操作系统作业

姓名，学号

1. The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, P0 and P1, share the following variables:

   ```
   boolean flag[2]; /* initially false */
   int turn;
   ```

   The structure of process Pi (i == 0 or 1) is shown in Figure 1; the other process is Pj (j == 1 or 0). Prove that the algorithm satisfies all three requirements for the critical-section problem.

```
do {
   flag[i] = true;

   while (flag[j]) {
      if (turn == j) {
         flag[i] = false;
         while (turn == j)
            ; /* do nothing */
         flag[i] = true;
      }
   }

      /* critical section */

   turn = j;
   flag[i] = false;

      /* remainder section */
} while (true);
```

Figure 1: The structure of process Pi for Question 1.

Answer：

(1) Mutual Exclusion:

Mutual exclusion is ensured through the use of the *flag* and *turn* variables.

If both processes set their *flag* to *true*, only one will succeed, namely, the process whose *turn* it is. The waiting process can only enter its critical section when the other process updates the value of *turn*.

(2) Progress:

Progress is provided, again through the *flag* and *turn* variables.

This algorithm does not provide strict alternation. Rather, if a process wishes to access their critical section, it can set their *flag* variable to *true* and enter their critical section. It sets *turn* to the value of the other process only upon exiting its critical section. If this process wishes to enter its critical section again—before the other process—it repeats the process of entering its critical section and setting *turn* to the other process upon exiting.

(3) Bounded Waiting:

Bounded waiting is preserved through the use of the *turn* variable. Assume two processes wish to enter their respective critical sections. They both set their value of *flag* to *true*; however, only the thread whose *turn* it is can proceed; the other thread waits. If bounded waiting were not preserved, it would therefore be possible that the waiting process would have to wait indefinitely while the first process repeatedly entered—and exited—its critical section. However, Dekker's algorithm has a process set the value of *turn* to the other process, thereby ensuring that the other process will enter its critical section next.

2. Consider the code example for allocating and releasing processes shown in Figure 2.
   a. Identify the race condition(s).
   b. Assume you have a mutex lock named `mutex` with the operations acquire() and release(). Indicate where the locking needs to be placed to prevent the race condition(s).

```
#define MAX_PROCESSES 255
int number_of_processes = 0;

/* the implementation of fork() calls this function */
int allocate_process() {
int new_pid;

   if (number_of_processes == MAX_PROCESSES)
      return -1;
   else {
      /* allocate necessary process resources */
      ++number_of_processes;

      return new_pid;
   }
}

/* the implementation of exit() calls this function */
void release_process() {
    /* release process resources */
    --number_of_processes;
}
```

Figure 2: Code for Question 2.

Answer：

a. There is a race condition on the variable *number_of_processes*.
b. A call to *acquire()* must be placed upon entering each function and a call to *release()* immediately before exiting each function.

3. Servers can be designed to limit the number of open connections. For example, a server may wish to have only N socket connections at any point in time. As soon as N connections are made, the server will not accept another incoming connection until an existing connection is released. Use semaphores to limit the number of concurrent connections in the server.

Answer：

1) A semaphore is initialized to the number of allowable open socket connections.
2) When a connection is accepted, the *acquire()* method is called; when a connection is released, the *release()* method is called.
3) If the system reaches the number of allowable socket connections, subsequent calls to *acquire()* will block until an existing connection is terminated and the release method is invoked.

4. Consider the traffic deadlock depicted in Figure 3.
   a. Show that the four necessary conditions for deadlock indeed hold in this example.
   b. State a simple rule for avoiding deadlocks in this system.

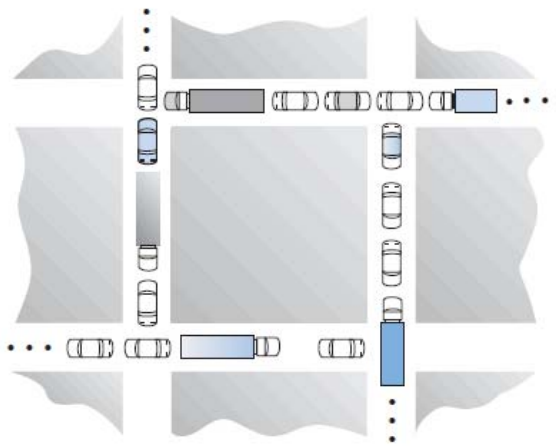


Figure 3. Traffic deadlock for Question 4.

Answer：

a. The four necessary conditions for a deadlock are (1) mutual exclusion; (2) hold-and-wait; (3) no preemption; and (4) circular wait.
   1) The mutual exclusion condition holds since only one car can occupy a space in the roadway.
   2) Hold-and-wait occurs where a car holds onto its place in the roadway while it waits to advance in the roadway.
   3) A car cannot be removed (i.e. preempted) from its position in the roadway.
   4) Lastly, there is indeed a circular wait as each car is waiting for a subsequent car to advance. The circular wait condition is also easily observed from the graphic.

b. A simple rule that would avoid this traffic deadlock is that a car may not advance into an intersection if it is clear it will not be able immediately to clear the intersection.

5. Consider the deadlock situation that can occur in the diningphilosophers problem when the philosophers obtain the chopsticks one at a time. Discuss how the four necessary conditions for deadlock hold in this setting. Describe a deadlock-free solution, and discuss which necessary conditions are eliminated in your solution.

Answer：

Deadlock is possible because the four necessary conditions hold in the following manner:

1) mutual exclusion is required for chopsticks,
2) the philosophers hold onto the chopstick in hand while they wait for the other chopstick,
3) there is no preemption of chopsticks in the sense that a chopstick allocated to a philosopher cannot be forcibly taken away,
4) there is a possibility of circularwait.

Deadlocks could be avoided by overcoming the conditions in the following manner: (anyone is ok)

- Allow simultaneous sharing of chopsticks. (NO “Mutual Exclusion”)
- Have the philosophers relinquish the first chopstick if they are unable to obtain the other chopstick. (NO “Hold-and-wait”)
- Allow chopsticks to be forcibly taken away if a philosopher has had a chopstick for a long period of time. (Preemption)
- Enforce a numbering of the chopsticks and always obtain the lower numbered chopstick before obtaining the higher numbered one. (NO “Circular Wait”)
- Allow at most four philosophers to be sitting simultaneously at the table. (NO “Circular Wait”)
- “The Final Solution” in our PPT(ch5_part3.pdf). (NO “Hold-and-wait” && NO “Circular Wait”)

6. Discuss how the following pairs of scheduling criteria conflict in certain settings.
   a. CPU utilization and response time
   b. Average turnaround time and maximum waiting time
   c. I/O device utilization and CPU utilization

Answer：

a. CPU utilization and response time:
CPU utilization is increased if the overheads associated with context switching is minimized. The context switching overheads could be lowered by performing context switches infrequently. This could, however, result in increasing the response time for processes.

b. Average turnaround time and maximum waiting time:
Average turnaround time is minimized by executing the shortest tasks first. Such a scheduling policy could, however, starve long-running tasks and thereby increase their waiting time.

c. I/O device utilization and CPU utilization:
CPU utilization is maximized by running long-running CPU-bound tasks without performing context switches. I/O device utilization is maximized by scheduling I/O-bound jobs as soon as they become ready to run, thereby incurring the overheads of context switches.

7. 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

Answer：

a. When $\alpha = 0$ and $\tau_0 = 100$ milliseconds, the formula always makes a prediction of 100 milliseconds for the next CPU burst.

b. When $\alpha = 0.99$ and $\tau_0 = 10$ milliseconds, the most recent behavior of the process is given much higher weight than the past history associated with the process. Consequently, the scheduling algorithm is almost memoryless, and simply predicts the length of the previous burst for the next quantum of CPU execution.

8. Consider the following set of processes, with the length of the CPU burst time given in milliseconds:

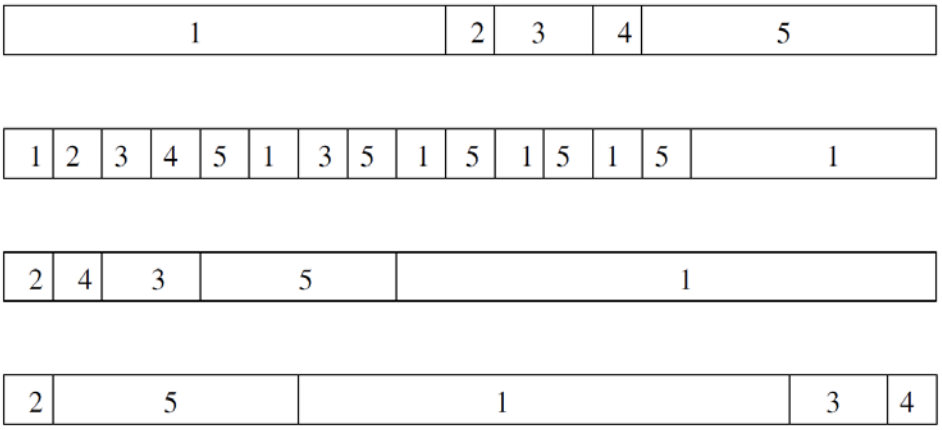
Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

The processes are assumed to have arrived in the order P1, P2, P3, P4, P5, all at time 0.

a. Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, nonpreemptive priority (a smaller priority number implies a higher priority), and RR (quantum = 1).
b. What is the turnaround time of each process for each of the scheduling algorithms in part a?
c. What is the waiting time of each process for each of these scheduling algorithms?
d. Which of the algorithms results in the minimum average waiting time (over all processes)?

Answer：

a. The four Gantt charts are:



b. Turnaround time

	FCFS	RR	SJF	Priority
P_1	10	19	19	16
P_2	11	2	1	1
P_3	13	7	4	18
P_4	14	4	2	19
P_5	19	14	9	6

c. Waiting time (turnaround time minus burst time)

	FCFS	RR	SJF	Priority
P_1	0	9	9	6
P_2	10	1	0	0
P_3	11	5	2	16
P_4	13	3	1	18
P_5	14	9	4	1

d. SJF

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

Answer： b d

10. 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 0.1millisecond and that all processes are long-running tasks. Describe is the CPU utilization for a round-robin scheduler when:
    a. The time quantum is 1 millisecond
    b. The time quantum is 10 milliseconds

Answer：

a. The time quantum is 1millisecond:
Irrespective of which process is scheduled, the scheduler incurs a 0.1 millisecond context-switching cost for every context-switch. This results in a CPU utilization of 1/1.1 * 100 = 91%.

b. The time quantum is 10 milliseconds:
The I/O-bound tasks incur a context switch after using up only 1 millisecond of the time quantum. The time required to cycle through all the processes is therefore 10*1.1 + 10.1 (as each I/O-bound task executes for 1 millisecond and then incur the context switch task, whereas the CPU-bound task executes for 10 milliseconds before incurring a context switch). The CPU utilization is therefore 20/21.1 * 100 = 94%.

11. Assume that two tasks Aand B are running on a Linux system. The nice values of Aand B are −5 and +5, respectively. Using the CFS scheduler as a guide, describe how the respective values of vruntime vary between the two processes given each of the following scenarios:
    a. Both Aand B are CPU-bound.
    b. A is I/O-bound, and B is CPU-bound.
    c. A is CPU-bound, and B is I/O-bound.

Answer：

a. Since A has a higher priority than B, *vruntime* will move more slowly for A than B. If both A and B are CPU-bound (that is they both use the CPU for as long as it is allocated to them),

vruntime will generally be smaller for A than B, and hence A will have a greater priority to run over B.

b. In this situation, *vruntime* will be much smaller for A than B as (1) *vruntime* will move more slowly for A than B due to priority differences, and (2) A will require less CPU-time as it is I/O-bound.

c. This situation is not as clear, and it is possible that B may end up running in favor of A as it will be using the processor less than A and in fact its value of *vruntime* may in fact be less than the value of *vruntime* for B.

12. Give an example to illustrate under what circumstances rate-monotonic scheduling is inferior to earliest-deadline-first scheduling in meeting the deadlines associated with processes?

Answer：

Example 1:

Consider two processes P_1 and P_2 where $p_1 = 50$, $t_1 = 25$ and $p_2 = 75$, $t_2 = 30$.

If P_1 were assigned a higher priority than P_2, then the following scheduling events happen under rate-monotonic scheduling.

P_1 is scheduled at $t = 0$, P_2 is scheduled at $t = 25$, P_1 is scheduled at $t = 50$, and P_2 is scheduled at $t = 75$. P_2 is not scheduled early enough to meet its deadline.

The earliest deadline schedule performs the following scheduling events:

P_1 is scheduled at $t = 0$, P_2 is scheduled at $t = 25$, P_1 is scheduled at $t = 55$, and so on.

This schedule actually meets the deadlines and therefore earliest-deadline-first scheduling is more effective than the rate-monotonic scheduler.

Example 2:

Example in our course PPT