Stephen Woodbury - 1429496 - swoodbur
HW1 - CMPS 111 - Spring 2018 - 4/18/2018 - Wednesday

1) *Briefly outline the evolution of Operating Systems from those used by the earliest stored program computers of the 1940s to their modern counterparts.*
A:
Batching Systems→Spooled Batching→Multi-Programmed Batching→Time Sharing Systems.
Batching Systems: Simple control programs that worked with jobs in a batch. Jobs in the same batch had similar run-time contexts as switching run-time contexts was tedious, expensive, slow, and prone to error. A job in a batch would be worked through one at a time. This means that before the next job can start, the first job must finish reading, computing, and printing. Given the I/O was slow and that the CPU was faster than the I/O, the CPU was under utilized in batching systems.
Spooled Batching: It's similar to CPU instruction pipe-lining, requiring hardware interrupts. The OS was still a simple control program, but instead of waiting for each job in a batch to finish (as in Batching systems), spooled batching allowed jobs to be ran closer together. A job has three parts roughly, reading, computing, and outputting. In Spooled Batching, job one could finish reading and start computing, allowing for job two to start being read in. The I/O was still slow, and the CPU was still under-utilized, but it is less under-utilized in Spooled Batching than in Batching Systems.
Multi-Programmed Batching: The start of sophisticated Operating Systems, Multi-Programmed-Batching had several jobs resident in memory at the same time, all sharing the same CPU. The OS would decide which job was running on the CPU (Job Scheduling), it had to protect and manage memory, and it had to ensure fairness between jobs(CPU scheduling). If one job was blocked due to I/O needed, the CPU would schedule a different job in the batch to run on the CPU. Here, CPU utilization was fully utilized if there was at least one job at all times that was ready to run on the CPU. IBM's 7030 was a noteworthy computer developed when Multi-Programmed Batching was prevalent. It pioneered many advanced Computing techniques such as memory protection, interrupts, and even multi-programming.
Time Sharing Systems: User interaction came into play here. The OS would allow for switching between user programs frequently so that users can interact with a program. If n people wanted to access the CPU, 1/n computer time would be allocated to each user as the CPU would switch between them all. OS complexity changed drastically with time sharing was wasn't fully implemented until the CDC 7600, made by Cray.
EXTRA: Personal Computers came out in the 70's and initially didn't multi-task like the bigger more powerful OS's. As power supply increased however, personal computers eventually came to adopt many of the abilities of bigger OS's. OS's were also perfected early on by Thomson and Ritchie. Modern day OS's are roughly the same as they were in in the 70's after the successful implementation of Time Sharing.

2) *In the following piece of C code, how many processes are created when it is executed? Explain your answer.*

```
1 int main() {
2  fork();
3  fork();
4  fork();
5  exit(1);
6}
```

A: fork() creates a near duplicate of the parent process that refers to the same code as the parent. Once the child process is created, both the parent process and the child process will start executing the next line following fork(). At line 1, there is only one process running. At line 2, fork() is called, creating two identical processes distributed over two levels. Level one being the parent process, level two being the child process. Following the successful creation of the child process at line two, both the parent process and the child process start running from line 3, which is a fork(). This means there will be four processes in total after line 3 has finished. After line 4, there will be eight processes. Afterwards, there are no longer any more processes created. The above C code creates a total of eight processes distributed over four levels. Level one contains one process, the original process, the parent. Level two contains three processes, all children of the process in level one. Level three contains two processes, children of processes on level two. Level four contains one process, child of a process in level three.

3) *If an Operating System assigns an unsigned 32bit integer to store current time as the number of seconds elapsed since 00:00 on January 1 1970, is this likely to be a problem? Explain your answer.*

A: An unsigned 32bit integer can hold any number between 0 and 4,294,967,295. There are 31,540,000 seconds in a year. If we do the math, we can account for 136.175 years worth of seconds. This means that sometime in the year 2106, our unsigned integer will not be able to hold the number of seconds required to keep the time in terms of seconds since 00:00 January 1st 1970. The latest date that our unsigned int can hold is February 7th, 2106 at 6:28am 15 seconds into the minute. Any second further than that though will break our time keeping system. Is this likely to be a problem for anyone alive in 1970 or even for someone in CMPS 111 right now? No, we'll likely be dead before that time. It's also likely that the system that this time is kept for, if it hasn't died or been broken already, will become redundant long before 2106 considering the pace of technological development.

Stephen Woodbury - 1429496 - swoodbur
HW1 - CMPS 111 - Spring 2018 - 4/18/2018 - Wednesday

4) *Describe how a web server might leverage multi-threading to improve performance. Include diagrams if you feel this will make your answer clearer.*
A: Multi-threading means using multiple threads within a process to run a program or even our operating system. There are different classifications for threads. Kernel-level threads, user-level threads, kernel threads, and user threads. A web server would likely involve themselves with Kernel-level threads and User-level threads. A web server could solely rely on user-level threads for its operations, which is equivalent to a one to many model where there is one kernel-level thread handling all user-level threads. This would be impractical however as if one user-level thread blocked, all other user-level threads would be blocked as the handling kernel-level thread would block. Another method for a web server would be to map every user-level thread to a kernel-level thread. This would ensure that if one user-level thread blocks, the other user-level threads won't block. This is expensive to maintain however and when a user-level thread is done being used, the kernel-level thread as well as itself are tossed away and wasted. This method is also impractical. The most efficient way to manage a web server is to use a many to many model when mapping kernel-level threads to user-level threads as well as to maintain a thread pool. The many to many model maps a certain number of user-level threads (n) to a certain number of kernel-level threads (m). Usually “n” is greater than “m”. This helps mitigate how many threads are blocked due to another user-level thread being blocked while keeping the cost of thread creation/maintenance down. Another way our way of managing a web-server keeps the cost of thread creation/maintenance down is through thread pools. These keep an active number of threads up and running at all times so that threads can be recycled back into the pool upon completion rather than being destroyed and wasted. Thread pools are especially good for burst like activity on web-servers as they can decrease the number of active threads in the thread pool when activity is low as well as they can raise the number of active threads in the thread pool when activity is high.

Stephen Woodbury - 1429496 - swoodbur
HW1 - CMPS 111 - Spring 2018 - 4/18/2018 - Wednesday

5) *(a) In a multiprogrammed environment with 16MB of memory where all processes require 1MB of unshared memory and spend 60% of their time in I/O wait, calculate how much memory will remain unused when approximately 99% CPU utilization is achieved. (b) In the same multiprogrammed environment, if each process now requires 3MB of unshared memory, calculate the maximum achievable CPU utilization. Show all your work.*

A:

a) Given we know the formula U = 1-p^n where U is the CPU utilization, p is the percentage of time spent in I/O on average for each process, and n is the number of processes running on our CPU, we can find the amount of memory unused given the information provided in the question.

First, find the number of processes running using our formula of U.

1) .99 = 1-(.60)^n.

2) .01 = (.60)^n

3) log (base=.60) .01 = n

4) 9.015 = n, ie, our number of processes running in the system is around 9.015.

Since our CPU utilization was approximate, we can approximate the number of processes being ran to 9. We know that each process takes 1MB of unshared memory, meaning that 9MB of unshared memory is needed to run our 9 processed. This leaves 7MB of memory unused in our system.

b) Given the same multi-program environment specified in the question and with the change of each process needing 3MB of unshared memory(instead of 1MB each), we can determine what our new CPU utilization is using the same formula from part a. U = 1-p^n.

With each process needing 3MB of memory, we can only run 5 processes with 16MB of memory accessible to us. Fill in our CPU utilization formula:

1)U = 1-(.60)^5

2)U = 1-.07776

3)U = .92224

This means that roughly 92.2% utilization of our CPU is the maximum CPU utilization we can achieve given our restrictions and programming environment.