Homework Assignment #2

J. H. Wang

Apr. 11, 2024

Homework #2

- Chap.4: 4.8, 4.10, 4.16
- Chap.5: 5.14, 5.18, 5.22, 5.25
- Chap.6: 6.7, 6.15, 6.18
- Programming exercises:
  - Programming problems: 4.27*, (4.24**,) 6.33*
    - Note: Each student must complete all programming problems on your own
  - Programming projects for Chap. 4* (*2) & Chap. 5*
    - Team-based: select at least one programming project from each chapter
- Due: two weeks (Apr. 25, 2024)

- Chap. 4
  - 4.8: Provide two programming examples in which multithreading does not provide better performance than a single-threaded solution.
  - 4.10: Which of the following components of program state are shared across threads in a multithreaded process?

    (a) Register values
    (b) Heap memory
    (c) Global variables
    (d) Stack memory

- 4.16: 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)

(…to be continued)

(… continued from the previous slide)

- 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.

- Chap. 5:
  - 5.14: Most scheduling algorithms maintain a run queue, which lists processes eligible to run on a processor. On multicore systems, there are two general options:
    - (1) each processing core has its own run queue, or
    - (2) a single run queue is shared by all processing cores.
  - What are the advantages and disadvantages of each of these approaches?

- 5.18: The following processes are being scheduled using a preemptive, priority-based, round-robin scheduling algorithm.
- Each process is assigned a numerical priority, with a higher number indicating a higher relative priority.
- For processes with the same priority, a round-robin scheduler will be used with a time quantum of 10 units.
- If a process is preempted by a higher-priority process, the preempted process is placed at the end of the queue.

(… to be continued)

Thread	Priority	Burst	Arrival
P_1	8	15	0
P_2	3	20	0
P_3	4	20	20
P_4	4	20	25
P_5	5	5	45
5	10	15	55

(… to be continued)

- (a) Show the scheduling order of the processes using a Gantt chart.
- (b) What is the turnaround time for each process?
- (c) What is the waiting time for each process?

- 5.22: Consider a system running ten I/O-bound tasks and one CPU-bound task.
- Assume that the I/O-bound tasks issue an 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 a round-robin scheduler when:
- (a) The time quantum is 1 millisecond
- (b) The time quantum is 10 millisecond

– 5.25: Explain the differences in how much the following scheduling algorithms discriminate in favor of short processes:
  (a) FCFS
  (b) RR
  (c) Multilevel feedback queues

- Chap.6:
  - 6.7: The pseudocode of Figure 6.15 illustrates the basic push() and pop() operations of an array-based stack. Assuming that this algorithm could be used in a concurrent environment, answer the following questions:
    - (a) What data have a race condition?
    - (b) How could the race condition be fixed?

- Fig. 6.15:

```
push(item) {
   if (top < SIZE) {
      stack[top] = item;
      top++;
   }
   else
      ERROR
}

pop() {
   if (!is_empty()) {
      top--;
      return stack[top];
   }
   else
      ERROR
}

is_empty() {
   if (top == 0)
      return true;
   else
      return false;
}
```

- 6.15: Explain why implementing synchronization primitives by disabling interrupts is not appropriate in a single-processor system if the synchronization primitives are to be used in user-level programs.

- 6.18: The implementation of mutex locks provided in Section 6.5 suffers from busy waiting.
- Describe what changes would be necessary so that a process waiting to acquire a mutex lock would be blocked and placed into a waiting queue until the lock became available.

The main disadvantage of the implementation given here is that it requires **busy waiting**. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to `acquire()`. This continual looping is clearly a problem in a real multiprogramming system, where a single CPU core is shared among many processes. Busy waiting also wastes CPU cycles that some other process might be able to use productively. (In Section 6.6, we examine a strategy that avoids busy waiting by temporarily putting the waiting process to sleep and then awakening it once the lock becomes available.)

The type of mutex lock we have been describing is also called a **spinlock** because the process "spins" while waiting for the lock to become available. (We see the same issue with the code examples illustrating the `compare_and_swap()` instruction.) Spinlocks do have an advantage, however, in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. In certain circumstances on multicore systems, spinlocks are in fact the preferable choice for locking. If a lock is to be held for a short duration, one thread can "spin" on one processing core while another thread performs its critical section on another core. On modern multicore computing systems, spinlocks are widely used in many operating systems.

In Chapter 7 we examine how mutex locks can be used to solve classical synchronization problems. We also discuss how mutex locks and spinlocks are used in several operating systems, as well as in Pthreads.

Programming Problems

- 4.27*: The *Fibonacci sequence* is the series of numbers 0, 1, 1, 2, 3, 5, 8, … . Formally, it can be expressed as:
  $fib_0=0$
  $fib_1=1$
  $fib_n=fib_{n-1}+fib_{n-2}$
- Write a multithreaded program that generates the Fibonacci sequence using either the Java, Pthread, or Win32 thread library.
  (… to be continued)

- This program should work as follows:
  - On the command line, the user will enter the number of Fibonacci numbers that the program is to generate
  - The program will then create a separate thread that will generate the Fibonacci numbers, placing the sequence in data that can be shared by the threads (an array is probably the most convenient data structure)
  - When the thread finishes execution, the parent thread will output the sequence generated by the child thread
  - Because the parent thread cannot begin outputting until the child finishes, the parent will have to wait for the child thread to finish

- [optional] (4.24**): An interesting way of calculating pi is to use a technique known as *Monte Carlo*, which involves randomization. This technique works as follows:
  - Suppose you have a circle inscribed within a square, (Assume that the radius of this circle is 1.)
  - First, generate a series of random points as simple (x,y) coordinates
  - These points must fall within the Cartesian coordinates that bound the square
  - Of the total number of random points that are generated, some will occur within the circle
  - (… to be continued)

- Next, estimate pi by performing the following calculation:
  - Pi=4*(number of points in circle) / (total number of points)

– Write a multithreaded version of this algorithm that creates a separate thread to generate a number of random points.
  - The thread will count the number of points that occur within the circle and store that result in a global variable.
  - When this thread has exited, the parent thread will calculate and output the estimated value of pi.

- 6.33*: Assume that a finite number of resources of a single resource type must be managed.
  - Processes may ask for a number of these resources and will return them once finished.
  - As an example, many commercial software packages provide a given number of ***licenses***, indicating the number of applications that may run concurrently.
  - When the application is started, the license count is decremented.
  - When the application is terminated, the license count is incremented.
  - If all licenses are in use, requests to start the application are denied.
  - Such a request will be granted only when an existing license holder terminates the application and a license is returned.

- #define MAX_RESOUCRES 5
  int available_resources = MAX_RESOURCES;
  int decrease count(int count) {
    if (available resources < count)
        return -1;
    else {
        available resources -= count;
        return 0;
      }
  }
  int increase count(int count) {
    available resources += count;
    return 0;
  }

- The preceding program segment produces a race condition. Do the following:
  - (a) Identify the data involved in the race condition.
  - (b) Identify the location (or locations) in the code where the race condition occurs.
  - (c) Using a semaphore or mutex lock, fix the race condition. It is permissible to modify the decrease_count() function so that the calling process is blocked until sufficient resources are available.

End-of-Chapter Programming Projects

- Programming Projects for Chap. 4: (Choose one)
  - Project 1. Sudoku solution validator
    - To design a multithreaded application that determines whether the solution to a Sudoku puzzle is valid.
    - Passing parameters to each thread
    - Returning results to the parent thread
  - Project 2. Multithreaded sorting application
    - Write a multithreaded sorting program that works as follows: A list of integers is divided into two smaller lists of equal size. Two separate threads sort each sublist using a sorting algorithm of your choice. The two sublists are then merged by a third thread.

End-of-Chapter Programming Projects

- Programming Projects for Chap. 5:
  - Project. Scheduling Algorithms
    - This project involves implementing several process scheduling algorithms
      - FCFS
      - SJF
      - Priority-based
      - Round-Robin
      - Priority with round-robin
    - The implementation of this project may be completed in either C or Java

Homework Submission

- For hand-written exercises, please hand in your homework on paper in class
- For programming exercises, please upload your program to the iSchool+ as follows:
  - Program uploading: a compressed file (in .zip format) including source codes, execution snapshot, and documentation
  - Documentation: containing -
    - Team members and responsibility
    - Compilation or configuration instructions if it needs special environment to compile or run
  - Please contact with the TA if you are unable to upload your homework

Any Question or Comments?