

1 (EC)

a. Does the number of philosophers impact your solution in any way? How about if an even number of forks are used?

Since my solution has only one philosopher eating at a time, as the number of philosophers increases, the time it takes for every philosopher to eat also increases. An even number of forks doesn't affect the solution since each philosopher will only eat when it is their turn.

In a more optimal solution where multiple philosophers eat at the same time, (for example, every other philosopher eats except the last one who eats alone in a third iteration), the extra fork would allow him to eat with the second group, thereby allowing each philosopher to eat after no more than 1 iteration after the last group to eat.

b. What happens to your solution if we give one philosopher priority over the rest?

In my solution, we already give a single philosopher priority over the rest. The priority such that only one philosopher is permitted to eat at a given point in time. This means that every time step, at least one philosopher is always able to eat!

c. What happens to your solution if we add a fork in the middle of the table?

Nothing -> the extra fork would go un-utilized because each philosopher attempting to eat would already have the ability to eat because he has both forks necessary.

d. Also, discuss who was Edgar Dijkstra, and what is so important about this dining problem, as it relates to the real world. Also discuss the algorithm that bears his name, Dijkstra's Algorithm. Make sure to cite your sources carefully.

The dining philosopher's problem is a generalization of a problem for competition over shared resources. The forks represent the resources that need to be shared and the diners the things attempting to use those resources. The solution describes how multiple contenders can cooperatively use a shared resource without ending up in a state of deadlock where both are waiting on the resources currently in use by the other.

Dijkstra was one of the early computer pioneers who built and defined the field as we know it. In particular he created the idea of "structured programming" which has served as the basis for today's programming methodologies.

Dijkstra's Algorithm is an algorithm for finding the shortest path between two nodes in a graph. The original algorithm has since been generalized to create a

minimum spanning tree to reach every node in a graph from a given starting node.

Source: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

2

Evaluate the speedup that you achieve by using pthreads and multiple cores. You are free to use as many threads as you like.

Make sure to document the system you are running on and the number of hardware threads available

System: Iota@Coe

- 24 threads
- Intel(R) Xeon(R) CPU X5650 @ 2.67GHz
- 2 threads/core
- 6 cores/socket
- 2 sockets

Finding primes to 1 million

Time to Find

1 thread	2 thread	4 thread	8 thread	16 thread
838ms	442 ms	249 ms	145ms	141ms

Scaling	1 thread	2 thread	4 thread	8 thread	16 thread
Strong	100%	94.79%	84.13%	72.24%	36.88%
Weak	100%	189%	337%	577%	590%

3 (EC)

In class we have learned about pthreads. These are also referred to as Posix threads. For the first part of this problem, discuss the differences between pthreads and processes. Pthreads operate out of shared memory and exist from [one to many) within a single process. When processes are forked, all of the memory must be copied over into the new context which is independent of the old processes. This differs from newly spawned threads who all refer to the existing virtual memory space within the parent processes.

Second pthreads are considered “operating system-level” threads, versus green threads are considered “program-level” threads. OS level threads are threads that are running on an OS level meaning they can have access to different pieces of hardware (cores). In order to use OS level threads, your OS must provide some sort of interface for you to use.

Program level threads operate at an application layer level (i.e. in user space). This means that they can't automatically be split up amongst available hardware. Instead they are scheduled within, and managed by the user's application. They are often used to give "multithreading" capabilities to software being developed on systems without support for OS level threads.

Also, there are hardware threads. Describe the differences/similarities between these 3 forms of threading and provide an example of where they have been implemented in a hardware or software system. (This question is required for graduate students, but extra credit (on the quiz average) for undergraduates.) Each level of threading gets progressively lower towards the hardware.

Green threads operate at the highest level, completely abstracted from the hardware. These threads run out of user space and are essentially "virtual" threads created by an application. Historically, Java has used green threads to offer developers access to threading even though the JVM doesn't have actual "cores" to give.

Software/OS threads are offered by the operating system and can be placed on different cores controlled by the operating system. An example of the OS level threads is the pthreads library which gives developers access to create, manipulate, and destroy threads with a given set of library calls.

Hardware threads are a processor feature which "tricks" the OS into thinking that a single core is actually two cores by allowing two "virtual" cores to share the underlying hardware provided by a single core. This takes advantage of latencies along the processing pipeline that allow multiple sets of instructions to be run at the same time while continuing to utilize resources during those latencies. This technology was developed by Intel and is used across their processors. For example, Iota in the COE clusters has hyperthreading enabled which turns the 12 physical cores on the system 24 CPUs seen by the OS.