CIS 415 Dylan Secreast
Prof. Sventek 4/14/16
Assignment 1

1. OSC 2.18: What are the two models of interprocess communication? What are the strengths and weaknesses of the two approaches?

The two models of interprocess communication are the message-passing model and the shared-memory model. Out of the two methods, message-passing communication is easier to implement than shared memory and is also most useful for for exchanging smaller amounts of data due to not having to worry about conflicts. However, message-passing comes with an overhead from copying the message's existing argument into a portion of the new message and depending on the argument, can be additional megabytes worth of data.

Since shared-memory communication can be done at memory transfer speeds, this method allows for the maximum speed and convenience of communication. However, shared-memory poses problems in protection and synchronization between processes sharing memory due to the operating system attempting to prevent one process from accessing another process's memory.

2. OSC 2.19: Why is the separation of mechanism and policy desirable?

The separation of mechanism and policy is desirable for the importance of proper resource allocation and to maintain flexibility. Policy changes are made often at the kernel level which require a change in its underlying mechanism. Consider the timer construct, a mechanism for ensuring CPU protection whose cycle length is determined by its policy. Without a separation of mechanism and policy for this process, the CPU poses unintended behavior.

3. OSC 3.9: Describe the actions taken by a kernel to context-switch between processes?

When a CPU receives an operation to switch context between processes, it first needs to save the current state of the process running to the kernel so that it can be continued once the process being switched to has been completed. After the state save has occurred, the saved context of the new process that has been scheduled to run is loaded.

4. OSC 3.18: What are the benefits and the disadvantages of each of the following? Consider both the system level and the programmer level.

a. Synchronous and asynchronous communication

Synchronous communication has the benefit of a rendezvous, that is, both send() and receive() are blocking. However, with asynchronous communication, a rendezvous may not be required due to a blocking send() and the message itself can arrive at a time too late to the sender. Due to the properties of the rendezvous, both methods of communication are typically used.

b. Automatic and explicit buffering

Automatic buffering uses a queue with potentially infinite length, only bound by the amount of disk storage. A queue of unbounded length has the benefit of having the sender never needing to block while waiting for a message. However, with automatic buffering, there may be unused memory that's wasted. This isn't the issue with explicit buffering where the buffer size is predetermined for the exact message. Even though the sender may be blocked while waiting for available space in the queue, less memory will be wasted with explicit buffering than automatic buffering.

c. Send by copy and send by reference

Send by reference allows for the receiver to alter the message if necessary whereas send by copy does not. Sending by copy allows for quick and easy manipulation of the message but poses the risk of malforming data when used without care. Whereas, send by copy does not allow for the original message to be altered but still has the capability to create a local copy to then be independently altered.

d. Fixed-sizes and variable-sized messages

Fixed-sized messages allow for a specific predetermined buffer size to minimize memory overhead where variable-sized messages do not. Due to the nature of variable-sized messages, the number of such messages that can be held by a buffer is unknown and stored in shared memory.

5. OSC 4.14: A system with two dual-core processors has four processors available for scheduling. A CPU-intensive application is running on this system. All input is performed at program start-up, when a single file must be opened. Similarly, all output is performed just before the program terminates, when the program results must be written to a single file. Between start-up and termination, the program is entirely CPU-bound. Your task is to improve the performance of this application by multithreading it. The application runs on a system that uses the one-to-one threading model (each user thread maps to a kernel thread).

a. How many threads will you create to perform the input and output? Explain.

I would not create any additional threads for assisting either input or output beyond the one thread dedicated to completing the operation. Adding more than one thread will not provide any further benefit due to the nature of reading and writing to a single file. If you were to add additional threads, they would just end up waiting for the I/O to be completed and could have otherwise been used towards the CPU-intensive portion of the application.

b. How many threads will you create for the CPU-intensive portion of the application? Explain.

I would create 4 threads to be executed on each of the 4 available cores to accomplish parallel threading and maximize the CPU's available processing power towards the application. Any number of threads less than the number of available cores would be considered a waste of processing resources.

6. OSC 4.18: Consider a multicore system and a multithreaded program written using the many-to-many threading model. Let the number of user-level threads in the program be greater than the number of processing cores in the system. Discuss the performance implications of the following scenarios.

a. The number of kernel threads allocated to the program is less than the number of processing cores.

If there is a fewer amount of threads than cores, then there will be unused processing resources available. A running program could potentially take advantage of the unused cores and additional processing power by creating addition threads to be used.

b. The number of kernel threads allocated to the program is equal to the number of processing cores.

If the number of threads is equal to the number of cores, then it is possible that a program or process may make full use of all available processing resources. However, if all threads were to be used across all cores and if a single thread were to become blocked, it's respective core would remain idle until the process became unblocked.

c. The number of kernel threads allocated to the program is greater than the number of processing cores but is still less than the number of user-level threads.

If there are more threads than available cores, then a blocked thread could potentially be exchanged with another thread that is ready to be executed, fully maximizing the maximal processing potential of a CPU.

Process Analysis

To determine information regarding a running process in which there is no access to source code, consider the following:

touch ~/Desktop/report.txt
Create a blank txt file that will hold all of the process's information.

sleep 600&
Start an example process to example, note the unique PID it returns.

cd /proc/UNIQUE_PID
Change directory to the kernel's virtual filesystem that contains all relevant information to the sleep process.

sudo cat io stack sched status limits > ~/Desktop/report.txt
Copy all of the process's available human-readable information to the report text file.

- io: Information regarding any file input and/or output the process is currently handling.

- stack: A stack of threads and their respective memory address that are associated with the process.
- sched: A list of the process's schedule, including execute start time, wait time, sleep/wait time, wakeups, load weight, and average runnable sum/period.
- status: A list of information regarding the process's active thread, including: name, state, PID, FDSize, CPU's allowed, and memory allowed.
- limits: A table of soft and hard limits for various properties such as: CPU time, file size, data size, stack size, open files, address space, and priority.

With all of the information generated into the report.txt file, you can deduct a much better idea as to the methodology used to implement the reported process. All of the human-readable information listed above describes how the process is handled by the operating system to streamline multiprocessing functionality. Note that the 4th command listed above applies to any process in which the PID is known.