Ahmed Alelg
201507470

Homework - 3

ICS-431:Operating Systems Term: 192
Due Date: 21nd March, 2020

Problem 4.1
Provide two programming examples in which multi-threading does not provide better performance than a single-threaded solution.

Solution: Any program that contains modules that must run consecutively may not achieve better performance if implemented multi-threaded. Consider for example a program which converts temperature from Fahrenheit to Celsius, or a program which calculates the average of exam grades.

Problem 4.3
Which of the following components of program state are shared acrossthreads in a multithreaded process?

Solution:

-	Shared ?
Register values	No
Heap memory	Yes
Global variables	Yes
Stack memory	No

Problem 4.5
In Chapter 3, we discussed Googles Chrome browser and its practice of opening each new website in a separate process. Would the same benefits have been achieved if instead Chrome had been designed to open each new website in a separate thread? Explain.

Solution: If each website was to run in new thread instead of new process, then this will increase the utilization since threads share major parts among them, and it would be less secure. If a thread is crashed for some reason, then all other websites will be affected.

Problem 4.6
Is it possible to have concurrency but not parallelism? Explain

Solution: Yes, a multi-processing OS running on a single processor is an example of that. Interleaving between CPU and I/O bound processes allows concurrency, parallelism is achieved when two or more processes running at the same time but on different processors.

Problem 4.9
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 startup 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). How many threads will you create to perform the input and output? Explain. How many threads will you create for the CPU-intensive portion of the application? Explain

Solution: One thread for input and another for output; because reading or writing are I/O bound processes and might be blocked until the resources is released.
Because we have four processors, to increase utilization and performance of the application we must create 4 threads, if more were created they will be queued at the ready queue.

Problem 4.9
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 startup 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). How many threads will you create to perform the input and output? Explain. How many threads will you create for the CPU-intensive portion of the application? Explain

Solution: One thread for input and another for output; because reading or writing are I/O bound processes and might be blocked until the resources is released.
Because we have four processors, to increase utilization and performance of the application we must create 4 threads, if more were created they will be queued at the ready queue.

Problem 5.5
Consider the exponential average formula used to predict the length of the next CPU burst. What are the implications of assigning the following values to the parameters used by the algorithm? a. $\alpha = 0$ and $\tau_0 = 100$ milliseconds, b. $\alpha = 0.99$ and $\tau_0 = 10$ milliseconds.

Solution:

a. $\tau_{n+1} = 0t_n + (1-0)(100) = 100ms$, Hence, the next prediction will depend entirely on the previous prediction and will discard the recent time burst, i.e the time will be constant $100ms$ for all predictions.

b. $\tau_{n+1} = 0.99t_n + (1-0.99)(10) = 0.99t_n + 0.1ms$, Hence, the next iteration will depend significantly on the recent time, with little to no consideration to past history.

Problem 5.10
Which of the following scheduling algorithms could result in starvation? a. First-come, first-served b. Shortest job first c. Round robin d. Priority

Solution: Shortest job first and Priority

Problem 5.12
Consider a system running ten I/O-bound tasks and one CPU-bound task. Assume that the I/O-bound tasks issue and I/O operation once for every millisecond of CPU computing and that each I/O operation takes 10 milliseconds to complete. Also assume that the context-switching overhead is 0.1 millisecond and that all processes are long running tasks Describe the CPU utilization for round-robin scheduler when: a. The time quantum is 1 millisecond b. The time quantum is 10 milliseconds

Solution:

a. Each I/O-bound will issue one I/O operation which takes $1ms$ during the quantum time of $1ms$ and then will be taken away from the cpu ($0.1ms$) to spend $10ms$ in the I/O queue before returning to the ready queue. the waiting time is covered by the remaining 10 processes. We have 10 I/O-bound processes each spending $1ms$ in and taking 0.1 ms for each context switch. Similarly, one cpu-bound process will spend $1ms$ before being switched $0.1ms$. So we have:

$$U_{\text{CPU}} = \frac{\text{useful time}}{\text{total time}} = \frac{10 \cdot 1 + 1}{10 \cdot 1 + 1 + 11 \cdot 0.1} = 90.91\%$$

b. Each I/O-bound process will only spend $1ms$ in the allocated quantum time of $10ms$, then will be dispatched to the I/O queue, while the CPU-bound will utilize all the allocated quantum time, hence:

$$U_{\text{CPU}} = \frac{10 \cdot 1 + 10}{10 \cdot 1 + 10 + 11 \cdot 0.1} = 94.78\%$$

Problem 5.15
Explain the differences in how much the following scheduling algorithms discriminate in favor of short processes: a. FCFS b. RR c. Multilevel feedback queues

Solution:

a. No discrimination; the short process will be dispatched to the CPU until it releases it. waiting time is highly variable if there are long processes ahead.

b. RR ensures fairness among different processes by allocating equal share of quantum time. However, if a short process lies in the auxiliary queue it is more likely to be chosen over others in the regular ready queue.

c. As a process ponders, MLFQ decrease its priority, and gives high priority to the short term processes; It discriminates in favor of short term processes.

Problem 5.17
Assuming that no threads belong to the REALTIME_PRIORITY_CLASS and that none may be assigned a TIME_CRITICAL priority, what combination of priority class and priority corresponds to the highest possible relative priority in Windows scheduling?

Solution: A priority class of high, with level highest corresponds to 15.

Practical Part

Problem 2
Consider a clinic with one doctor and a very large waiting room (of infinite capacity). Any patient entering the clinic will wait in the waiting room until the doctor is free to see him/her. Similarly, the doctor also waits for a patient to arrive to treat. All communication between the patients and the doctor happens via a shared memory buffer. Any of the several patient processes, or the doctor process can write to it. Once the patient enters the doctors office, s/he conveys him/her symptoms to the doctor usinga call to consultDoctor(),which updates the shared memory with the patients symptoms. The doctor then calls treatPatient()to access the buffer and update it with details of the treatment. Finally, the patient process must call noteTreatment()to see the updated treatment details in the shared buffer, before leaving the doctors office. A template code for the patient and doctor processes is shown below. Enhance this code to correctly synchronize between the patient and the doctor processes. Your code should ensure that no race conditions occur due to several patients overwriting the shared buffer concurrently. Similarly, you must ensure that the doctor accesses the buffer only when there is valid new patient information in it, and the patient sees the treatment only after the doctor has written it to the buffer. You must use only semaphores to solve this problem. Clearly list the semaphore variables you use and their initial values first. Please pick sensible names for your variables.

Solution:

a.

```
    int session, symptoms, treatment;
    session = 1;
    symptoms = treatment = 0;

```

b.

```
        wait(session);
        consultDoctor();
        signal(symptoms);
        wait(treatment);
        noteTreatment();
        signal(session);

```

c.

```
    while(1) {
        wait(symptoms);
        treatPatient();
        signal(treatment);
    }

```

Problem 3
Consider a producer-consumer situation, where a process P produces an integer using the function produceNext()and sends it to process C. Process C receives the integer from P and consumes it in the function consumeNext(). After consuming this integer, C must let P know, and P must produce the next integer only after learning that C has consumed the earlier one. Assume that P and C get a pointer to a shared memory segment of 8 bytes that can store any two 4-byte integer-sized fields, as shown below. Both fields in the shared memory structure are zeroed out initially. P and C can read or write from it, just as they would with any other data object. Briefly describe how you would solve the producer-consumer problem described above, using only this shared memory as a means of communication and synchronization between processes P and C. You must not use anyother synchronization or communication primitive. You are provided template code below which gets a pointer to the shared memory, and produces/consumes integers. You must write the code for communicating the integer between the processes using the shared memory, with synchronization logic as required

Solution:

```
struct shmem_structure {
int field1;
int field2;
};

```

a. Producer:

```
    struct shmem_structure *shptr = get_shared_memory_structure();

    while(1) {
        int produced = produceNext();
        shptr->field1 = produced;
        shptr->field2 = 1;
            while(shptr->field2); /* DO NOTHING: Consuming */
    }

```

b. Consumer:

```
  struct shmem_structure *shptr = get_shared_memory_structure();

  while(1) {
      while(~shptr->field2); /* DO NOTHING: Producing */
    int consumed = shptr->field1;
    shptr->field2 = 0;
    consumeNext(consumed);
  }

```

Problem 5.17
Consider the readers and writers problem discussed in the slides/text book. Recall that multiple readers can be allowed to read concurrently, while only one writer at a time can access the critical section. Write down pseudocode to implement the functions readLock, readUnlock, writeLock, and writeUnlock that are invoked by the readers and writers to realize read/write locks. You must use only semaphores, and no other synchronization mechanism, in your solution. Further, you must avoid using more semaphores than is necessary. Clearly list all the variables (semaphores, and any other flags/counters you may need) and their initial values at the start of your solution. Use the notation down(x)and up(x)to invoke atomic down and up operations on a semaphore x that are available via the OS API. Use sensible names for your variables

Solution:

```
  int read_count = 0;
  int count_lock = 1;
  int resource_lock = 1;

  readLock() {
    down(count_lock);
    read_count++;
    if (count == 1)
      down(resource_lock);
    up(count_lock);
  }

  readUnlock() {
  down(count_lock);
  read_count--;
  if (count == 0)
    up(resource_lock);
  up(count_lock);
  }

  writeLock() {
    down(resource_lock);
  }
  writeUnlock() {
    up(resource_lock);
  }

```

Problem 5
Consider a multithreaded banking application. The main process receives requests to transfer money from one account to the other, and each request is handled by a separate worker thread in the application. All threads access shared data of all user bank accounts. Bank accounts are represented by a unique integer account number, a balance, and a lock of type mylock (much like a pthreads mutex) as shown below. struct account int accountnum;int balance;mylock lock;; Each thread that receives a transfer request must implement the transfer function shown below,which transfers money from one account to the other. Add correct locking (by calling the dolock(&lock) and unlock(&lock) functions on a mylock variable) to the tranfer function below, so that no race conditions occur when several worker threads concurrently perform transfers. Note that you must use the fine-grained per account lock provided as part of the account object itself, and not a global lock of your own. Also make sure your solution is deadlock free, when multiple threads access the same pair of accounts concurrently.

Solution: To get rid of deadlock we get rid of unsorted hierarchical locks, we can consider account number to sort out the locks.

```

  void transfer(struct account *from, struct account *to, int amount){
  if (from->accountnum > to->accountnum) {
    dolock(from->lock);
    dolock(to->lock);
  } else {
    dolock(to->lock);
    dolock(from->lock);
  }


  from->balance -= amount; // dont write anything...
  to->balance += amount; // ...between these two lines

  unlock(from->lock);
  unlock(to->lock);
  }

```