ADJ Assignment 1
SER 334: Operating Systems

Swanson, Douglas
dswanso6@asu.edu

Speidel, Clay
cspeidel@asu.edu

November 27, 2019

1 Question 1: Operating System Structures

1.1 Problem (P)

[Acuña, Lisonbee] Different operating system structures offer both benefits and drawbacks over one another, and it's important to understand what kind of structure should be used for different use-cases. Consider the following situation. You are designing an operating system for an interstellar probe that is meant to run for hundreds of years. The probe should also support monitoring many specific instruments.

What structure (simple, layered, microkernel, or modular) should you choose for its kernel?

1.2 Analysis (A)

An interstellar probe would have some of the following requirements on it's operating system kernel. The operating system would need to be robust, self reliant, memory efficient, power efficient, and support various sensors and subsystem equipment. It should also allow the programmers to develop the kernel almost independently from the spacecraft and allow for extensive testing.

The kernel will need to be robust means that it would handle its own errors and anomlous conditions in a manner that would allow it to continue operating normally thus keeping the satellite alive for the entirety of its mission. This should handle things like subsystem failures. This requirement compliments the self reliant requirement becuase to be self reliant it should be able to make decisions, that are in the best interest of the spacecraft's mission, in solving the errors and anomlous conditions that may appear over it's lifetime.

This kernel should be both memory and power efficient because a spacecraft has very limited tolerances for both of these resources. The memory should be conserved because the length of the mission is so very long, hundreds of years, thus the limited memory of the system would need to last. The power is another finite resource which will, over the lifetime of the mission, become reduced, thus the kernel should require minimal power.

The spacecraft will have multiple subsystems on it, which will range from core function such as temperature, attitude, and navigational control to more mission specific functions such as data logging and payload control. The kernel will need to allow for the addition of these modules as well as handle their health and data.

The requirements stated above become the following technical requirements:

- Handle all of the iteractions between the software and the hardware.
- Allow for scheduling of tasks over long periods of time.
- Control interactions between subsystems, not allowing one subsystem to diretly control another.
- Allow for new or different subsystems to be added during development.

The first requirement would allow for the kernel to have direct and exclusive control over the hardware which would promote the robustness and self reliance of the design because it would only allow processes that have been approved and tested to act upon the hardware. This reduces the risk of anomlous hardware condtions, only our processes will be used, as well as promotes easy testing because only the prescribed processes would need to be tested.

The second requirement would assist with the memory and power requirement as well as supporting inter subsystem timed communications. This would allow the system to to toggle subsystems on and off when they are required to do something thus reducing the power and memory requirements to the bare minimum at any given time.

The third requirement allows for the operating system on the spacecraft to have complete control over all of the subsystems onboard which will allow it to have complete information of all of the messages generated on board as well as the state of each of the subsystems. This gives the operating system the ability to make informed decisions to maintain the health and sucess of the mission.

The final requirement decreases the development time of the operating system which will decrease the overall labor cost on the spacecraft. This also decreases both the risk of failure of the operating system as well as the amount fo time spent testing. The testing is reduced because everytime a subsystem on the spacecraft is added, removed, or changed the entirety of the operating system doesn't need to be retested which in turn reduces the changes to the system and the percentage of human errors appearing in the code.

1.3 Design (D)

There are two kernel structure that was choosen was the microkernel. This structure satisfies the requirements listed above in the following ways. The microkernel, by definition, handles all communication between the plugins attached to its core. By controlling the communication between the subsystem plugins the core can then authenticate the requests between subsystems before passing them along which would allow it to catch bad requests before they are sent to the subsystem, thus reducing the risk of a subsystem breaking another subsystem. This will also allow the core to track all of the communications on the spacecraft which could be useful should the system enter an anomlous state that requires debugging. This core of the system will aslo handle accessing the hardware and scheduling of tasks. Since the micro kernel core can monitor all of the modules on board, it can then make informed decisions and safe guard the rest of the spacecraft in emergency situations. The microkernel would have an API that would allow the team to pick and choose their sensor and subsystems through the development process and write moduluar code that interacts with the core in a predefined manner, thus making testing of modules and the core easier. Also allows for flexibility throughout the development cycle without having to redesign the core kernel.

1.4 Justification (J)

The design stated above out performs the other operating system structures and is the optimal solution. Take a look at the simple structure, it would allow for everything on the spacecraft to directly access the spacecraft hardware which would be very bad if one of the subsystems failed for some reason and was commanding the spacecraft hardware to do things based upon erroneous data that could cause the mission to fail. The layered approach would solve this issue but it could be harder to maintain throughout the development of the spacecraft, harder to test, because when a new, or different, subsystem is added to the spacecraft it would have to be integrated into a layer in the kernel. Everytime we change a layer the entire kernel would have to be retested. Finally the modular structure would bring all of the benefits of the microkernel but would allow the subsystems to talk directly to each other. This cross communication would be a benefit on a system that was being actively monitored and accessed by a human, but on our spacecraft we would want our core to have complete control and knowledge of the entire system so that it can perform emergency actions. The cross communication also introduces the risk of a lazy developer bypassing the core and having their subsystem directly command another, which becomes a problem should their subsystem break and start commanding another subsystem to do things based upon erroneous data. So in the end the microkernel process would be the best structure to use.

2 Question 2: Processes

2.1 Problem (P)

[Acuña] Consider parallelizing the insertion and selection sort algorithms. Which would be more amenable to parallelism?

2.2 Analysis (A)

Insertion sort and selection sort are two types of sorting algorithms, typically used by programmers to sort arrays of elements. Insertion sort compares two elements in an array that are next to each other and places the value that is less than to the left. This comparison is done within a loop and the loop exits as soon as the value being sorted is greater than an element to the left of it. The algorithm will proceed through the entire array until it is completely sorted from a minimum value to a maximum value.

Selection sort is similar in that it compares elements within an array. However, selection sort will search through the entire array to find the minimum value using a loop, then place the minimum value to the front of the array and move everything else down. Once the selection sort algorithm has placed the minimum value down it increments the loop, thus separating the array into a sorted and an unsorted section. The algorithm will search through the array again to find a new minimum value and then place that into the next position in the array, and so on.

Insertion sort and selection sort algorithms both have an average case of O(n2) for comparisons, while insertion sort has an average case of O(n2) for swaps and selection sort has O(n) for swaps. This means that when sorting an array, the amount of time taken to compare values is a quadratic function of the number of elements within the array and for a very large array, these sorting algorithms can take a very long time to complete. For swapping, it is quadratic for insertion sort and linear for selection sort. We begin to look for a solution to this conundrum: how can we make these sorting algorithms faster?

One option is parallelism, the splitting up of a process into smaller parts in order to have separate processors working on the same problem at the same time. This act is sometimes referred to as “dividing and conquering”. In terms of the insertion sort and selection sort algorithms, it should be possible to divide the array that needs to be sorted into equal parts and have different processes sort the divided parts. However, the question “which would be more amenable to parallelism” remains.

2.3 Design (D)

The answer is not so simple: we can divide both of these algorithms into smaller parts and sort each of the divided parts. The design for the solution is as follows: we can divide the arrays into the number of processors we wish to use to work on the sorting algorithm; we can sort the arrays using the algorithms; we can join the divided parts together again and sort the final array.

As stated before in the analysis, both algorithms have an average case of O(n2) for comparisons, but in terms of swapping, selection sort has a linear function of time while insertion sort has a quadratic function of time. This leads us to believe that on average, the selection sort will be faster. However, the best case for insertion sort is O(n) for comparisons and O (1) for swapping. O(1) is a constant value, and is considered to be the fastest speed possible of sorting algorithms. This best case for insertion sort occurs when the array is already sorted, because there does not need to be any swapping at all.

For selection sort, there is no ideal situation. No matter how the array is sorted in the beginning, the time for comparisons will always be O(n2) and will always be O(n) for swapping. This is because selection sort goes through the entire array to find the minimum value, no matter what order the array is in.

This leads us to parallelism and what parallelism can do for our sorting algorithms. Breaking down the array that we need to sort into smaller parts will allow more processors to work on sorting. If we run both sorting algorithms on the smaller parts, the average values for comparisons and swapping will remain the same: selection sort will have an edge on swapping with its linear value.

When we finish sorting the smaller parts, putting the parts together becomes the deciding factor. Since the smaller parts are already sorted, when sorting the smaller parts into one large array, it will be easier for insertion sort. Selection sort will still have to work through the entire array for each element in the array. However, insertion sort will have an easier time since the array is already partially sorted. Insertion sort will very nearly have its best case of O(n) for comparisons and O(1) for swapping.

2.4 Justification (J)

Ultimately, dividing the work among the processors will speed up both sorting algorithms. However, one type of sorting algorithm is more amenable because it has a special trait that gives it an advantage against other sorting algorithms. Insertion sort does not have to run through the entire array before moving an element in the array. In the analysis, it is stated that insertion sort only compares an element to its neighbor, while selection sort compares all elements until it finds the minimum value. While dividing up an array and using selection sort is an option, insertion sort is preferable because it will take less time when putting the smaller portions of the array back together.

3 Question 3: Threads

3.1 Problem (P)

[Acuña] Consider the algorithmic task of compressing a video file for a cartoon. Initially, the video is a stream of separate images. The images are large, and in many places, differ only slightly from frame to frame. For instance, when an object is moving, the part of the image not involving the object does not change from one frame to the next.

Say we want to design a compression mechanism based on determining the difference from the previous frame to the next frame, and saving only the difference into a compressed result. Of the five issues in multicore programming, which is the most problematic for multithreading this system?

3.2 Analysis (A)

The five issues of multicore programming are: Balance, Data Splitting, Data Dependency, and Testing and Debugging. Balance is breaking the task into equal size chunks so that each thread has a similar amount of work to complete. Data splitting is figuring out how to break up the data up so that each thread gets the data that it requires to complete the task at hand. On to data dependency which is the association between the data in question that can make splitting the data up evenly because some data requires other data. Finally the testig and debugging is how each thread will be tested in order to ensure that it is producing the correct response and if it isn't then how will the threads be debuged in order to find out why it is producing an anomlous result. The program will determine how to split the data in a balanced manner in such a way that can be easily tested and debugged without breaking any data dependencies.

A video file is a series of similar images with minor changes between them, as stated in the problem. The threading system needs to beable to determine the similar images and store only the portions that are different between them. The similar portion of the images will be stored only once to reduce the amount of memory used, thus compressing the file. The threads should be assigned a series of similar images and operate upon them to find the similarities. Once the video is decompressed the changes will be applied to the similar image to create the series of images once again.

Multicore programming is difficult and multithreading is problematic, but with this analysis we will determine the most problematic of the five issues stated.

3.3 Design (D)

The first step is identifying tasks for the multithreaded program, which is relatively easy. In this instance of the video file, the video is broken up into frames and each frame is broken into pixels. We could break the frame into a 2D array of pixels and compare the pixels of one frame to the next, identifying any changes found. That is a lot of work for a program to do on its own and would take a lot of time, which leads to balancing.

For balancing, we could break the array of pixels into layers and have different threads work on an equal portion of layers. This way, each processor working on the task will do a comparable amount of work. This cuts down on the time needed for the program to work, relative to the number of processors on the system being used.

For data splitting, since we are looking only for the data that is changing with each frame in the video, we can remove all copies of the data from memory. Memory is a finite resource, and so each thread need only save the data that is changing. This will compress the file greatly, saving time during uploading and downloading.

In terms of data dependency, there are some things that cannot be parallelized. In this situation, we are not splitting the video into a section of frames and having each processor work on a section of frames because it would be too costly, in terms of memory, to create that many large separate arrays in order to parallelize, when we could create a single array that is broken up into parts and worked on by multiple processors. While this method may take more time, since it is working on one frame, it will be less costly in terms of memory.

Finally, testing and debugging, which is always troublesome. However, with multithreading, it becomes much more difficult, as it is more difficult to pinpoint an issue with many threads woven together. This will take the most time among the issues, as testing is the number one reason for cost and timing problems with a project. For this instance, we would need to test many different video files and when there are issues, it will be difficult to determine where they are occurring without properly splitting up and identifying the threads being used.

3.4 Justification (J)

Designing a choice requires a form of measurement, and in this instance we will determine the most problematic of the multicore programming issues using the time needed to solve these issues. We will use time because of the importance of deadlines to companies and clients and because time is money for all involved.

Identifying the tasks, balancing, data splitting and data dependency are all relatively quick to work out. With the popularity of Agile programming, these things can be determined on the fly, during coding. However, testing and debugging will be the most problematic and time-consuming of the five issues and will cause the most headaches.