread_t id1, pthread_t
  id2);
```

Mutual Exclusion and Pthreads

- Pthreads provide Mutex variables as a primary method of implementing thread synchronization and for protecting shared data when multiple writes occur
- A mutex variable acts like a "lock" protecting access to a shared data resource
- Mutexes can be used to prevent “race conditions”

Mutual Exclusion and Pthreads
(Cont'd)

- Pthreads provide a simple mutual exclusion lock
- Lock creation

```
int pthread_mutex_init(
        pthread_mutex_t *mutex,
        const pthread_mutexattr_t *attr);
```

  - returns 0 on success, an error code otherwise
  - **mutex**: output parameter, lock
  - **attr**: input, lock attributes
    - NULL: default
    - There are functions to set the attribute (look at the man pages if you're interested)

Mutual Exclusion and Pthreads
(Cont'd)

- Locking a lock
  - If the lock is already locked, then the calling thread is blocked
  - If the lock is not locked, then the calling thread acquires it

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

  - returns 0 on success, an error code otherwise
  - **`mutex`**: input parameter, lock

Mutual Exclusion and Pthreads
(Cont'd)

- Just checking
  - Returns instead of locking

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

  - returns 0 on success, EBUSY if the lock is locked, an error code otherwise
  - **`mutex`**: input parameter, lock

Mutual Exclusion and Pthreads
(Cont'd)

- Releasing a lock

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

  - returns 0 on success, an error code otherwise
  - **`mutex`**: input parameter, lock

- Pthreads implement exactly the concept of locks as it was described in the previous lecture notes

Mutual Exclusion and Pthreads
(Cont'd)

- Cleaning up memory
  - Releasing memory for a mutex

```
int pthread_mutex_destroy(
        pthread_mutex_t *mutex);
```

  - Releasing memory for a mutex attribute

```
int pthread_mutexattr_destroy(
        pthread_mutexattr_t *mutex);
```

Lock example

```
#include <pthread.h>
#include <stdio.h>
#include <malloc.h>

/*
The following structure contains the necessary
      information
to allow the function "dotprod" to access its input data and
place its output into the structure.
*/

typedef struct
 {
   double      *a;
   double      *b;
   double     sum;
   int     veclen;
 } DOTDATA;

/* Define globally accessible variables and a mutex */

#define NUMTHRDS 4
#define VECLEN 100
DOTDATA dotstr;
pthread_t callThd[NUMTHRDS];
pthread_mutex_t mutexsum;
```

```
void *dotprod(void *arg)
{
  int i, start, end, offset, len ;
  double mysum, *x, *y;
  offset = (int)arg;

  len = dotstr.veclen;
  start = offset*len;
  end   = start + len;
  x = dotstr.a;
  y = dotstr.b;

  mysum = 0;
  for (i=start; i<end ; i++)
   {
     mysum += (x[i] * y[i]);
   }

 pthread_mutex_lock (&mutexsum);
      dotstr.sum += mysum;
 pthread_mutex_unlock (&mutexsum);

  pthread_exit((void*) 0);
}
```

Lock example (Cont'd)

```
int main (int argc, char *argv[])
{
  int i;
  double *a, *b;
  void *status;
  pthread_attr_t attr;

  /* Assign storage and initialize values */
  a = (double*) malloc
      (NUMTHRDS*VECLEN*sizeof(double));
  b = (double*) malloc
      (NUMTHRDS*VECLEN*sizeof(double));

  for (i=0; i<VECLEN*NUMTHRDS; i++)
   {
    a[i]=1.0;
    b[i]=a[i];
   }
  dotstr.veclen = VECLEN;
  dotstr.a = a;
  dotstr.b = b;
  dotstr.sum=0;

  pthread_mutex_init(&mutexsum, NULL); //default

  /* Create threads to perform the dotproduct  */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr,
      PTHREAD_CREATE_JOINABLE);

  for(i=0; i<NUMTHRDS; i++)
  {
    pthread_create( &callThd[i], &attr, dotprod, (void*) i);
  }
  pthread_attr_destroy(&attr);

       /* Wait on the other threads */
  for(i=0; i<NUMTHRDS; i++)
  {
       pthread_join( callThd[i], &status);
   }

   /* After joining, print out the results and cleanup */
   printf ("Sum =  %f \n", dotstr.sum);
   free (a);
   free (b);
   pthread_mutex_destroy(&mutexsum);
   pthread_exit(NULL);
}
```

Condition Variables

- Pthreads also provide condition variables
- Condition variables are of the type pthread_cond_t
- They are used in conjunction with mutex locks

- Let's look at the API's functions

Condition Variables (Cont'd)

- pthread_cond_init()
  - Creating a condition variable

```
int pthread_cond_init(
pthread_cond_t *cond,
 const pthread_condattr_t *attr);
```

    - returns 0 on success, an error code otherwise
    - **cond**: output parameter, condition
    - **attr**: input parameter, attributes (default = NULL)

Condition Variables (Cont'd)

- pthread_cond_wait()
- Waiting on a condition variable

```
int pthread_cond_wait(
            pthread_cond_t *cond,
                pthread_mutex_t *mutex);
```

  - Returns 0 on success, an error code otherwise
    - **cond**: input parameter, condition
    - **mutex**: input parameter, associated mutex

Condition Variables (Cont'd)

- pthread_cond_signal()
  - Signaling a condition variable

```
int pthread_cond_signal(
        pthread_cond_t *cond;
```

- "Wakes up" one thread out of the possibly many threads waiting for the condition
  - The thread is chosen non-deterministically

Condition Variables Example

(Cont'd)

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

#define NUM_THREADS  3
#define TCOUNT 10
#define COUNT_LIMIT 12

int     count = 0;
int     thread_ids[3] = {0,1,2};
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;

void *inc_count(void *idp)
{
  int j,i;
  double result=0.0;
  int *my_id = idp;

  for (i=0; i<TCOUNT; i++) {
    pthread_mutex_lock(&count_mutex);
    count++;
    if (count == COUNT_LIMIT) {
      pthread_cond_signal(&count_threshold_cv);
      printf("inc_count(): thread %d, count = %d  Threshold
         reached.\n", *my_id, count);
    }
    printf("inc_count(): thread %d, count = %d, unlocking mutex\n",
          *my_id, count);
    pthread_mutex_unlock(&count_mutex);

    /* Do some work so threads can alternate on mutex lock */
    for (j=0; j<1000; j++)
          result = result + (double)random();
  }
  pthread_exit(NULL);
}

void *watch_count(void *idp)
{
  int *my_id = idp;

  printf("Starting watch_count(): thread %d\n", *my_id);

    pthread_mutex_lock(&count_mutex);
    if (count<COUNT_LIMIT) {
        pthread_cond_wait(&count_threshold_cv, &count_mutex);
        printf("watch_count(): thread %d Condition signal
          received.\n", *my_id);
    }
    pthread_mutex_unlock(&count_mutex);
    pthread_exit(NULL);
}
```

Condition Variables Example

(Cont'd)

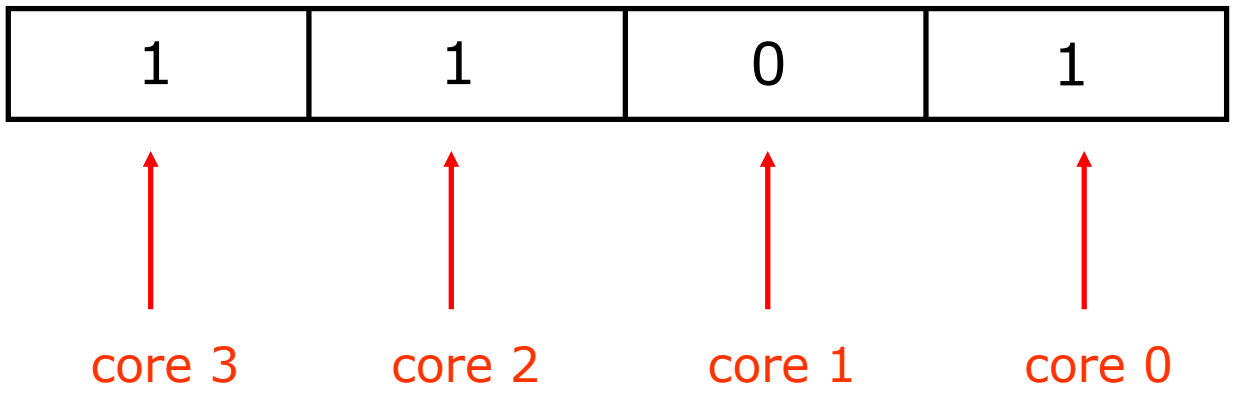
```
int main (int argc, char *argv[])
{
  int i, rc;
  pthread_t threads[3];
  pthread_attr_t attr;

  /* Initialize mutex and condition variable objects
       */
  pthread_mutex_init(&count_mutex, NULL);
  pthread_cond_init (&count_threshold_cv,
       NULL);

  /* For portability, explicitly create threads in a
       joinable state */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr,
       PTHREAD_CREATE_JOINABLE);
  pthread_create(&threads[0], &attr, inc_count,
       (void*) &thread_ids[0]);
  pthread_create(&threads[1], &attr, inc_count,
       (void*) &thread_ids[1]);
  pthread_create(&threads[2], &attr,
       watch_count, (void *)&thread_ids[2]);

  /* Wait for all threads to complete */
    for (i=0; i<NUM_THREADS; i++) {
      pthread_join(threads[i], NULL);
    }
    printf ("Main(): Waited on %d  threads. Done.\n",
         NUM_THREADS);

    /* Clean up and exit */
    pthread_attr_destroy(&attr);
    pthread_mutex_destroy(&count_mutex);
    pthread_cond_destroy(&count_threshold_cv);
    pthread_exit(NULL);

}
```

Assigning threads to the cores

- Each thread/process has an *affinity mask*
- Affinity mask specifies what cores the thread is allowed to run on
- Different threads can have different masks
- Affinities are inherited across fork()

Affinity masks are bit vectors

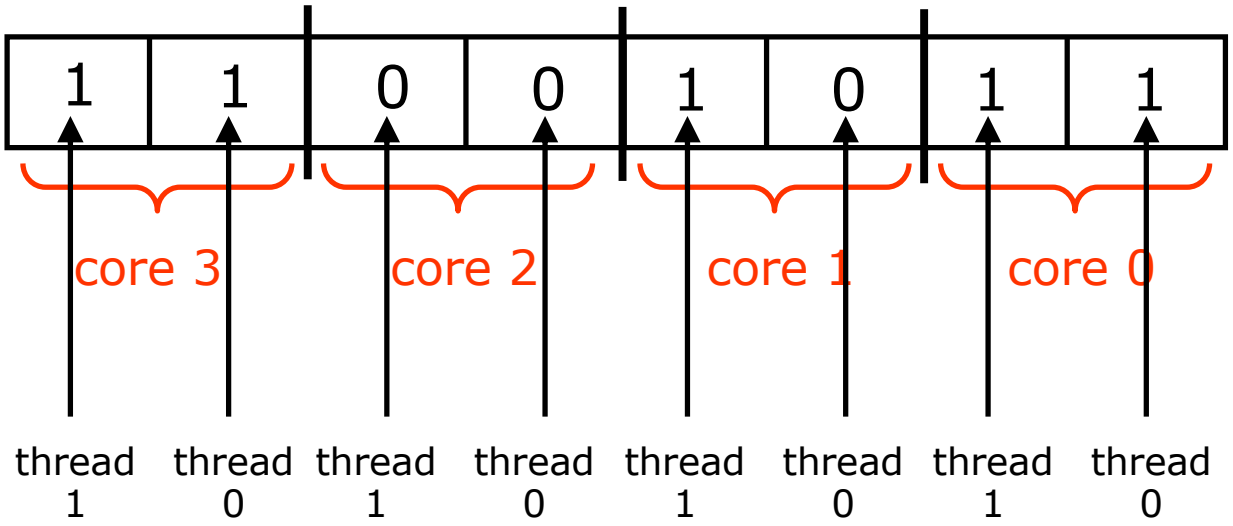
- Example: 4-way multi-core, without SMT



• Process/thread is allowed to run on cores 0,2,3, but not on core 1

Affinity masks when multi-core and SMT combined

- Separate bits for each simultaneous thread
- Example: 4-way multi-core, 2 threads per core



•Core 2 can't run the process

Core 1 can only use one simultaneous thread

Default Affinities

- Default affinity mask is all 1s:
  all threads can run on all processors
- Then, the OS scheduler decides what threads run on what core
- OS scheduler detects skewed workloads, migrating threads to less busy processors

Process migration is costly

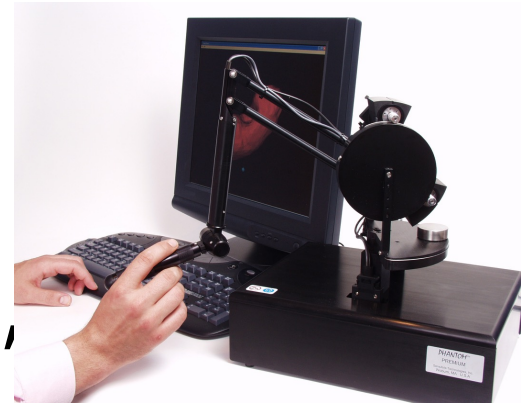
- Need to restart the execution pipeline
- Cached data is invalidated
- OS scheduler tries to avoid migration as much as possible:
  it tends to keeps a thread on the same core
- This is called *soft affinity*

Hard affinities

- The programmer can prescribe her own affinities (hard affinities)
- Rule of thumb: use the default scheduler unless a good reason not to

When to set your own affinities

- Two (or more) threads share data-structures in memory
  - map to same core so that can share cache
- Real-time threads:
  Example: a thread running
  a robot controller:
  - must not be context switched,
    or else robot can go unstable
  - dedicate an entire core just to this thread



•Source: Sensable.cor

Kernel scheduler API

```
#include <sched.h>
int sched_getaffinity(pid_t pid,
  unsigned int len, unsigned long * mask);
```

Retrieves the current affinity mask of process ‘pid’ and stores it into space pointed to by ‘mask’.

‘len’ is the system word size: sizeof(unsigned int long)

Kernel scheduler API

```
#include <sched.h>
int sched_setaffinity(pid_t pid,
    unsigned int len, unsigned long * mask);
```

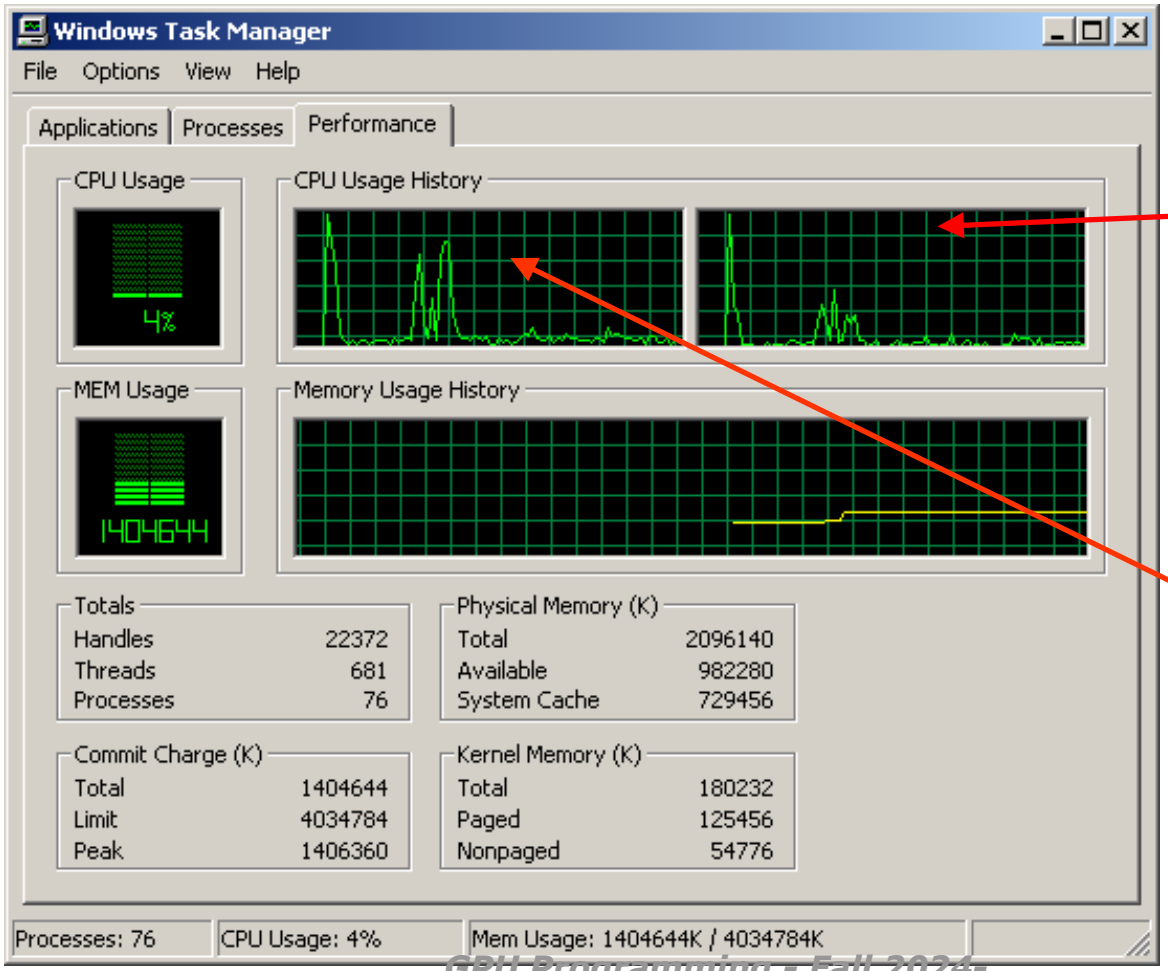
Sets the current affinity mask of process 'pid' to *mask

'len' is the system word size: sizeof(unsigned int long)

To query affinity of a running process:

```
[barbic@bonito ~]$ taskset -p 3935
pid 3935's current affinity mask: f
```

Windows Task Manager



core 2

core 1