Homework 1: Questions

1. (10 points) The central processing unit (CPU), also called a processor, receives a program's instructions; decodes those instructions, breaking them into individual parts; executes those instructions; and reports the results, writing them back into memory. The format for that processor comes in one of two primary types: vector and scalar.

    a. What is a vector processor?
        i. A vector processor uses a single instruction to perform n operations simultaneously– where n represents the length of the vector, which are single-dimensional arrays used for storing sequential members of the same data type.

    b. What is a scalar processor? How about superscalar processor?
        i. A scalar processor uses a single instruction to perform an operation on a single piece of data.
        ii. A superscalar processor uses multiple instructions to simultaneously process multiple pieces of data. This processor is a sort of crossover between a scalar processor and a vector processor.

    c. What are the differences between vector and scalar processor?
        i. Since scalar processors focus on a single instruction and piece of data at a time, versus vector processors which focus on an entire vector, scalar processors are typically considered more basic. Furthermore, since data is handled independently, scalar processors maintain efficiency on various data block sizes, whereas vector processors require large blocks of data susceptible to parallel computation in order to be efficient. However, in these cases, the simultaneous nature of vector processors performs less operations, which produces a lower computational cost than scalar processors. Furthermore, as these operations are conducted simultaneously, each result is independent of another, thus producing higher clock rates.

2. (10 points) Instruction-Level Parallelism (ILP) can be implemented in both hardware and software level to increase the number of instructions executed in parallel. Briefly describe the following ILP techniques:

    a. Instruction Pipelining
        i. The Instruction pipelining technique refers the process of using the processor to execute multiple instructions in parallel, while still maintaining the sequential nature of the code. This is completed by executing independent operations first, in parallel, storing necessary calculations in “pipeline registers”, then repeating this cycle for dependent operations with available results in the register. This process maintains code sequence while executing parallelism.

b. Speculative Execution
    i. The speculative execution technique attempts to predict future instructions to improve performance. Instead of waiting for a conditional jump to complete and instruction to appear, extra available resources are applied to potential tasks. Thus, when the instruction does appear, the task is already completed, wasting no additional time. While this may result in possible needless executions, the data is simply discarded.

c. Branch Prediction
    i. Speculative execution uses branch prediction to attempt to determine which 'branches' the code will execute in a conditional operation. The more accurate the branch prediction is, the less performance is wasted on unnecessary tasks. Without branch prediction, speculative execution would not exist, as each conditional would require resolution before the instruction could execute.

3. (10 points) Why parallel computing is critical to improve computer performance? In another words, what are the limitations by increasing transistor density in a single chip to improve its performance?

a. The main limit to increasing transistor density in a single chip is *power density*. While it is technically feasible to keep adding transistors, there is no possible way to dissipate the heat after a certain level. Parallel computing presents two possible power-saving solutions by adding additional cores. Core multiplication can be used to either increase performance while maintaining the same clock speed, or the performance can be maintained by reducing the clock speed. Both options limit the power density issue.

However, this brings up the second limit – we have hit a *parallelism limit*. Unless we can find new parallelism techniques, increasing transistors will not be possible.

Furthermore, as transistors are increasingly added to a single chip, the design becomes more complicated, thus complicating the manufacturing process. This would not only result in a *decrease in chip yield*, but also an *increase fabrication costs*. Parallelism can help with this by, instead of creating a single and complicated chip, multiple simple chips can be used. This decreases both complexity and manufacturing costs.

The last major limitation is in regard to physical limitations as a result of *the speed of light*. Consider a single chip CPU at 1Tflop (10^{12}), data would need to travel from the memory to the CPU at 1 Terabyte/ second – this is only possible if the entire terabyte of storage resides within a $\leq 0.3mm$ area. This would only be possible if each bit took up the space of a small atom. In other words, this is not currently possible. Physical limitations require parallelism to function after a certain point.

4. (10 points) What are the tradeoffs between preemptive scheduling and non-preemptive scheduling?

    a. Non-preemptive scheduling allocates the entire CPU to a single process and allows that process to continue until it terminates or gives back the CPU. While the running process holds the CPU, it cannot be interrupted. This remains true even if a higher priority process becomes available – this can result in lower priority processes running before higher priority. Another major disadvantage of this scheduling format is the possibility for data loops. If, for some reason, the process does not exit, or reach a waiting state, it will continue to hold the CPU indefinitely. Since the entire CPU is allocated to this process, there is no way to manually terminate it; to reobtain the CPU, the computer must be powered off/ restarted.

    b. Alternatively, preemptive scheduling allocates predetermined time-slices to processes. A running process may only hold the CPU until the current time-slice is up or a higher priority process becomes 'ready', at which point the OS switches to another process, whether that process is complete or not. Preemptive scheduling uses a waiting queue to maintain a list of processes that are ready to execute; if a process runs out of time but still requires completion, it is placed back in the queue. For this to be possible, a time-slice controller must be implemented, which increases complexity.

    While preemptive scheduling is more complicated than non-preemptive scheduling, it is the preferred method due to allowing processes to run 'simultaneously.' Instead of waiting for a task to complete before the next begins, the time-slice controller jumps between processes, creating the illusion that they are all running at the same time. Furthermore, if a higher priority process is introduced, it can be run immediately, instead of waiting for a low process to complete. This scheduling method can be seen in modern browsing and streaming services, where multiple processes are running at once (i.e the audio/video on YouTube). Lastly, if a deadlock or data loop transpires, it is possible to retrieve the CPU without having to shut down/ restart the computer.

5. (10 points) Two sample solutions for the dinning philosopher have been posted in the class web site: one solution is based on mutex and condition variable, the other solution is based on semaphore. Go through the two solutions and answer the following two questions:

    a. (5 points) Describe the difference of these two solutions.

        i. Mutex:

            - `main()`

                ❖ Program Begins

                1. Variable creation and general initialization, including initialization for semaphores with a value of 0 and mutexes for each philosopher
                    - `sem_init()`
                    - `pthread_cond_init()`
                    - `pthread_mutex_init()`

2. Using a for loop, a thread for each philosopher is created and verified. The starting routine for each thread is set to `philo()`
    - `pthread_create()`
    - `perror()`
3. User interaction via keyboard input is set up using an endless loop which continues until a 'q'/'Q' is entered. Program is initialized to accept the following input:

INPUT	ACTION
q Q	Quit
0-4b	Block Philosopher 0-4
0-4u	Unblock Philosopher 0-4
0-4p	Proceed Philosopher 0-4

- If 'q' or 'Q' are selected during the program, this handles the unblocking, joining, and destruction of threads.

❖ Program Ends

- `philo()`

❖ Main thread routine

1. Using a continuous while loop, the following series of functions are called:
    1. `checkForB()`
        1. If a keyboard input of 0-4b happened, the calling process is put to sleep
    2. `think()`
        1. Using a for loop, the philosopher enters a 'thinking' state for a predetermined period
        2. If, during this loop, there is a keyboard input of 'p', it is overwritten with a 'd', and the loop is broken prematurely
    3. `checkForB()`
    4. `get_sticks()`
        1. Mutex is locked
        2. Current philosopher states checked for errors, converted, and displayed

VALUE	DESCRIPTION
E	Eat
T	Think
H	Hungry
EXAMPLE	0H 1H 2E 3H 4E

        3. The current philosopher state is set to Hungry since he is attempting to get the sticks

4. A while loop continues as long as either the left or right stick is in use
    a. If either stick is in use, a conditional wait is entered until the stick is released and a signal is delivered
    b. Upon the return from the conditional wait, the loop is repeated to verify that the other stick is not in use and/or the current stick was not picked up by another waiting process
    c. The loop only exits upon both sticks being available
5. The current philosopher state is set to Eating
6. Both sticks are updated to in use
7. Mutex is unlocked

5. `eat()`
    1. Using a for loop, the philosopher enters an 'eating' state for a predetermined period
    2. If, during this loop, there is a keyboard input of 'p', it is overwritten with a 'd', and the loop is broken prematurely
6. `checkForB()`
7. `put_sticks()`
    1. Mutex is locked
    2. Current philosopher states checked for errors, converted, and displayed

VALUE	DESCRIPTION
E	Eat
T	Think
H	Hungry
EXAMPLE	0H 1H 2E 3H 4E

    3. Stick states are updated from in use to free
    4. Philosopher state is set to thinking
    5. Signals are dispatched to notify any conditional waits that the sticks are now available
    6. Mutex is unlocked
8. If a keyboard input of 'q' or 'Q' is entered, the loop exits. Otherwise, the loop begins again

ii. Semaphore

- `main()`
  - ❖ Program Begins
  1. Variable Creation and general initialization, including semaphores for chopsticks[5], screen, and a mutex for shared variables
     - `semaphore_create`
  2. Using a for loop, a thread for each philosopher is created and verified. The starting routine for each thread is set to `philosopher()`
     - `pthread_create()`
     - `stderr`
  3. Once the philosophers have finished eating, the threads are joined and the semaphores for the chopsticks[5], screen, and shared variables are released.
  4. The final results for total meals served and average hungry time are displayed.
  - ❖ Program Ends
- `philosophers()`
  1. Since this program uses a 'life' value to represent the time a philosopher sits at the table, the first thing this function does is record the starting time
  2. A while loop is entered which continues as long as the Current Time – Start Time is less than the decided duration
     1. The current time is recorded again to mark the beginning of when the philosopher became hungry (in order to calculate the final average hungry time later)
     2. An H is drawn on the screen in the philosophers location to represent 'Hungry'
        - `draw_hungry()`
        - a `semaphore_wait` is used on the screen during any attempted changes
     3. To obtain the chopsticks, the philosopher attempts to grab either the left or right chopsticks depending on their position (0,2,4 grab left first and 1 and 3 grab right first).
        1. A semaphore wait is used to check if the chopstick is available – if it is, the chopstick is shown as "picked up" on screen, otherwise the philosopher waits.

2. The grabbed chopstick is displayed on screen by removing the corresponding ‘stick’ next to the philosopher
3. After grabbing the first chopstick, the semaphore wait is conducted for the chopstick on the other side.

4. After the chopsticks are grabbed, the (Current Time – Became Hungry Time) is recorded (to calculate the final average hungry time later). Similarly, the total number of philosophers that have eaten is incremented by 1
5. The display is updated to represent the philosopher eating by placing an ‘E’ at their position
    - `draw_eating()`
    - a `semaphore_wait` is used on the screen during any attempted changes
6. The thread is temporarily suspended, for a random duration, to simulate eating time
7. The chopsticks are released by dispatching a signal to notify any waiting processes that the sticks are now available. This is displayed on screen by replacing the previously removed ‘stick’
8. After the sticks are released, the display is updated to represent the philosopher thinking by placing a ‘T’ at their position
    - `draw_thinking()`
    - a `semaphore_wait` is used on the screen during any attempted changes
9. The thread is temporarily suspended, for a random duration, to simulate thinking time
10. If the Current Time – Start Time remains less than the decided duration, the loop repeats. Otherwise, the while loops concludes.

3. The display is updated to represent the philosopher is done by placing a ‘D’ at their position
    - a `semaphore_wait` is used on the screen during any attempted changes in `draw_done()`
4. A `semaphore_wait` is used to access and update the shared variables, including `total_number_of_meals` and `total_time_spent_waiting`.
5. The thread is closed
    - `pthread_exit()`

b. (5 points) which solution is better? Why?

i. I believe both programs have their strengths and weaknesses and are relatively comparable. However, the mutex program (diningphilos.c) has one major flaw – there is a possibility of starvation. Therefore, I believe the semaphore program (philosopher.c) to be better. Both programs prevent deadlocks, albeit differently. The semaphore program using the modulus operator to have even philosophers pick up the left chopstick first, then the right, while the odd number philosophers pick up the right chopstick first, then the left. Alternatively, the mutex program only allows the philosopher to pick up a chopstick if both are free. While both prevent the deadlock scenario of each philosopher obtaining only one stick and waiting for the other, I believe the semaphore option to be superior. I hold this opinion because the second solution leaves a possibility of starvation. If the other threads are consistently faster than one, that thread may never see both sticks available.

Furthermore, the semaphore program has a specified 'run-time,' while the mutex program runs continuously. Once that time is up, the philosophers finish their loop and enter a 'Done' state. If a philosopher had somehow been starved out, he would have a chance to complete his loop and 'eat' before entering the 'done' state. As such, under the semaphore program, every philosopher will eat at least once. However, the alternation of sticks eliminates this possibility anyway. Also, the stick alternation keeps the number of times eaten balanced between the philosophers. Alas, the semaphore program is visually easier to follow and allows customization of philosopher 'lifetime.' However, this does come at the expense of a more complex code. Thankfully the author did a wonderful job using notes to guide a reader through the program.

If it were not for the starvation issue, the mutex program would be a strong contender. As mentioned, it offers deadlock prevention as well and is packaged much more simply. Furthermore, the mutex program incorporates keyboard input for program customization and the use of conditional variables. Although I am not a fan of the output format, I did appreciate the ability to interact with the program by blocking, unblocking, and proceeding philosophers. I think a combination of both programs would result in an efficient and educational solution of the philosophers dining problem.

	SEMAPHORE	**MUTEX**
PROS	Deadlock Prevention: Alternation	Deadlock Prevention: Availability
	Stop Time	Conditional Variables
	No starvation: Alternation, Program Completion	Keyboard Input/ Program Customization
	Visually Appealing	Simple
CONS	Complex	Possible Starvation

6. (Programming Assignment, 50 points) myhttpd1.cpp is a simple webserver written in C. The sample code has been posted in the class website. You can compile and run the program using the following commands in a Linux environment

```
> gcc myhttpd1 –o myhttpd1
> ./myhttpd1 –p 8080
```

After you run the program, open a browser and type `http://localhost:8080` and you will see "Welcome to my first page!" in the browser.

   a. (10 points) The posted program has a bug. A function call has been skipped on purpose in the program. Review the myhttpd1.cpp code, find the problem, and fix the code.

   b. (15 points) myhttpd1.cpp is a single thread program. Create a new program, myhttpd2.cpp, based on myhttpd1.cpp and use multiple threads to process http request.

   c. (15 points) Create a new program, myhttpd3.cpp, based on your modification in b). Change the program to daemon (running in background).

   d. (10 points) WeChat is a popular social networking application in China. Red envelope is an application in WeChat which can be used to send "lucky money" to friends. As estimated in 2017, a total of 14.2 billion red envelopes were exchanged via WeChat on New Year's Eve alone, peaking at midnight with 760,000 transactions per second. It usually takes seconds for WeChat to process each transaction. For such high-volume concurrent transaction requests, what remediations could be used on the server side to ensure each request is served promptly?
      i. Some remediation examples that WeChat could take on their server to ensure promptness are implementing multithreading using conditional variables, critical sections, and semaphores/mutexes, increase system reliability and performance with multiprocessing, ensuring software remains updated, fortify security of their code and communications to protect against unnecessary/unexpected downtime, enable a caching database, utilize shared memory machines, and maintain healthy/updated hardware. Furthermore, WeChat could implement instruction pipelining to execute instructions in parallel, speculative execution/branch prediction to save time during conditional jumps in code, a superscalar processor to process multiple pieces of data simultaneously, and lastly, ensure that an efficient scheduling algorithm is being used. Essentially, maintenance of hardware/software, ensuring security, and utilizing multiprocessing/ multithreading are necessary to provide a prompt server response at a large scale.