

Exercise 3: Scheduling and Concurrency Control (25%)

The students are required to answer any three of the following four questions. If a student answers all the questions, the best ones will be chosen to maximize the student's grade. Weights sum up to 120%. The maximal grade, however, is 100. Partially solved questions do count.

(30%) Problem 1: scheduling

In a non-preemptive batch system, at time t the ready queue contains three jobs with run times r_1, r_2, r_3 known in advance. These 3 jobs arrived to the system at times t_1, t_2, t_3 respectively. No additional jobs arrive. Figure 1 shows the linear increase of their response ratios (also known as slow-down factor) over time. To remind you, the response ratio of process i is defined as follows: $RR_i = (w_i + r_i)/r_i$ where w_i is the total time that a process waited for the CPU.

Use this example to find a scheduling algorithm, which minimizes the maximum response ratio for any given batch of jobs ignoring further arrivals. Explain (briefly) why your algorithm indeed minimizes the maximal response ratio. (*Hint*: Decide first which job to schedule as the last one).

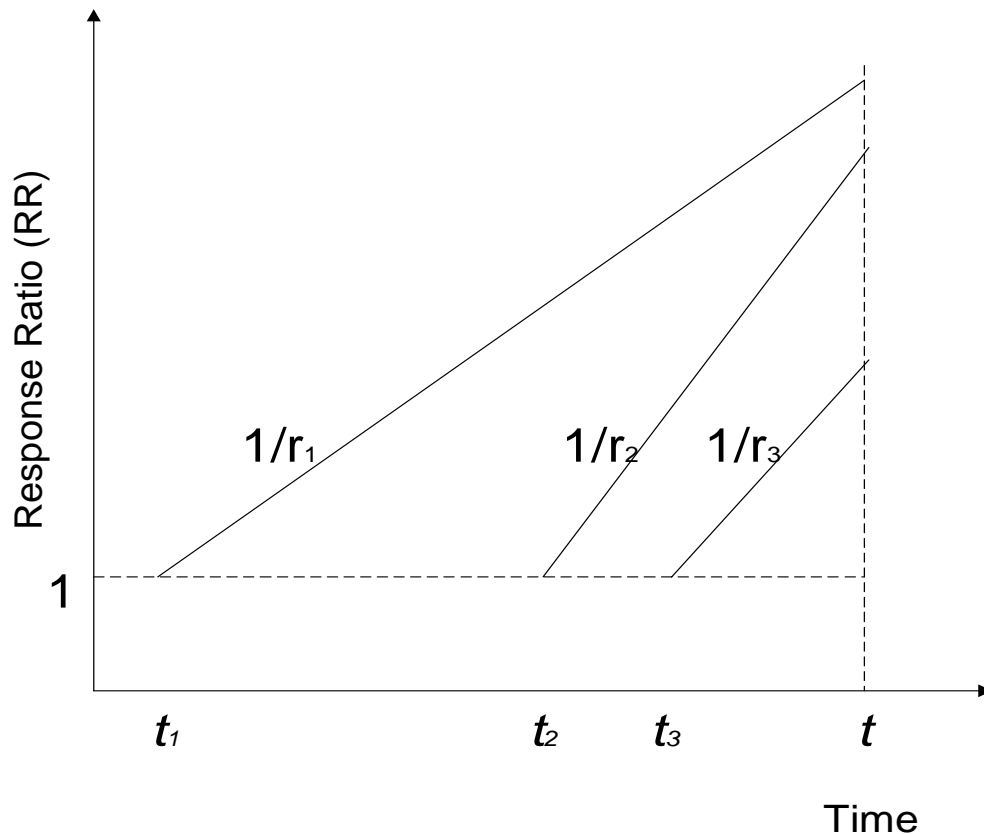


Figure 1

Solution:

The scheduling algorithm is as follows:

Compute the response ratios of the jobs $1, 2, \dots, n$ at time $T = t + r_1 + r_2 + \dots + r_n$. Let J_1 be the job with the smallest response ratio among the jobs $1, 2, \dots, n$. J_1 will be executed *last*.

Examine the response ratios of the jobs $1, 2, \dots, n-1$ at time $T = t + r_1 + r_2 + \dots + r_{n-1}$. Let J_2 be the job with the smallest response ratio among the jobs $1, 2, \dots, n-1$. J_2 will be executed before the last. Repeat the above procedure for the remaining jobs. The scheduling sequence is: $J_n J_{n-1} \dots J_1$

This algorithm minimizes the maximum responses ratio because it consistently postpones jobs that will suffer least increase of their response ratios. The algorithm complexity is $O(n^2)$.

(30%) Problem 2:

Prove or disprove (by giving a detailed counterexample) that the following modified version of the Lamport's Bakery algorithm satisfies the mutual exclusion property:

```

Shared:
    boolean choosing[n];
    int my_ticket[n];
    int current_ticket;

Initially:
    for all i,  $0 \leq i < n$ :
        choosing[i] = FALSE;
        my_ticket[i] =  $\infty$ ;
        current_ticket = 1;

Process Pi code:

entry:
    choosing[i] = TRUE;
    my_ticket[i] = current_ticket;
    current_ticket = current_ticket + 1;
    choosing[i] = FALSE;
    for (j=0; j < n; j++) {
        while(choosing[j]);
        while((my_ticket[j] < my_ticket[i]) ||
              ((my_ticket[j] == my_ticket[i]) && (j < i)));
    }
    Critical section

exit:
    my_ticket[i] =  $\infty$ ;

```

*"...Everything I learned about concurrency came from studying the Bakery algorithm".
Leslie Lamport (the inventor of the Bakery algorithm)*

Solution:

Since `current_ticket` is a shared read/write variable (i.e., the only atomic operations over `current_ticket` are (1) read: taking the value of `current_ticket`; and (2) write: assigning a value to the `current_ticket`), it cannot be incremented without involving an auxiliary variable, `tmp`, which is local to the process. This means that the statement `current_ticket = current_ticket + 1` is translated into the following code:

```

tmp = current_ticket;
tmp = tmp + 1;
current_ticket = tmp;

```

Thus, if a process `P` is suspended after it has completed the first two lines, and another process `Q` increments `current_ticket` in the meantime, `P` can override `current_ticket` with its old (possibly a

smaller) value when it resumes. (Note that this does not happen in the original Bakery algorithm, because the maximum is computed each time anew out of the numbers of all the processes). As a result the algorithm violates the mutual exclusion as demonstrated by the following scenario involving three processes denoted P1, P2 and P3 respectively:

P1	P2	P3	my_tick et[1]	my_tick et[2]	my_tick et[3]	choosing [1]	choosing [2]	choosing [3]	current_ticket
choosing[1]=T my_ticket[1]=current_ticket tmp=current_ticket			inf	inf	inf	F	F	F	1
	choosing[2]=T my_ticket[2]=current_ticket tmp=current_ticket tmp=tmp+1 current_ticket=tmp choosing[2]=F		1			T			
		choosing[3]=T my_ticket[3]=current_ticket tmp=current_ticket tmp=tmp+1 current_ticket=tmp choosing[3]=F			2		F	T	2
tmp=tmp+1 current_ticket=tmp choosing[1]=F Enters C.S. with ticket (1,1) CRITICAL SECTION my_ticket[1]=inf								F	3
	Enters C.S. with ticket (1,2) CRITICAL SECTION my_ticket[2]=inf		inf						2
		Enters C.S. with ticket (2,3) CRITICAL SECTION		inf					3
choosing[1]=T my_ticket[1]=current_ticket tmp=current_ticket tmp=tmp+1 current_ticket=tmp choosing[1]=F Enters C.S. because (2,1)<(2,3) CRITICAL SECTION			2			T			3
Both P1 and P3 are in the Critical Section									

(40%) Problem 3:

In the class, we have seen that hardware primitives, being capable of performing several computational steps atomically, can solve the critical section problem. As a specific example, we examined the Test-and-Set primitive. In this exercise we will consider another such primitive, called Fetch-and-Add (FAA) that is defined by the following pseudo-code:

```

FAA(int s, int val):
    tmp = s;
    s = s + val;
    return tmp;

```

The following code is proposed as a solution to the critical section problem for n processes ($n \geq 2$):

```

Shared: int s;
Initially: s = 1;

Process Pi code:

entry:
    while (FAA(s,-1) != 1) {
        FAA(s,1);
    }

    Critical section

exit:
    FAA(s,1);

```

Answer the following questions:

1. Prove or disprove:
 - 1.1 The algorithm above satisfies the mutual exclusion requirement: i.e., if process P_i is executing in the critical section, then no other process is in the critical section. (10%)
 - 1.2 The algorithm above satisfies the progress requirement: i.e., if P_i is in its entry section and no process is in the critical section., then some process eventually enters the critical section. (10%)
 - 1.1 The algorithm above satisfies the fairness requirement: i.e., if no process remains in the critical section forever, then any process requesting entry into the critical section will eventually enter the critical section. (10%)
2. Give a simple solution to the critical section problem using FAA, which satisfies all the three critical section requirements (10%).

Solution:

1.
 - 1.1. The mutual exclusion is satisfied.
Proof: For the sake of the proof, we assume the existence of a global discrete clock with the set of integer numbers as a domain of its clock ticks. The clock is initialized to 0, and is incremented each time an FAA() operations (either FAA(s,-1) or FAA(s,1)) is invoked by some process.
We first note that the following facts hold for each time t:
 - a) $s=1-\#FAA(s,-1)_t+\#FAA(s,1)_t$, where #FAA() indicates the number of times the corresponding FAA operations has been invoked before t;
 - b) There exists a one-to-one function f from the set of all FAA(s,1) operations invoked before t to the set of FAA(s,-1) operations invoked before t, satisfying the following:
 - 1) Both ϕ and $f(\phi)$ are invoked by the same process; and
 - 2) $f(\phi)$ precedes ϕ .

We now show that the following simple results hold:

Lemma 1: Let t be a time such that $s=1$ at t, then $\#FAA(s,-1)_t=\#FAA(s,1)_t$.

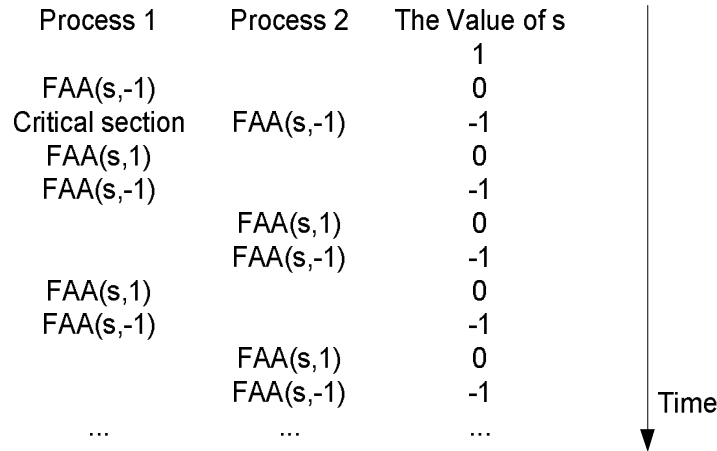
Proof: Follows directly from (a) above.

Lemma 2: If $\#FAA(s,-1)_t=\#FAA(s,1)_t$ at some step t, then each FAA(s,-1) executed by a process p before t must be followed by FAA(s,1) executed by the same process p before t.

Proof: Assume by contradiction that there exists a process p such that $\phi=FAA(s,-1)$ is the last operation invoked at p before t. Since $\#FAA(s,-1)=\#FAA(s,1)$, there exists a one-to-one function which maps all operations FAA(s,1) invoked before t to all operations FAA(s,-1) invoked before t. Then, for any such function f, there exists $\psi=FAA(s,1)$ such that $f(\psi)=\phi$, and either ϕ does not precede ψ , or ϕ and ψ are invoked by different processes. However, (b) above implies that a one-to-one function such that both ψ and $f(\psi)$ are invoked by the same process, and $f(\psi)$ precedes ψ must exist. A contradiction.

Finally, let p be a process such that FAA(s,-1) returns 1 at p at some time t_1 . By the algorithm, p can enter the critical section at some time $t > t_1$. Since s becomes 0 after FAA(s,-1), then in order for a process q to enter the critical section at some time $t' > t$, there must exist time $t_2 > t_1$ such that s becomes 1 again at t_2 . By Lemma 1, $\#FAA(s,-1)_{t_2}=\#FAA(s,1)_{t_2}$. Therefore, by Lemma 2, FAA(s,-1) invoked by p at $t_1 < t_2$, must be followed by FAA(s,1) invoked by p before t_2 . By the algorithm, this FAA(s,1) can be invoked only in the exit section of p. Therefore, p must have been left the critical section by t_2 . Therefore, q will be executing alone in the critical section as needed.

- 1.2. Progress is violated. The following scenario shows how progress can be violated even for 2 processes:



1.3. Since progress is violated, fairness is violated as well.

2. The idea is to use FAA as an atomic “ticket producer”. The resulting code is much like what the Bakery would have been if the increment was an atomic operation. It is easy to see that all the three C.S. requirements are satisfied:

```

Shared: int s, cur_ticket;
Initially:
    s = 0;
    cur_ticket=0;

Process Pi code:
entry:
    my_ticket = FAA(s,1);
    while(cur_ticket < my_ticket); // busy wait for my turn

Critical section

exit:
    cur_ticket = my_ticket+1;

```

(20%) Problem 4:

Implement a thread-safe finite stack of size k . The stack is an object that supports two operations:

1. `push(<item>)` : if there is a free space in the stack adds the item to the stack’s top, and returns immediately. Otherwise, this operation blocks, until a free space on the stack is available.
2. `<item> pop()` : return the item that is currently on the top of the stack. If stack is empty, this operation blocks.

Thread-safety (i.e., the guaranteed consistent behavior of the data structure in spite of multiprogramming) should hold for any number n of threads. Your solution should not make use of busy waiting. Hint: use semaphores.

Solution (one of many possible):

Define Stack data structure as following. This is “very pseudo” code.

```

Stack {
    item buf[k]; //buffer of items
    int top = 0; //points to the first free slot
    mutex.count = 1; //binary mutex semaphore initialized to 1

```

```

    empty.count = k; //counting semaphore initialized to k
    items.count = 0; //counting semaphore initialized to 0
}

void push(item x) {
    this.empty.P();
    this.mutex.P();
    this.buf[this.top++] = x;
    this.mutex.V();
    this.items.V();
}

item pop() {
    item k;
    this.items.P();
    this.mutex.P();
    k = this.buf[--this.top];
    this.mutex.V();
    this.empty.V();
    return k;
}

```