Operating System HW2 110590450 歐佳昀

Chap. 4

4.8: Provide two programming examples in which multithreading does not provide better performance than a single-threaded solution.

Any kind of sequential program is not a good candidate for threading. Like a program that calculates an individual tax return is one such example.

Such a program must closely monitor its own working space,Like "shell" program

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

share (b) heap memory and (c) global variables.

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)

(1) How many threads will you create to perform the input and output? Explain.

Because the file must be accessed sequentially, the work of reading and writing it cannot be parallelized, and only a single thread can use for each of these tasks.

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

The CPU-bound part should be evenly divided to 4 processors, and 4 threads can be created for this part of the program. Using fewer than 4 would waste processor resources, more threads till not be able to run simultaneously.

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?

(1) each processing core has its own run queue, or

advantage: Less chance of conflicts.

disadvantage : Management inconvenience.

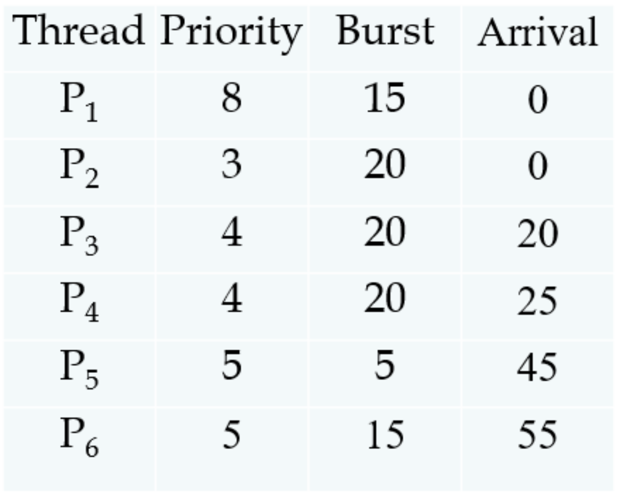
(2) a single run queue is shared by all processing cores.

advantage: Convenient management.

disadvantage: Synchronization issues, potential for conflicts.

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.



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
P_6	5	15	55

(a) Show the scheduling order of the processes using a Gantt chart.

P1	P1	P2	P3	P4	P3	P5	P4	P6	P6	P3	P4	P2
0 10	15	20	30	40	45	50	55	65	70	75	80	95

Turnaround time = Exit time – Arrival time

Waiting time = Turnaround time – Burst time

(b) What is the turnaround time for each process?

P1 = 15, P2 = 95, P3 = (75 – 20) = 55, P4 = (80 – 25) = 55, P5 = 5, P6 = 15

(c) What is the waiting time for each process?

P1 = 0, P2 = (95-20) =75, P3 =(55-20) = 35, P4 =(55-20) = 35,

P5 = (5-5) = 0, P6 = (15-15) = 0

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

$11/(10*(1+0.1) + (1+0.1)) = 1/1.1 = 91\%$.

(b) The time quantum is 10 millisecond

$(10+10)/(10*(1+0.1) + (10+0.1)) = 20/21.1 = 94\%$

5.25: Explain the differences in how much the following scheduling algorithms discriminate in favor of short processes:

(a) FCFS

First-Come, First-Served，Short jobs are disadvantaged because if they arrive after long jobs, they will experience longer waiting times

(b) RR

All jobs are treated equally with the same CPU time slices, allowing short jobs to exit the system faster

(c) Multilevel feedback queues

Like RR, they favor short jobs

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:

```
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;
}
```

(a) What data have a race condition?

The variable top have race condition because ex . the top in pop() may change before stack[top] = item is executed, affecting the content of stack[].

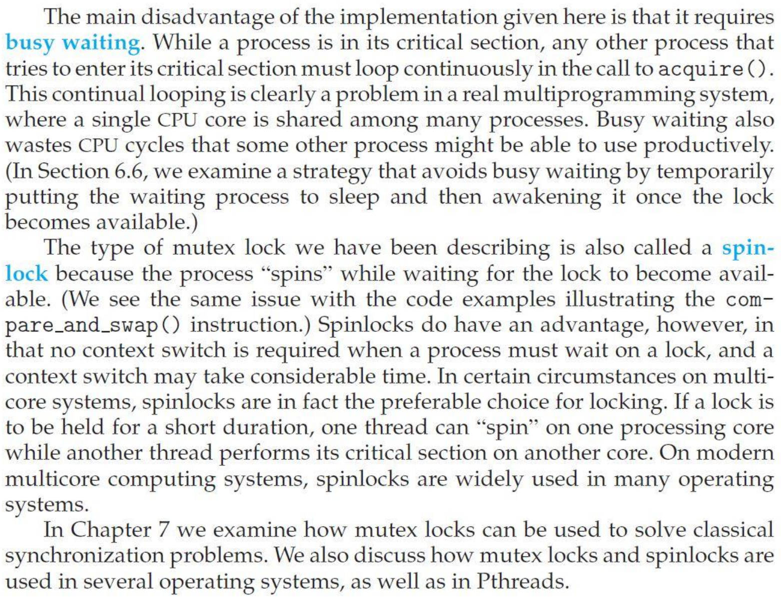
(b) How could the race condition be fixed?

Mutex locks can be used to resolve this race condition. Utilizing mutex locks ensures that only one thread can access the shared resource at any given time, thus preventing the occurrence of race conditions. By employing mutex locks before accessing the stack in the push() and pop() functions, it can be ensured that they do not execute simultaneously, deal race conditions.

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.

If user-level programs are given the ability to disable interrupts, they can disable timer interrupts and prevent context switching, allowing them to use the processor without letting other processes execute.

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.

ANS:

The changes needed are very similar to those made for semaphores. Associated with each mutex lock would be a queue of waiting processes. When a process determines the lock is unavailable, they are placed into the queue. The calling process must be placed into the wait queue and initiate a context switch. When a process releases the lock, it removes and awakens the first process from the list of waiting processes.