COSC 407
Intro to Parallel Computing

Topic 3 – Parallel Concepts and Threads

COSC 407: Intro to Parallel Computing

1

Qutline

Previous pre-recorded lecture (Students' led Q/As):

More on C programming:

- Pointers (intro, memory allocation, 2D arrays, functions)
- Error Handling, String processing
- struct, typedef
- Preprocessors, Compiling C programs

Today's topics:

- Intro to parallel computing
- Intro to POSIX Threads

Next Lecture:

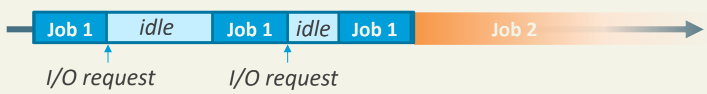
- More POSIX Threads

2

Sequential Systems

- a system in which each activity must complete execution before another activity can start.

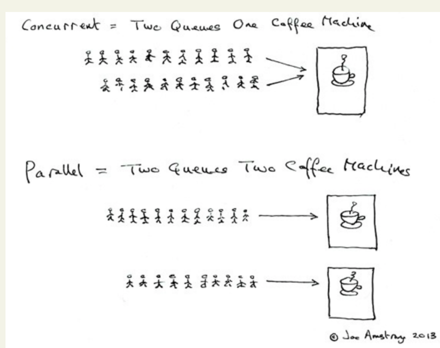
- **Benefit**: simplicity of design and use.
- **Cost**: might be slower in many cases
  - one job at a time (Not making use of the multiple cores in the system.)
  - the system must wait for its slower components if it needs to do more than one job (e.g. reading from a hard-drive (slow). Even if you want to do some processing that are not related to the data being read, you will still have to wait!).



3

Concurrency vs. Parallelism

- **Concurrency**
  - multiple operations are making progress during the same time period (but not necessarily running at the same time
  - Interleaving the execution steps of each task via time-sharing slices
    - i.e. Two 'programs' making progress on a single core processor



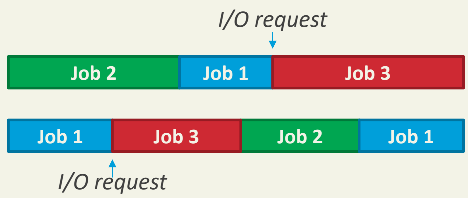
Credit: Joe Armstrong (co-author of Erlang Programming Language)

- **Parallelism**
  - Multiple operations are making progress at the exact same time.
  - i.e. two threads on a dual-core machine means that both threads can be executed at the same time

4

Concurrent Systems

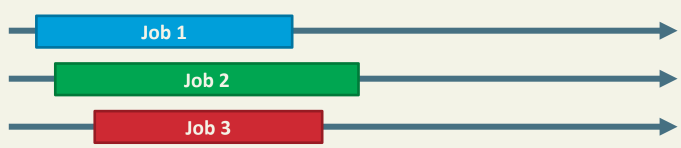
- A system that can run several of activities at the same time.
  - **Benefit:** efficient use resources, and probably faster.
  - **Cost**: complexity of hardware and software:



- Two types of concurrent systems:
  1. Parallel systems
  2. Pseudo-parallel systems

Concurrent Systems, *cont'd*

1. Parallel systems
   - More than one processor that can actually carry out several activities simultaneously



2. Pseudo-parallel systems
   - Share processor time between a number of activities (e.g., if we have two activities running on a single processor, then there is only one that is actually running at any time)

What is a "Process"

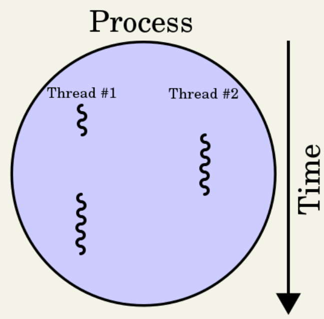
- An instance of a computer program that is being executed
- Components of a process:
  - The executable machine language program
  - A block of memory
  - Descriptors of resources the OS has allocated to the process
  - Security information
  - Information about the state of the process

Multitasking

- Gives the illusion that a single processor system is running multiple programs simultaneously
- Each process takes turns running
  - **time slice**
  - After its time is up, it waits until it has a turn again
  - **blocks** – nothing proceeds with this program until the next time it has a turn

Threading

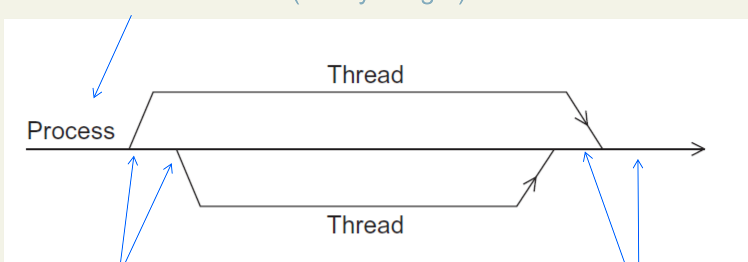
- Threads are contained within processes
- They allow programmers to divide their programs into (more or less) independent tasks
  - A stream of instructions that can be scheduled to run independently from its main program
- The hope is that when one thread blocks because it is waiting on a resource, another will have work to do and can run



Attribution Churnett Creative Commons Attribution-Share Alike 3.0 Unported https://upload.wikimedia.org/wikipedia/commons/a/a5/Multithreaded_process.svg

Process and Threads

the "master" thread (heavy-weight)



starting a thread
Is called ***forking***
(light weight)

terminating a thread
Is called ***joining***

Processes vs. Threads

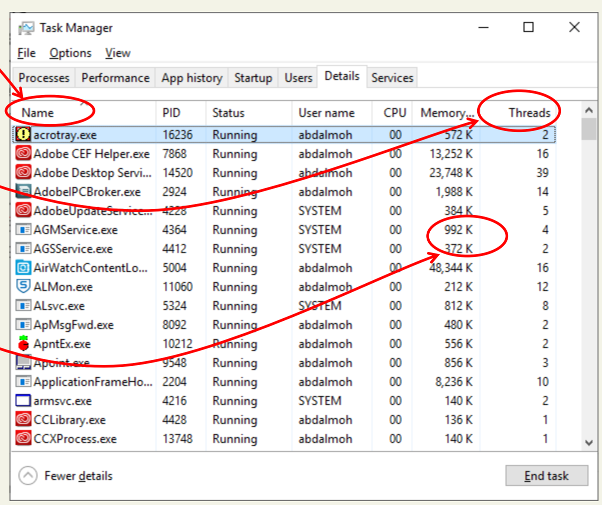
- Threads exist within a process
  - every process has at least one thread
  - A process is multithreaded when it contains more than one thread of execution.
- Both threads and processes are **units of execution** (tasks)
  - Both are items that can be scheduled for execution
- But
  - **Processes** will, by default, **<u>not share memory</u>**.
  - **Threads** of the same process will, by default, have access to the same **shared memory** (the process memory)
    - Data can be shared or private
    - Private data is available only to the thread that owns it

Topic 3: Parallel Concepts and Threads — COSC 407: Intro to Parallel Computing

11

Processes vs. Threads

- A **process** is **multithreaded** when it **contains more than one thread** of execution.
- Memory used by a process and shared by its threads


Task Manager

File Options View

Processes | Performance | App history | Startup | Users | Details | Services

Name	PID	Status	User name	CPU	Memory...	Threads
acrotray.exe	16236	Running	abdalmoh	00	572 K	2
Adobe CEF Helper.exe	7868	Running	abdalmoh	00	13,252 K	16
Adobe Desktop Servi...	14520	Running	abdalmoh	00	23,748 K	39
AdobeIPCBroker.exe	2924	Running	abdalmoh	00	1,988 K	14
AdobeUpdateService...	4228	Running	SYSTEM	00	384 K	5
AGMService.exe	4364	Running	SYSTEM	00	992 K	4
AGSService.exe	4412	Running	SYSTEM	00	372 K	2
AirWatchContentLo...	5004	Running	abdalmoh	00	48,344 K	16
ALMon.exe	11060	Running	abdalmoh	00	212 K	12
ALsvc.exe	5324	Running	SYSTEM	00	812 K	8
ApMsgFwd.exe	8092	Running	abdalmoh	00	480 K	2
ApntEx.exe	10212	Running	abdalmoh	00	556 K	2
Apoint.exe	9548	Running	abdalmoh	00	856 K	3
ApplicationFrameHo...	2204	Running	abdalmoh	00	8,236 K	10
armsvc.exe	4216	Running	SYSTEM	00	140 K	2
CCLibrary.exe	4428	Running	abdalmoh	00	136 K	1
CCXProcess.exe	13748	Running	abdalmoh	00	140 K	1

Fewer details — End task

Topic 3: Parallel Concepts and Threads — COSC 407: Intro to Parallel Computing

12

Multi-thingys

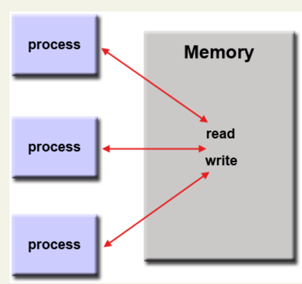
- **Multithreaded process**:
  - a process having more than one thread.
  - A thread of execution is a **sequence of instructions** that can be **managed independently (executed, stopped, etc.).**
- **Multitasking:** the ability of a platform to run more than one task (thread or a process) concurrently.
  - Two multitasking operating systems:
    1. Pre-emptive multitasking
       - The OS decides when a task should give way to another to allow sharing of resources (e.g. Windows 95 and later)
    2. Cooperative multitasking
       - The process is coded such that it decides when to allow other processes to run from time to time (e.g. Windows 3.1 and prior).

Topic 3: Parallel Concepts and Threads — COSC 407: Intro to Parallel Computing

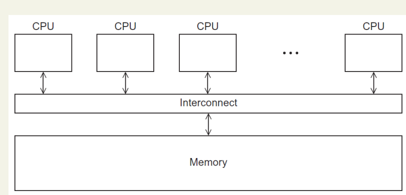
13

How are Things Shared?

- Shared Memory
  - In general, allows processors to have access to global address space
  - Multiple processors/processes can operate independently but share the same memory resources
  - Changes in a memory location effected by one processor are visible to all other processors



https://hpc.llnl.gov/training/tutorials/introduction-parallel-computing-tutorial#MemoryArch

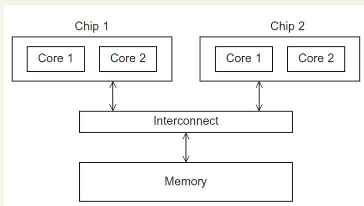


Topic 3: Parallel Concepts and Threads — COSC 407: Intro to Parallel Computing

14

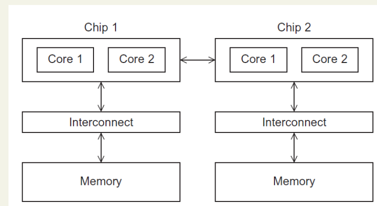
How are Things Shared?

UMA
(uniform memory access)



- Time to access all the memory locations will be the same for all the cores

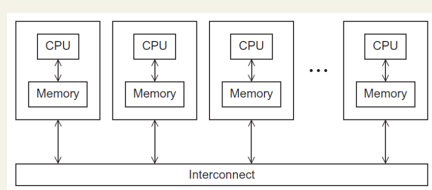
NUMA
(non-uniform memory access)



- A memory location a core is directly connected to can be accessed faster than a memory location that must be accessed through another chip.

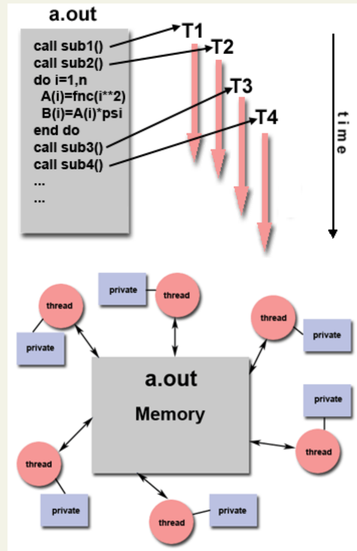
How are Things Shared?

- Distributed memory
  (more later on this)
  – Clusters (most popular)
    • ie a group machines
  – A collection of commodity systems
  – Connected by a commodity interconnection network
  – Nodes of a cluster are individual computations units joined by a communication network



Back to Threads..
Thread Model

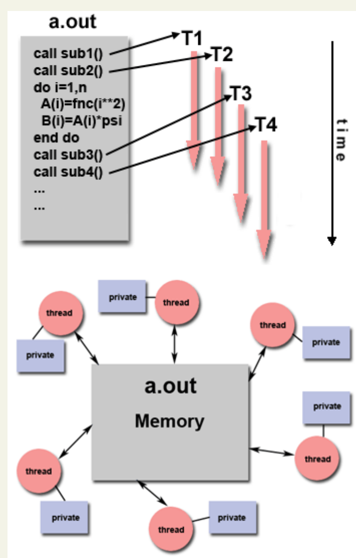
- POSIX/openMP
- Type of shared memory programming
- **a.out** is scheduled to run by the native operating system and acquires all of the necessary system and user resources to run
  - "heavy weight" process
- Does some serial work, and then creates a number of tasks (**threads**) that can be scheduled and run by the operating system concurrently
- Programmer is responsible for determining the parallelism



https://hpc.llnl.gov/training/tutorials/introduction-parallel-computing-tutorial

Back to Threads..
Thread Model

- Each thread has local data and shares the entire resources of **a.out**
  - Saves the overhead associated with replicating a program's resources for each thread ("light weight")
  - Each thread also benefits from a global memory view because it shares the memory space of **a.out**.
- Threads communicate with each other through global memory (updating address locations)
- Requires **synchronization** to ensure that more than one thread is not updating the **same global address** at any time
- Threads can come and go, but **a.out remains present to provide the necessary shared resources until the application has completed**



https://hpc.llnl.gov/training/tutorials/introduction-parallel-computing-tutorial

Task Scheduling

- A **scheduler** is a computer application that uses a **scheduling policy** to decide which processes should run next.
  - The scheduler uses a **selection function** to make the decision.
  - The selection function considers several **factors** such as:
    - **Resources** the processes requires (and whether they are available)
    - **Time** processes have been **waiting** and/or **been executing**
    - The processes **priority**
- The **scheduling policy** should try to **optimize** many characteristics such as:
  - **Responsiveness** of interactive processes
  - **Turnaround**: the time the user waits for the processes to finish
  - **Resource utilization** (e.g., keep the CPU busy)
  - **Fairness**: dividing the CPU time fairly
    - This is opposite to ‘**starvation**’ where a processes is not given the chance to run for a long time

Some Scheduling Policies

Non-pre-emptive policies (*each task runs to completion before the next one can run*)

- First-In-First-Out (FIFO)
  - Tasks are placed in a queue as they arrive.
- Shortest-Job-First (SJF)
  - The process that requires the least execution time is picked next

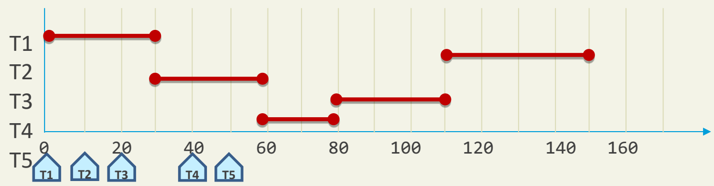
Pre-emptive

- Round-Robin (RR)
  - Each task is assigned a fixed time to run before it is required to give way to the next task and move back to the queue (e.g., at the end of the queue). RR is pre-emptive policy.
- Earliest-Deadline-First (EDF)
  - The process with the closes deadline is picked next. (pre-emptive)
- Shortest-Remaining-Time-First (SRTF)
  - The process with the shortest remaining time is picked next (pre-emptive)

Shortest-Job-First

- Consider the **SJF** scheduling policy on the processes in the table below and find finish and turnaround times according to the given values
  - Turnaround time is the total amount of time spent by the process from coming in the ready state for the first time to its completion (Turnaround time = Exit time - Arrival time)

ID	Arrival time	Service time	Finish time	Turnaround
T1	0	30		
T2	10	40		
T3	20	30		
T4	40	30		
T5	50	20		



Some Key Terms

- **Shared resource:** a resource available to (shared by) all processes in a concurrent program
  - e.g., a shared data-structure
- **Critical section:** section of code with a process that requires access to **shared resources [1]**
  - **Cannot be executed while another process is in a corresponding section of code**
  - e.g. updating a shared resource
  - Critical sections should be **executed as serial code**
- **Mutual Exclusion:** requirement that when one process is in a **critical section** that accesses a **shared resource, no other process may be in a critical section that accesses any of those shared resources [1]**
  - a mechanism to ensure that no two concurrent processes access a critical section at the same time

[1] Stallings, William. (2005). Operating systems : internals and design principles. Upper Saddle River, N.J. :Prentice Hall,

Some Key Terms

- **Condition synchronization:** a mechanism to make sure that a process does not proceed (i.e., blocked) until a certain condition is satisfied
  - For example, a consumer process cannot read from a buffer unless there is some data written to the butter. Similarly, a producer process cannot write to a full buffer
  - Synchronization lock mechanism
- **Deadlock:** a situation in which **two or more processes** are **unable to proceed** because each is **waiting for one of the other processes to do something** [1]
- **Livelock:** a situation in which **two or more processes continuously change their state** in response to changes in other processes **without making any process (no useful work)[1]**

[1] Stallings, William. (2005). Operating systems : internals and design principles. Upper Saddle River, N.J. :Prentice Hall,

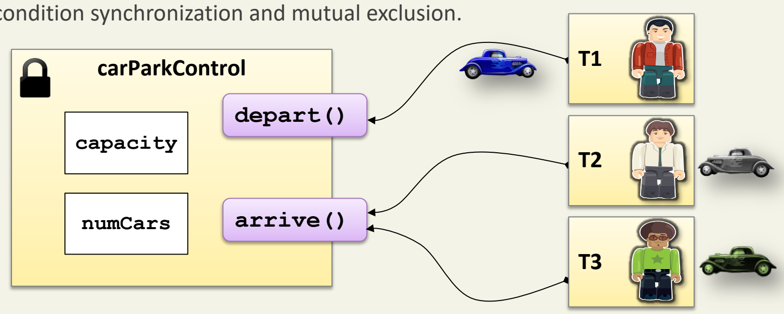
Some Key Terms

- **Race Condition:** a situation where multiple threads/processes read/write a shared data item and the result depends on the relative timing of their execution
- **Starvation:** a situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen [1]
  - Occurs when a thread is unable to gain access to a shared resource and thus cannot make progress

[1] Stallings, William. (2005). Operating systems : internals and design principles. Upper Saddle River, N.J. :Prentice Hall,

Example (Synch/Mutal Ex)

- Assume we have an object named carParkControl:
  - *attributes*: capacity and spaces
  - *methods*: depart() and arrive()
- Now assume three threads are using carParkControl:
  - T1 first invokes the depart method
  - **Then** T2 and T3 both invoke the arrive method.
  - Assume the car park is **initially empty** (numCars=0) – i.e. no cars to depart !!
- Two important aspects to insure the code works properly:
  - condition synchronization and mutual exclusion.



A possible scenario

1. T1 **invokes the `depart`** method and **acquires** the **synchronization lock**.
2. T1 examines the **condition** and determines that the data is not in the desired state.
3. T1 invokes a `wait` method, which **frees the lock**; T1 is in **WAITING** state now.
4. **Scheduler** allows **T2** to run (**RUNNING** state)
5. T2 **invokes the `arrive`** method, acquiring the **synchronization lock**.
6. If **T3** gets a chance to **run**, and attempts to invoke `arrive`, it'll end up in the **BLOCKED** state waiting for the lock.
7. **T2 continues to run**, checks the condition and finds that it can continue without having to call `wait`. It then updates the data.
8. **T2** finishes the `arrive` method, **frees the lock,** and **`notifies`** all threads it is done.
9. Both **T1 and T3 receives the notification** and become **RUNNABLE** (not running yet)
10. T1 and T3 now both try to get hold of the lock.
11. Assume **T1 obtains the lock** now, it checks the condition and finds that the condition is satisfied, so it processes the data.
12. **T1** exits `depart` method and **gives up the lock**, then **`notifies`** everyone it is done.
13. **T3** now has a chance to **run**, acquires the **lock** and **checks the condition**, which is satisfied. It changes the data and sends a notification and completes the method.
14. The lock is freed up.

Dead or Alive(lock)….

- Concurrent programs must satisfy two properties:
  - **Safety**: the program doesn't enter a bad state (e.g. wrong output)
    - i.e., performs **according to its specification**.
  - **Liveness**: the program must perform **progress**.
- Two problems related to **liveness**:

Deadlock (see key terms):

A process is waiting for a resource that will never be available. This resource may be is held by another process that waits for the first process to finish.

– E.g., Dining philosopher's problem

Livelock (see key terms):

A process is continuously running but without making any progress.

– e.g.1 process P1 acts in response to P2, and then P2 reverses the action of P1, bringing both to their initial state before the action.

– e.g.2, two people meeting in a corridor and moving continuously sideways without making progress.

Deadlock

For a deadlock to happen, four conditions must hold :

1. **Mutual Exclusion**
   - The program involves a shared resource protected by mutual exclusion (only one process can have that resources)
2. **Hold while waiting**
   - A process can hold a resource while it is waiting for other resources
3. **No pre-emption**
   - The OS cannot force a process to deallocate the resource it holds (the process must deallocate it voluntarily)
4. **Circular wait**
   - P1 is waiting for a resource held by P2, and P2 is waiting for a resource held by P1

Dealing with Deadlock

1. **Prevention**
   – Prevent deadlock by preventing one of the four deadlock conditions from occurring.
2. **Detection**
   – Numerous algorithms exist which enable the detection of deadlock
   – A data structure, such as a table, is used to record:
     • all objects in the system + the requests for locks on objects + the allocations
     • Through analysis of this data it is possible to detect whether or not deadlock exists
     • If deadlock is detected, the processes involved in the deadlock can be aborted.
3. **Ignoring deadlock**
   – If deadlocks rarely happen and the data loss incurred each time is tolerable

Data Race and Race Condition

- **Data Race** (see key terms)
  – Occurs when ALL of the following 3 conditions are satisfied:
    • **2 or more** concurrent threads are accessing **same memory location**
    • at least one thread is **writing** to the shared location
    • there is **no** synchronization that is enforces ***any* particular order** among these accesses
  – The computation may give different results from run to run.
  – The discovery of data races can be automated.
- **Race Condition:**
  – Program or output state depends on the ordering of events (by threads)
  – Many race conditions ***can be* caused by data races**, *but this is not necessary*.
  – Most race conditions are data races
    • But you can have a race condition without data race

Discussion

- Assuming x is a shared resource initially equal to 10. Two threads are accessing x using the following code (note that we are enforcing mutual exclusion when accessing x). **Is there data race**? Discuss
  - *(the number beside lock/unlock indicates we are using the same lock for ensuring mutual exclusion of x)*

```
Thread 1         Thread 2
 lock(l)         lock(l)
 x = x+2;         x = x*2
 unlock(l)       unlock(l)
```

A. Yes

B. No

C. What do you mean by data race??

POSIX Threads

- Historically, hardware vendors have implemented their own proprietary versions of threads
  - These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications
- In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required.
  - For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995).
  - Implementations adhering to this standard are referred to as POSIX threads, or Pthreads.
  - Most hardware vendors now offer Pthreads in addition to their proprietary API's.
- The POSIX standard has continued to evolve and undergo revisions, including the Pthreads specification
- Pthreads are defined as a set of C language programming types and procedure calls
  - pthread.h header/include file and a thread library (comment on windows)
  - this library may be part of another library, such as libc, in some implementations

https://hpc-tutorials.llnl.gov/posix/why_pthreads/

Why Pthreads?

- Light weight
  - Much less overhead than creating a process
  - Requires fewer system resources
- Efficient communications/data exchange
  - For Pthreads there is no intermediate memory copy required because threads share the same address space within a single process
  - There is no data transfer as it can be as efficient as simply passing a pointer
- Efficiently interleave tasks

Pthreads API

- Library has around 100 functions (we are just going to examine a few)
  - **Thread management:**
    - Routines that work directly on threads - creating, detaching, joining, etc
    - They also include functions to set/query thread attributes (joinable, scheduling etc.)
  - **Mutexes**:
    - Routines that deal with synchronization, called a "**mutex**", which is an abbreviation for "mutual exclusion"
    - Provides for creating, destroying, locking and unlocking mutexes
  - **Condition variables:**
    - Routines that address communications between threads that share a mutex and are based upon programmer specified conditions
  - **Synchronization:**
    - Routines that manage read/write locks and barriers.

Hello World(s)….

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

void* say_hello(void* data)
{
    char *str;
    str = (char*)data;
    while(1)
    {
        printf("%s\n",str);
        sleep(1);
    }
}

void main()
{
    pthread_t t1,t2;

    pthread_create(&t1,NULL,say_hello,"hello from 1");
    pthread_create(&t2,NULL,say_hello,"hello from 2");
    pthread_join(t1,NULL);
}
```

1) Include pthread.h

2) `pthread_t` is a data type that holds information about threads.
Each thread requires one pthread_t variable

3) Create a thread

3) Function to execute

3) Data to pass to fn

4) Waits for t1 to finish…… will it?

Compiling

- Need to include pthread library

Compiler / Platform	Compiler Command	Description
INTEL Linux	icc -pthread	C
	icpc -pthread	C++
PGI Linux	pgcc -lpthread	C
	pgCC -lpthread	C++
GNU Linux, Blue Gene	gcc -pthread	GNU C
	g++ -pthread	GNU C++
IBM Blue Gene	bgxlc_r / bgcc_r	C (ANSI / non-ANSI)
	bgxlC_r, bgxlc++_r	C++

```
gcc test.c -o test -lpthread
```

- mingw issue….
- Use https://www.onlinegdb.com for testing as we are interested in understanding the behavior of threads

Conclusion/Up Next

- What we covered today (review key concepts):
  - Intro to parallel computing
  - Intro to POSIX Threads
- Next Lecture:
  - Parallel concepts with Pthreads
  - Mutexes
  - Thread management
  - Synchronization
  - Condition variables
  - CPU and I/O bound threads

Homework

- Please review
  - POSIX Threads Programming
    - https://hpc-tutorials.llnl.gov/posix/ (sections 1 – 8)
  - Additional resources on Canvas