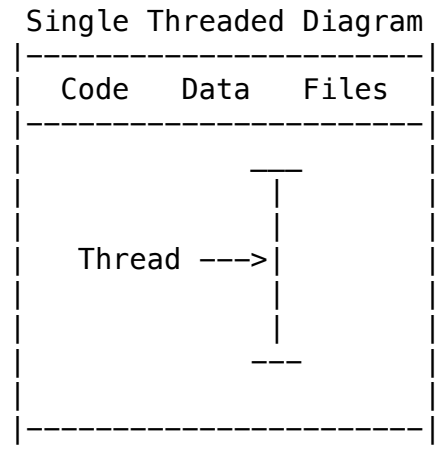
Lecture.5.Threads.txt

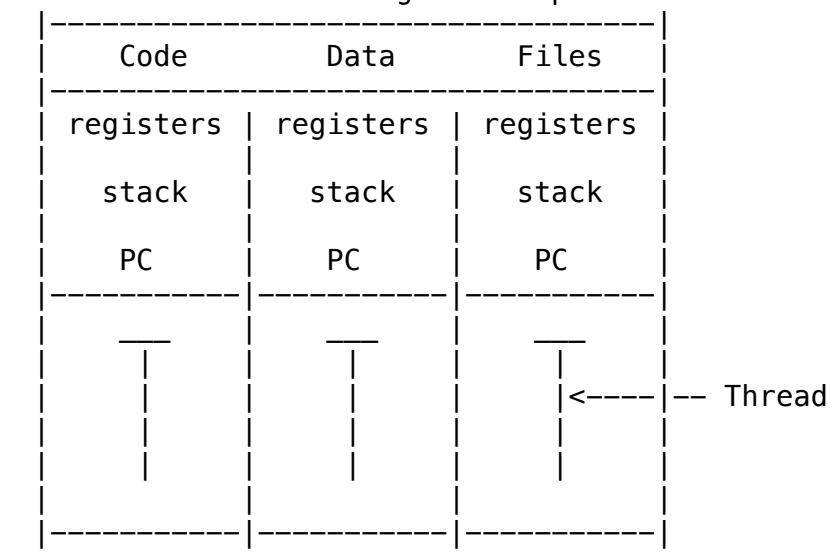
- What Is A Thread?
    - A sequential execution stream (thread) within a process
        - Also called lightweight process
    - A process has at least one thread of control
        - This is the main/primary thread
    - A process has two parts:
        - Threads (concurrency)
        - Address spaces (protection)
            - Processes do not share address space, but threads do share address space
    - All threads are processes, but not all processes are threads
        - Threads are a subset of processes
            - Threads cannot exist without a running process
            - Multiple threads can be part of the same process
                - Multiple processes can be part of the same program

- Motivation
    - Most modern applications are multithreaded
        - i.e. A word processor may have separate/unique threads for:
                - Displaying graphics
                - Responding to keystrokes from the user
                - Spell/grammar checking
        - Each thread takes care of a different task/job
    - Multithreading can simplify code, and increase efficiency
    - Multiprocessing (systems) splits a program into multiple threads to make it run faster by running it on multiple processors
        - Each thread is carried out on a different core
        - This leads to parallel programming
    - Parallel programming VS. Concurrent programming
        - Concurrent programming is done on one core
        - Parallel programming requires multiple cores
            - i.e. Two and/or more
                - At least two cores
        - They are separate entities
            - One is not a subset of the other

- Single & Multithreaded Processes
    - Singlethreaded process
        - The process has two parts:
            1. register, program counter (PC), stack
                - This is local data
                - The program counter is a special register
            2. code, data, files
                - This is global data
        - Has one thread
            - This thread has access to local and global data
                - All data is dedicated to one thread
        - i.e.



- Multithreaded process
  - Has multiple threads
    - Each thread has its own (dedicated) set of registers, stack, and program counter (PC)
      - The local variables for each thread are different
    - The code, data, and files are shared among all threads
      - The global data is the same for all threads
      - Code is not duplicated when a new thread is created
  - Each thread has its own dedicated register
    - This is why you cannot have as many threads as you want
    - Creating another thread is less intensive than creating another process
  - i.e.



  - Context switching is for processes
  - Thread switching is for threads
    - Much faster than context switching

- Multithreaded Server Architecture
  - Web server accepts client requests for web pages, images, sound, and so forth

        - Each client request creates a new thread/job to serve the request
            - The old approach to this was to create a new process to serve the request
                - This is time consuming and resource intensive
            - The new approach is to create a new thread to service incoming requests, and then resume listening for new requests
                - This decreases the amount of resources needed and speeds up the entire process

- Thread States
    - The states that are shared by all threads in the same process or address space are:
        - Global variables
        - File system
            - i.e. Data
        - Code
    - The states that are private to each thread, regardless of whether they are in the same process or not, are:
        - Program counter (PC)
        - Registers
        - Stack
            - Contains parameters, temporary variables, and return addresses
    - The biggest challenge is sharing global variables
        - The OS should help take care of this

- Benefits
    - Responsiveness
        - May allow continued execution if part of process is blocked
            - Time consuming operations are performed in a separate asynchronous thread, the application remains responsive to the user
                - This is especially important for user interfaces
                - i.e. Upload video to YouTube, and simultaneously watch YouTube/Netflix, or check Email
    - Resource Sharing
        - Processes can exchange data through shared memory, and message passing
            - But, this must be explicitly arranged by the programmer
        - Threads share resources of process by default
    - Economy
        - Thread switching has lower overhead than context switching
        - Thread creation consumes less time and memory than process creation
            - This helps to improve the overall speed
    - Scalability
        - Multithreading can take advantage of multicore architectures where threads may be running in parallel on different

processing cores
- The OS determines what thread/process runs on which core (CPU)
  - However, this can be changed and programmed
    - A programmer can specify if the computation should be carried out in parallel, or not
- Sometimes, there are computations that cannot be paralleized, and some computations will have no improvements if parallelized
  - This occurs when there is a high data dependency
  - i.e. Imagine you have a set of data, X. You compute some statistic and get Y. If the subsequent calculations rely on the previous data (i.e. In order to get Z, you need both X and Y), then you cannot parallelize this.

- Multicore Programming
  - Multicore or multiprocessor systems are harder for programmers to develop
    - Some challenges of multicore programming are:
      - Dividing activities
      - Balance
        - i.e. Core #1 does more work than core #2
      - Data splitting
      - Data dependency
        - i.e. Core #2 cannot compute if core #1 has not finished
      - Testing and debugging
        - Very complex with multicore programming
          - Because the possibility of states grows exponentially
          - Covering all test cases is not possible nor feasible
  - Parallelism implies that a system can perform more than one task simultaneously
    - Cannot occur on a single core processor
      - At least two cores are required
  - Concurrency supports more than one task making progress
    - Possible on a single processor/core
      - The scheduler provides concurrency
    - CPU switches between two or more tasks, very quickly
  - It's very important to understand the difference between parallelism and concurrency
    - They are separate entities; one is not a subset of the other

- Multicore Programming
  - There are two types of parallelism in multicore programming
    1. Data Parallelism
      - Distributes subsets of the same data across multiple cores, and performs the same operation on each core
        - i.e. Sorting a huge list. If a system has 4 cores, then the list is divided into 4 sections, and

each core sorts a different section using Quicksort. Finally, MergeSort is used to combine each piece

2. Task Parallelism
    - Distributes threads across cores, each thread performing unique operation on the same data
        - i.e. You have a huge set of numerical data containing weather information for 2020. Each core operates on the same dataset, but performs different operations on it. For example, core #1 will get the average temperature for the year, and core #2 will get themonthly average temperature, and core #3/#4 will calculate the median and mode, respectively.

- Example: Summing an array of size N
    - Single core: Sum the elements [0]..[N - 1]
    - Dual core: Thread #1 sums the elements from [0]..[N/2 - 1],
      AND (simultaneously)
      Thread #2 sums the elements from [N/2]..[N - 1]
        - Data is divided into two equal parts, each core performs summation on one half of the array, and they combine their result at the end
            - This is data parallelism

- Data & Task Parallelism
    - Data Parallelism
        - If a system has four cores, then the data is divided into 4 pieces, and each core operates on one piece of the data
            - i.e. Splitting a large array of numbers into 4 pieces & each core sums up all of the numbers in its piece
            - i.e. Brute forcing a password. A dictionary is split into 4 sub-lists and each core tries a list
    - Task Parallelism
        - If a system has four cores, each core is operating on the same data, and performing its own operations on the data
            - i.e. Computing statistics - mean, median, mode, etc. - on the same set of age data for a population
    - In reality, both data and task parallelism are used to compute data
        - i.e. Data parallelism is used to split the data into four parts, and then task parallelism is used to operate on each part
        - i.e. Assume you have a six core processor, and weather data for all of 2020. First it gets split into 2 parts, where core #1 and core #4 handle each part. Then, core #2/#3 operate on the first part of the data, while core #5/#6 operate on the second part of the data. So, maybe, core #1/#4 are sorting the weather data for their respective sections while core #2/#3 and core #5/#6 are computing statistics

(i.e. Mean, mode, median, etc.) for their part.

- Amdahl's Law
  - Identifies performance gains from adding additional cores to an application that has both serial and parallel components
    - We cannot parallelize everything due to serial code
      - Even with an infinite number of cores, the program cannot be sped up
    - If most of the code is serial, then the speedup from parallelization is smaller
  - The formula for Amdahl's law is:

$$\text{Speedup} \leq \frac{1}{S + \frac{(1 - S)}{N}}$$

    - Where:
      - S = Serial portion (%) of code
      - N = Number of processing cores
    - There is a limit to 'Speedup', it does not go to infinity
    - Sample Question:
      - What is the speedup of an application if it is 75% parallel, and 25% serial?
  - If there is a lot of data dependency, then it is very hard to parallelize the application/process

- User Threads & Kernel Threads
  - User threads
    - Threads that are running inside the user-space
    - Management is done by user-level thread libraries
      - There are three primary thread libraries:
        - POSIX Pthreads
          - Will be used in labs
        - Windows threads
        - Java threads
  - Kernel threads
    - Threads that are running inside the kernel (space)
      - Supported by the Kernel
    - Used on:
      - Windows
      - Linux
      - macOS
      - iOS
      - Android
  - Different types of mappings between user and kernel threads are:
    - Many user threads mapped to one kernel thread
    - One user thread mapped to one kernel thread
    - Many user threads mapped to many kernel threads
    - Multi level model

        - Incorporates two or more of the other models
            - i.e. One-to-one and many-to-many

- Many User Threads Mapped To One Kernel Thread
    - One kernel (space) thread is mapped to multiple threads in user space
    - The kernel (space) thread can only run one thread in user space
        - If one thread is blocked, then it causes all threads to be blocked
            - This is bad
    - Multiple user threads may not run in parallel on multicore systems because only one thread may be in the kernel at a time
        - This is bad
    - Few systems currently use this model:
        - Solaris Green Threads
        - GNU Portable Threads

- One To One
    - Each user level thread maps to one kernel thread
        - Is a one to one relationship
        - This offers more concurrency than many to one
            - This is good
    - Number of threads per process may sometimes be restricted due to overhead
    - The one to one model is the best model
        - It is the easiest to implement with little downside
        - It is used by most of the operating systems
            - i.e.
                - Windows
                - Linux

- Many To Many
    - Some operating systems use a many to many relationship for mapping user space threads to kernel (space) threads
    - Limits the number of threads in the kernel
        - The number of threads in the kernel space is less than the number of threads in user space
    - Allows the operating system to create a sufficient number of kernel threads
        - i.e. A smaller or equal number of user threads
    - This is very hard to implement
        - Thus, it is not very common
            - But it is possible on Windows using the ThreadFiber package

- Two Level Model
    - Incorporates two models into the system; many-to-many, and one-to-one
        - Some applications follow the many-to-many model, and others follow the one-to-one model

        - A single user thread can be bound to a single kernel thread
    - Although the many-to-many (M:M) model appears to be the most flexible, in practice it is difficult to implement
    - Most operating systems use the one-to-one model
        - i.e. Windows, Linux, etc.

- Q/A (Lecture Introduction)
    - Concurrent systems are harder to debug because the state space of the system has a much higher combination of different states in which the system can be in
        - More threads exponentially increase the number of possible combinations that can be in your system
        - This issue is also present in parallel programming

- Thread Libraries
    - Thread libraries provide programmers with an API for creating and managing threads
    - There are two primary ways of implementing threads:
        - Library entirely in user space
        - Kernel level library supported by the OS
            - Done via system calls

- Pthreads
    - Maybe provided by either as user-level or kernel-level
    - When dealing with libraries, it is important to have a specification or standard that defines it
        - For the Pthread library it is the POSIX specification standard API for thread creation and synchronization
            - i.e. IEEE 1003.1c
        - A general rule for standards is that they can only be defined by behaviour, not implementation
            - The implementation can be different but the behaviour should be the same
                - i.e. A square function takes in an integer (n) and returns the square of that number. This is the behavior. You can implement this any way you want. You can multiple the number by itself, or add the number to itself 'n' times.
    - The API specifies the behaviour of the thread library
        - The implementation is left up to the development of the library
    - Pthreads are common in Unix, Linux, & Mac OS X

- Pthreads Example
    - Program name: sum.c
        - i.e.

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

```
/* This data is shared by the thread(s) */
int sum;

/* The thread */
void *runner(void *param);

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

    /* The thread identifier */
    pthread_t tid;

    /* Set of attributes for the thread */
    pthread_attr_t attr;

    /* Get the default attributes */
    pthread_attr_init(&attr);

    /* Create the thread */
    pthread_create(&tid, &attr, runner, argv[1]);

    /* Wait for the thread to exit */
    pthread_join(tid, NULL);

    printf("sum = %d\n", sum);

    return 0;
}

/**
 * The thread will begin control in this function
 */
void *runner(void *param) {

    // atoi is a function that converts strings into ints
    int i, upper = atoi(param);
    sum = 0;

    if (upper > 0) {
        for (i = 1; i <= upper; i++) {
            sum += i;
        }
    }
    pthread_exit(0);
}

// On Linux compile with:
// gcc -g -Wall -pthread sum.c -lpthread -o sum
```

- Threaded Programming Challenges
    - Writing programs with multiple simultaneous points of execution,

synchronizing with shared memory
  - Can be solved with mutex, semaphores, monitors, etc.
  - Without multi-threading, a program will have less bugs, but it will run significantly slower
- Global variables are shared among all the threads of the same process
- Threads can read and write the same memory locations
  - Multiple threads reading the same memory location is not a problem
  - Multiple threads writing to the same memory address can create lots of problems
- The programmer is reponsible for using the synchronization mechanisms of the thread facility to ensure that the shared memory is accessed in a manner that will give the correct answer
  - If a thread is reading/writing on a memory location, it needs to have exclusivity on it

- Implicit Threading
  - As the number of threads increase in a program, implicit threading is gaining traction and growing in popularity
    - Also, program correctness is more difficult with explicit threads
  - The creation and management of (implicit) threads are done by compilers, and run-time libraries rather than programmers
    - This can prevent some errors that occur due to careless programmers when they use explicit threading
  - There are five methods for this, they are:
    1. Thread Pools
    2. Fork-Join
    3. OpenMP
    4. Grand Central Dispatch
       - This is for Apple's line of computers
    5. Intel Threading Building Blocks

- Thread Pools
  - Thread pools create a number of threads in a pool where they await work
    - The thread is created before you need it; this eliminates the process of thread creation
      - When you need the thread, you enable it and give it CPU time
  - The advantages of thread pools are:
    - It is usually (slightly) faster to service a request with an existing thread than create a new thread
    - Allows the number of threads in the application(s) to be bound by the size of the pool
      - This solves the problem of creating too many threads
    - Separating the task to be performed from mechanics of creating the task allows different strategies for running it
      - Tasks could be scheduled to run periodically

- Fork Join Parallelism
  - Multiple threads (tasks) are forked, and then joined
  - i.e.

```
Task(problem) {
    if (problem is small enough) {
        solve the problem directly
    } else { // Use threads to solve big problems
        s1 = fork (new Task (subset of problem));
        s2 = fork (new Task (subset of problem));
        // Problem is split across two threads

        result1 = join(s1);
        result2 = join(s2);

        // Results are combined at the end
        return (results1 + results2);
    }
}
```

- Open Multi-Processing (OpenMP)
  - Set of compiler directives and an API for C, C++, FORTRAN
  - Support for parallel programming in shared-memory
  - Identifies parallel regions
    - These are blocks of code that can run in parallel
  - Sample code:
    - i.e.

```
#include <omp.h>
#inlcude <stdio.h>

int main(int argc, char * argv[]) {
    /* Sequential code */

    /*
     * Tells the compiler to try to execute this
     * portion of the code in parallel
     */
    #pragma omp parallel {
        printf("I am a parallel region");
    }

    /* Sequential code */
}
```

  - More information at:
    https://openmp.org/

- Grand Central Dispatch
  - Apple's implicit threading implementation for macOS and iOS operating systems
  - Provides extensions to C, C++, and Objective-C languages, API,

  and the run-time library
- Blocks are placed in dispatch queue
  - Then, they are assigned to an available thread in a thread pool when removed from the queue
    - As a programmer you do not know how this will be done in parallel, or when it will be done
      - All you know is that you want it to be done in parallel
  - There's a queue for single core processor, and another queue for multicore processors
- Syntax:
  ^{ printf ("I am a block"); }

- Intel Threading Building Blocks (TBB)
  - Intel's template library for designing parallel C++ programs
  - Syntax:
    - Normal C Code:

```
// simple for loop in C
for (int i = 0; i < n; i++) {
    apply(v[i]);
}
```

    - TBB Code:

```
// TBB version of a simple for loop
parallel for (size_t (0)), n,
    [=] (size_t i) {apply(v[i]);}
);
```

- Issues In Designing Multithreaded Programs
  - Some issues in multithreaded programming are:
    - Semantics of 'fork()' and 'exec()' system calls
      - Assume you have a process called 'P' running 2 additional threads. If you call 'fork()' on 'P' and create a child process, does it have the 2 additional threads or no?
    - Signal handling; synchronous and asynchronous
    - Thread cancellation of target thread
      - i.e. Asynchronous or Deferred?
        - Deferred cancellation stops the thread at later time
    - Thread local storage
      - i.e. Space for static variables
    - Scheduler activations
      - Related to mapping of threads between user and kernel space

- Semantics Of 'fork()' & 'exec()'
  - The semantics of the 'fork()' and 'exec()' system calls change in a multithreaded program
  - If one thread in a program calls 'fork()', does the new process duplicate all threads, or is the new process single-threaded?
    - There is no clear answer to this, because sometimes we may

want to duplicate all threads, and sometimes we don't too
  - If a thread invokes the 'exec()' system call, the program specified in the parameter to 'exec()' will replace the entire process - including all threads
  - Using the two versions of 'fork()' depends on the application

- Signal Handling (1)
  - A signal is used in UNIX systems to notify a process that a particular event has occured
  - A signal handler is used to process signals
    - i.e.
      - A signal is generated by the occurence of a particular event
      - The signal is delievered to a process
        - Problem: Which thread inside a process is the signal for?
      - Once delievered, the signal must be handled either by default or user-defined signal handler
  - Every signal has a default handler that the kernel runs when handling signals
    - You can design your own signal handler and have it override the default signal handler

- Signal Handling (2)
  - Signals can be synchronous or asynchronous
    - Synchronous
      - Is delievered to the same process that performed the operation that caused the signal
        - This is why they are considered synchronous
      - Examples of synchronous signal handling:
        - Illegal memory access
        - Division by 0
    - Asynchronous
      - When a signal is generated by an event external to a running process, that process receives the signal asynchronously
      - Examples of asynchronous signals:
        - Terminating a process with specific keystrokes
          - i.e. CTRL + C
        - Having a timer expire

- Signal Handling (3)
  - For single threaded processes, the signal is delievered to the process
  - But, where should the signal be delievered for multi-threaded systems?
    - Four possible options are:
      - Deliver the signal to the thread to which the signal applies
      - Deliver the signal to every thread in the process

        - Deliver the signal to certain threads in the process
        - Assign a specific thread to receive all signals for the processes
            - This is a dedicated thread that handles signals
    - The method for delievering signals depends on the type of signal generated
        - i.e. CTRL + C
            - This should be sent to all threads because this key-stroke should terminate the process
        - i.e. ALT + F4
            - Sent to all processes and all threads because the application is being terminated
    - Tasks on single-threaded processes are quite complex on multi-threaded processes, and have their own challenges
    - Watchdog timers are used in embedded systems to terminate or reset non-responding threads or threads that take too long

- Thread Cancellation (1)
    - Terminating a thread before it has finished
    - The thread to be canceled is the target thread
    - There are two types of cancellations:
        1. Asynchronous cancellation
            - Terminates the target thread immediately
        2. Deferred cancellation
            - Allows the target thread to periodically check if it should be cancelled
                - This is the default type
    - Sample Code Snippet
        - i.e.

```
/* Create the thread */
pthread_create(&tid, 0, worker, NULL);

/* Cancel the thread */
pthread_cancel(tid);
// Note: This only requests for the cancellation of the
//       thread. It does not actually cancel it.

/* Wait for the thread to terminate */
pthread_join(tid, NULL);
```

- Thread Cancellation (2)
    - Invoking 'pthread_cancel()' indicates only a request to cancel the target thread
        - However, actual cancellation depends on how the target thread is setup to handle the request
    - When the target thread is finally canceled, the call to 'pthread_join()' in the canceling thread returns
    - A thread may set its cancellation state and type using an API
        - The mode of cancellation can be set (Deferred VS. Async)
            - i.e.

Mode	State	Type
Off	Disabled	-
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

    - The cancellation only occurs when the thread reaches the cancellation point, and then the cleanup handler is invoked
      - i.e. pthread_testcancel()

- Thread Local Storage (LTS)
    - Allows each thread to have its own copy of data
      - This data is dedicated to the thread
      - Useful if reading/writing to some memory is expensive
        - Local storage is a quicker option
    - Useful when you do not have control over the thread creation process
      - i.e. When using a thread pool
    - Different from local variables
      - Local variables are only visible during single function invocation
        - TLS is visible across function invocations
    - Similar to static data; it is unique to each thread

- Scheduler Activations
    - Both many-to-many and two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
    - Communication between user thread and kernel thread is typically done through an intermediate data structure called Lightweight process (LWP)
      - Lightweight processes (LWPs) are created by the kernel
        - If the kernel has 4 threads, then it will create 4 LWPs
          - Each LWP will be associated with a user thread
      - It appears to be a virtual processor on which a process can schedule a user thread to run
        - i.e. If you want to process 5 files, they can be done concurrently, even if you have a single processor system
    - If a kernel thread notices that a user thread will terminate unexpectedly, then it will cancel the user thread, and save the current state of the lightweight process, or create a new one that will run in a normal way
      - This is done through upcalls
    - Scheduler activations provide upcalls
      - This is a communication mechanism from the kernel to the upcall handler in the thread library

    - Due to this complexity, many-to-many and two-level models are not as common as one-to-one
        - The one-to-one model is used in most operating systems
            - i.e. Windows, Linux

- End
    - Operating Systems are among the most complex pieces of software ever developed!