

#### Lecture 4 Outline

- Programming for multicore systems
- > Thread execution model
- ➤ A simple concurrent queue
- ➤ At last: Lab 1

### Explicit Parallelism Review

- MIMD: Multiple Instruction, Multiple Data (Flynn's taxonomy)
- Pipelining parallelism on the granularity of instructions, even in sequential code
- Programmers' responsibility: task/thread/process parallelism

## Review: Multicore Programming Tenets

- Keep all the cores busy
  - Break down a problem in smaller tasks
  - Perform tasks as independently as possible or with little synchronization
  - Always try to reduce the number and duration of sequential (non-parallelizable) tasks
- Your code would only go as fast as the duration of your sequential tasks
- Your code would scale with the number of cores to the proportion of its parallelizable tasks (Amdhal's law)

Easier said than done... Let's try it.

## Making Programming Models Concrete

- Programming models should
  - 1. Start/stop tasks
  - 2. Allow tasks to communicate
  - 3. Synchronize tasks
  - 4. Schedule tasks

### Starting/Stopping Tasks

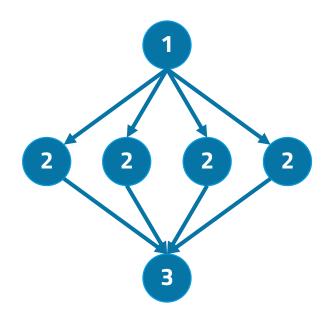
- Language support to identify concurrent tasks
  - Processes
  - Fork/Join
  - Co-routines
  - (Others too numerous to mention)

## Cobegin/Coend

 Marks a portion of where several "threads" of execution are allowed

Example: OpenMP

```
#pragma omp parallel for
for(int i = 0; i < n; i++) {
    c[i] = a[i] + b[i];
}</pre>
```



#### Coroutines

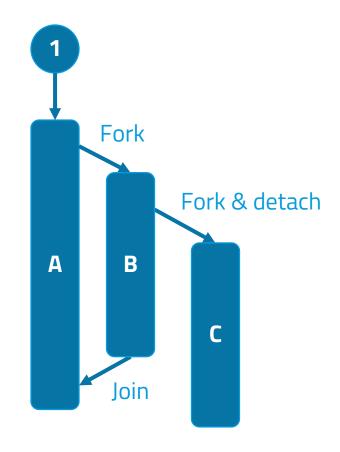
- For concurrency (and possibly parallelism)
- Execute part of a task
  - Control transfer granularity

#### C++ (Boost) Example:

```
#include <boost/coroutine/all.hpp>
using namespace boost::coroutines;
coro(coroutine<void>::yield_type &yield) {
 printf("Exec 1\n");
 yield();
 printf("Exec 3\n");
int main(int argc, char* argv[]) {
  coroutine<void>::yield type routine{coro};
 printf("Exec 2\n");
 routine();
 printf("Exec 4\n");
}
```

### Fork/Join

- Any task can start another task at any point
- Parent and child execute concurrently (and possibly in parallel)
- Parent may or may not wait for child to finish
- Examples
  - Unix processes
  - POSIX threads



## Communication and Synchronization

- Synchronization goals
  - To delay processing until certain conditions hold
  - To guarantee that a block of code behaves as if it were executed atomically
- Message passing
  - Asynchronous communication
- Shared memory
  - Coordinating access...

### Message Passing

- Asynchronous inter-task communication
- Channels: Go example (thanks to Prof. Lerner)

```
func pump(ch chan int)
  for i:=0; ; i++ { ch <- i }
ch1 := make(chan int)
go pump(ch1)
fmt.Println(<- ch1) # prints 0</pre>
func sink(ch chan int)
  for{ fmt.Println(<- ch) }</pre>
go sink(ch1) # prints continuosly
```

### Shared Memory

- Communication via shared state
  - Same effective coordination as message passing, achieved differently
  - Main difference: multiple tasks can manipulate same state
- Sychronization primitives (coming momentarily)

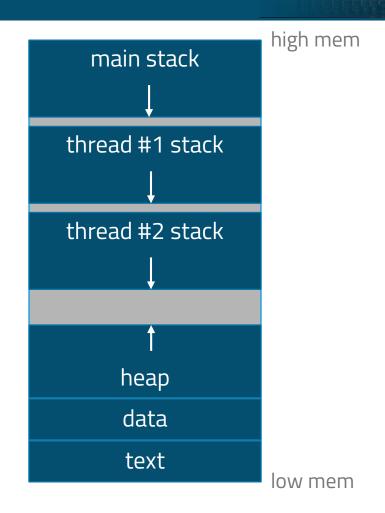


#### Threads

- Fork/join, shared memory model
- No consensus on whether this is optimal abstraction
  - Writing correct threaded programs is hard
  - Current hardware actually "understands" threads
  - Arguably, this model is where other higher level abstractions would be layered
- This has been ongoing research (for a while) and we'll have a chance to explore other models later in the course

## Background: Hardware "Contexts"

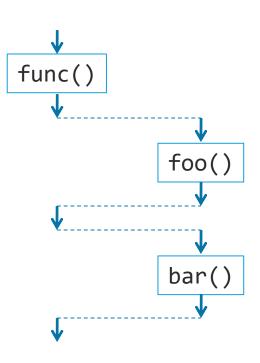
- Context: thread's current state
  - PC
  - Stack
  - Registers/flags
  - Memory map
- Threads share address space, but have local context



### Sequential Execution Model

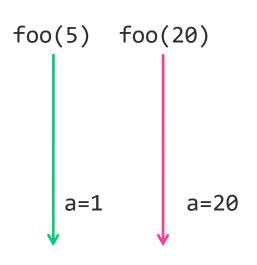
A single flow of instructions at any point in time. (This was not exactly always true, e.g., signals...)

```
func() {
...
foo(arg)
...
bar(arg)
...
return
}
```



Several flow of instructions at any point in time. How independent are they?

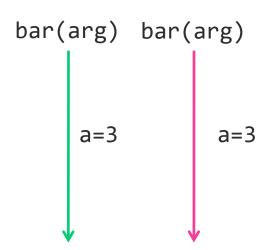
- Each thread executes independently of the others
- Therefore, several threads could be executing the very same instruction of the code
- But each thread has its own stack (remember how functions calls work?)



#### local variables

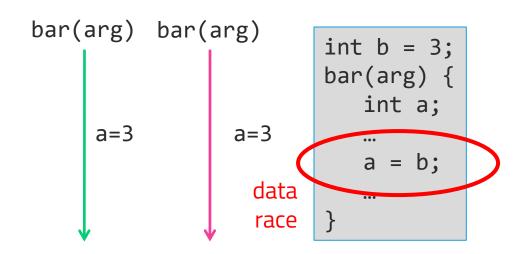
```
foo(arg) {
   int a;
   if (arg < 10)
        a = 1;
   else
        a = arg;
   ...
   ... access a ...
   return
}</pre>
```

But all threads share the same address space.



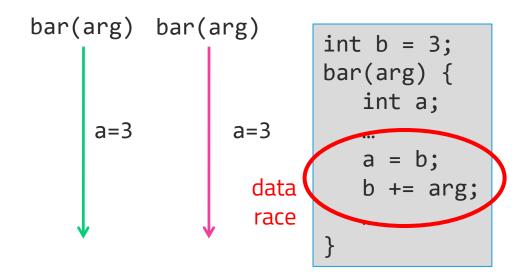
```
int b = 3;
bar(arg) {
    int a;
    ...
    a = b;
    ...
}
```

But all threads share the same address space.



## Thread Execution Model: Data Races

- What if two threads want to change a process variable?
- Rule of thumb: if it is not a local variable, do assume others may change it
- Simultaneous accesses may occur and have unpredictable results



#### Critical Sections

- We need a mutual exclusion mechanism.
- At most one thread can execute inside each critical section at any point in time.

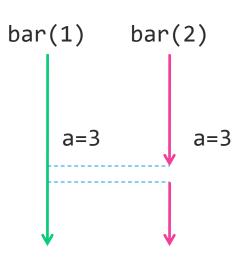
```
bar(1) bar(2)

a=3 a=3
```

```
int b = 3;
bar(arg) {
   int a;
   ...
   enter crit section
   a = b;
   b += arg
   leave crit section
}
```

#### Locks

- Used to implement critical sections
- Lock
  - If no other thread is in the critical section, proceed
  - Otherwise, wait
- Unlock
  - · Leave critical section
  - Allow another thread to enter, if one is waiting.



```
int b = 3;
bar(arg) {
    int a;
    ...
    lock
    a = b;
    b += arg
    unlock
}
```

## Threading and Synchronization Gotchas

- Simultaneous access & data races
  - Guard with locks if shared
- Single-instruction atomicity
  - x += 1;
  - Lock around anything that must look atomic
- Data atomicity
  - struct Foo a = struct Foo b;
- Data-race-free programs ONLY!

### Pthreads Concepts: Threads

- In this course: Pthreads (POSIX Threads)
  - Shared memory, shared file pointers: need explicit synchronization
- Create: pthread\_create()
- Join or detach: pthread\_join(), pthread\_detach()
- Exit: pthread\_exit()

### Pthreads Concepts: Mutices

- Mutex ("Mutual Exclusion"): pthread\_mutex()
- Lock that can be used for exclusive access to any shared resource(s)
- Programmer defines what is protected
  - Programmer responsible for locking/unlocking around access to protected resource

### Pthreads Concepts: Mutices

- Using a mutex
  - pthread\_mutex\_lock(mutex)
  - Use shared resources
  - pthread\_mutex\_unlock(mutex)
  - Do not use shared resource until lock is re-acquired

## Pthread Concepts: Condition Variables

**Threads** 

Mutices

Condition Variables

Semaphores

- pthread\_cond()
- Synchronization based on data values
- Always used with mutex
- Replaces:
  - 1. Lock mutex
  - 2. Check value of data
  - 3. Unlock mutex, repeat.

## Pthread Concepts: Semaphores

- Counting mutex: sem\_init()
- Atomically increase or decrease
- Sample use: communication management
  - sem\_post() (atomically increment) when sending new message to receiver
  - sem\_wait() (atomically decrement) to receive (or wait for) new message from sender

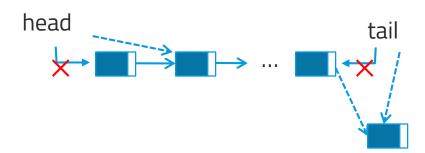


## Building a Concurrent Queue

## Building a Concurrent Queue

```
element dequeue():
    element = head.data
    head = head.next
    return element
```

```
enqueue(element)
  n = new node(element)
  tail.next = n
  tail = n
```

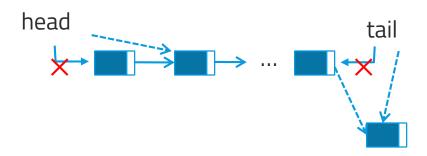


new node

## Building a Concurrent Queue

```
element dequeue():
    pthread_lock(queue_lock)
    element = head.data
    head = head.next
    pthread_unlock(queue_lock)
    return element

    enqueue(element)
    pthread_lock(queue_lock)
    n = new node(element)
    tail.next = n
    tail = n
    pthread_unlock(queue_lock)
```



new node

#### Invariants

- A way to reason about correctness.
- Queue consistency
  - To be consistent, a queue has to maintain certain properties throughout its lifetime
  - If the properties hold during the queue operations, then the queue is consistent
- What are the invariants that guarantee our queue consistency?

### Handling the Empty Queue Case: Invariants

```
element dequeue():
                                            enqueue(element)
 pthread lock(queue lock)
                                              pthread lock(queue lock)
 while (empty):
                                              n = new node(element)
                                              if tail != NULL:
    pthread cond wait(cond, queue lock)
                                                tail.next = n
 // Inv: queue is not empty
                                              tail = n
  element = head.data
 head = head.next
                                              // Inv: either !empty or head=tail=NULL
                                              empty = false
                                              pthread cond signal(cond)
 // Inv: either head=tail=NULL or !empty
 if head is NULL:
                                              pthread unlock(queue lock)
    tail = NULL
     empty = true
 pthread_unlock(queue_lock)
  return element
```

 Why do we wait with the lock taken? How can the signal code execute if signal is inside the lock?

•

•

- Why do we signal inside the lock?
  - •

- Why do we wait with the lock taken? How can the signal code execute if signal is inside the lock?
  - The wait code releases the lock if the thread goes to sleep and the wakeup only proceeds if it manages to reacquire the lock.
  - If the lock was not taken, by the time we got outside the while loop, the predicate the condition is on may not be true any more.
- Why do we signal inside the lock?
  - We don't have to but it's strongly suggested you do
  - What we must do is never to change the predicate outside the lock (data-race-free, right?)

• Why did we signal at every enqueue? Isn't our condition based on the queue being non-empty?

•

• Can we allow enqueue/dequeue in parallel (e.g., by using two locks instead of one)?

•

lacktriangle

- Why did we signal at every enqueue? Isn't our condition based on the queue being non-empty?
  - We're waiting on every dequeue. Signal/wait here cannot "peek into" our predicate. Our code needs to make the checking of the predicate itself because of the semantics of cond vars.
- Can we allow enqueue/dequeue in parallel (e.g., by using two locks instead of one)?
  - No, enqueue and dequeue are sharing state
  - Yes, if we made them not share state (hint: use a sentinel)

#### Lab 1

- Semester-long (2.25-month) lab: building a threaded server
  - HTTP-like: GET and POST
- Will test your abilities with:
  - Sockets
  - Threads
  - Concurrent data structures

#### Lab 1

- Due: Monday, October 23<sup>rd</sup>
- Create a multithreaded O(1) key-value data structure and O(1) queue data structure
  - 1. Create the (thread-safe) data structures
  - 2. Create fixed number of workers to service requests
- Preview
  - Lab 2: Add GET/POST frontend, hashing, and thread pool
  - Lab 3: Measure performance
  - Lab 4: Complete the system: on-disk cache