Leiðbeiningar

Leiðbeiningar um notkun:

- **formatta hverja spurningu með HEADING 1**
- skrifa heiti spurningar með forminu:
    - ár, tegund spurningar, númer dæmis, nafn þess sem svarar:
        - 2020, lokapróf, 3
        - 2022, skilaverkefni, 4
        - 2019, lokapróf, 7
- skrifa tags fyrir neðan til að skrá um hvaða efni spurninginn nær til með forminu:
    - #endurkvæmni #n_queens, #gráðug_reiknirit
    - **formatta tags með HEADING 2**
- setja page break á milli spurninga
- ekki nota önnur “headings” í skjalinu - bara formatta “manually” :-)

Efnisyfirlit

2020, lokapróf, 1, bjarni

#os_related_questions

a. System calls are needed to remove memory protection and share memory between processes.

b. By using suitable scheduling like Round Robin scheduling algorithm.

c. Direct-memory access.

2020, lokapróf, 2, bjarni

a. Process synchronization, disadvantage: Race conditions can occur. Advantage: Fast

b. TCP (Transmission control protocol):

    i. Used to connect to an endpoint other than your computer through sockets. TCP is a stream type connection and lets the sender know if data has been arrived. If not it will be retransmitted.

    ii. Sockets use ports to communicate. TCP uses a server as a connection hub for the communication. This opens an endless steam for data transferring.

2021, lokapróf, 3, ósvaldur

#cpu_scheduling_algorithms

3) (15%) Consider the processes P_1 to P_5 with the *following arrival times* a_i, *service times* d_i, *and priorities* p_i:

Process	P_1	P_2	P_3	P_4	P_5
Arrival time a_i	0	2	3	8	9
Service time d_i	4	3	9	8	6
Priority p_i	low	low	high	middle	high

Let the time quantum be 4. Determine the resulting schedule for the algorithms below and calculate the average residence time (sufficient to provide the result as fraction):

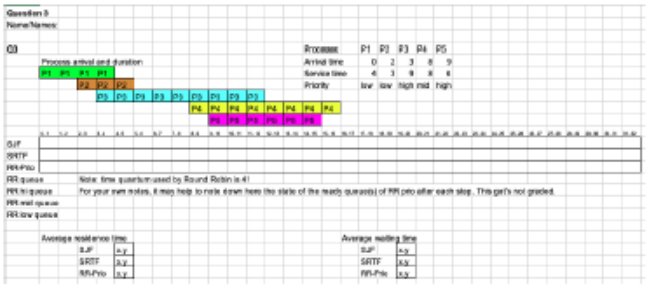
(a) Shortest Job First (SJF) (5%)

(b) Shortest Remaining Time First (SRTF) (5%)

(c) Round-Robin with priorities (RRPrio)[1] (5%)

Use the Gantt chart from below link to fill in your solution and paste on the next page a screenshot or PDF of the sheet here below.

https://docs.google.com/spreadsheets/d/1oq3YKTwNGDZ5KFKz9-FIML7dWufX1GvAx6XZlj-obnc/edit#gid=0



You may fill in the solution here below:

Average residence time SJF =
Average residence time SRTF =
Average residence time RRPrio =

[1] If a new process (even one with a higher priority) arrives, a process that has started its time slice is still allowed to consume it and will not get preempted during this time slice.

Hér er svarið mitt:

Question 3

Name/Names:

Q3

Processes	P1	P2	P3	P4	P5
Arrival time	0	2	3	8	9
Service time	4	3	9	8	6
Priority	low	low	high	mid	high

Process arrival and duration

	0..1	1..2	2..3	3..4	4..5	5..6	6..7	7..8	8..9	9..10	10..11	11..12	12..13	13..14	14..15	15..16
	P1	**P1**	**P1**	**P1**												
			P2	P2	P2											
				P3	**P3**	**P3**	**P3**	**P3**	**P3**	**P3**	**P3**	P3				
									P4	P4	P4	P4	P4	P4	P4	P4
										P5	**P5**	**P5**	**P5**	**P5**	**P5**	
SJF	P1	P1	P1	P1	P2	P2	P2	P3	P3	P3	P3	P3	P3	P3	P3	P3
SRTF	P1	P1	P1	P1	P2	P2	P2	P3	P3	P5	P5	P5	P5	P5	P5	P3
RR-Prio	P1	P1	P1	P1	P3	P3	P3	P3	P3	P3	P3	P3	P5	P5	P5	P5

	16..17	17..18	18..19	19..20	20..21	21..22	22..23	23..24	24..25	25..26	26..27	27..28	28..29	29..30	30..31	31..32
SJF	P5	P5	P5	P5	P5	P5	P4	P4	P4	P4	P4	P4	P4	P4		
SRTF	P3	P3	P3	P3	P3	P3	P4	P4	P4	P4	P4	P4	P4	P4		
RR-Prio	P3	P5	P5	P4	P4	P4	P4	P4	P4	P4	P4	P2	P2	P2		

RR queue — Note: time quantum used by Round Robin is 4!

RR hi queue — For your own notes, it may help to note down here the state of the ready queue(s) of RR prio after each step. This get's not graded.

RR mid queue

RR low queue

Average residence time		p1	p2	p3	p4	p5
SJF	11.4	4	5	13	22	13
SRTF	11.2	4	5	19	22	6

Average waiting time		p1	p2	p3	p4	p5
SJF	5.4	0	2	4	14	7
SRTF	5.2	0	2	10	14	0

sjá -
https://docs.google.com/spreadsheets/d/1ohzN7MH8dSF2N7GpG5uHQK4aspWEvyUfRH1CV6v2ToI/edit?usp=sharing

2021, lokapróf, 4, ósvaldur

#monitor_pseudocode #producer_consumer_problem

4) (10%) Producer-consumer problem

The monitor pseudocode in the right column has been written to solve a producer-consumer problem with one producer process (which is running method `producer ()`), two independent consumer processes (running the identical methods `consumer1()` and `consumer2()`), and a shared unbounded buffer (`queue`).

Each consumer may arbitrarily consume an item from the buffer as long as the buffer is not empty. The three methods run concurrently, using the mutual exclusion provided by a monitor.

In the underlying monitor approach, `signalAll` wakes up *all* waiting processes and signal-and-continue semantics is used (=waiting processes are unblocked after the signaling process left the monitor).

Does the provided pseudocode solve the described producer-consumer problem or is the code incorrect? If you think, the code is incorrect, provide and explain an example scenario or schedule in which a problem occurs, such as a deadlock, a race condition, or an empty queue is dequeued, or anything else which is unintended happens!
(**Note** that according to the above description, it is OK that access to the queue does not occur in a certain order or may be even unfair/lead to starvation. This does therefore not qualify as a problem.)

```
Monitor {                                   1
  condition notEmptyAnymore;                2
  private Queue queue=empty;                3
                                            4
  public producer() {                       5
    while (true) {                          6
      queue.enqueue(42);                    7
      notEmptyAnymore.signalAll;            8
    }                                       9
  }                                         10
                                            11
  public consumer1() {                      12
    int item;                               13
    while (true) {                          14
      if (queue.isEmpty) {                  15
        notEmptyAnymore.wait;               16
      }                                     17
      item=queue.dequeue();                 18
    }                                       19
  }                                         20
                                            21
  public consumer2() {                      22
    int item;                               23
    while (true) {                          24
      if (queue.isEmpty) {                  25
        notEmptyAnymore.wait;               26
      }                                     27
      item=queue.dequeue();                 28
    }                                       29
  }                                         30
}                                           31
void main() {                               33
  parallel { producer(),                    34
    consumer1(), consumer2() };             35
}                                           36
```

Svar:
This does not solve the producer-consumer problem. This might lead to a consumer “consuming” (dequeuing) an item from an empty queue. Let's say the queue is empty and both

consumer1 and consumer2 are waiting. Next the producer produces an item (adds an item to the queue) and signals the consumers. Now both consumers wake up and try to consume an item but there is only one item in the queue, so one of the consumers ends up trying to dequeue an empty queue.

2021, lokapróf, 5, ósvaldur

#semaphores #synchronization problem

5) (13%) Use semaphores to solve the following synchronization problem of tourists visiting an eruption in times of COVID:

1) Each tourist needs a gas sensor that is handed out at the start of the hike and needs to be returned at the end of the hike: Only 100 gas sensors are available and they are therefore re-used, but need to be disinfected which takes time. Hence, Víðir and Alma decide to share the work of disinfecting and handing out gas sensors in a way that these steps can overlap so that Víðir and Alma can work on the same steps in parallel: each of them takes a sensor from a box, disinfects it and puts it on a single table that is shared by Víðir and Alma. Due to COVID, crowds need to be avoided and therefore, only one sensor will be put on the table and tourists have to wait that a sensor is available so that only one tourist at a time fetches a sensor from the table. (And Víðir and Alma need to wait as well if the table is still in use.) After having put a gas sensor on the table, Víðir and Alma start again their steps, i.e. take the next gas sensor from the box, disinfect it, and put it on the table. Of course, the box can be empty: in this case, they have to wait for a tourist coming back who simply throws the gas sensor into the box (no special COVID measures needed for throwing into the box).

2) Þórólfur is very concerned about a rope used by tourist on the hiking trail that has been set up to ease climbing the hill up and down. He therefore ordered that never more than two tourists are allowed to use the rope at the same time.

Your solution needs to avoid busy waiting and to be obviously free of deadlocks and race conditions. The restriction of the parallelism of the processes needs to be minimized (e.g. do not restrict to only one visitor in the area or do not enforce that Víðir and Alma work in alternating turns: Víðir is still suffering from “Long COVID” and therefore, he is sometimes slower than Alma, so that Alma may do the steps multiple times while Víðir is still busy disinfecting). Provide a pseudocode solution by filling in the skeleton on the next page!

Svar:

```
//Definition of semaphores and their initialisation
Sem
        gasSensor.init(100)
        table.init(1)
        rope.init(2)

VíðirAndAlma: // Two processes, each executing the same code

while(true) {
        var gasSensor gasSensorOffered

        gasSensor.wait()

        gasSensorOffered:=TakeGasSensorFromBox()
        disinfectGasSensor(gasSensorOffered)

        table.wait()

        putGasSensorOnTable(gasSensorOffered)
}


Tourist: // Many processes, each executing the same code

while(true) {
        var gasSensor gasSensorFetched

        gasSensorFetched:=fetchGasSensorFromTable()
        table.signal()

        rope.wait()
        climbRopeUp()
        rope.signal()

        ViewEruption()

        rope.wait()
        climbRopeDown()
        rope.signal()

        ThrowGasSensorIntoBox(gasSensorFetched)
        gasSensor.signal()
}
```

2021, lokapróf, 6, ósvaldur

#safe_state #bankers_algorithm

6) (10%) Processes A, B, and C use a non-sharable, non-preemptable resource type for which 12 instances exist in the system. The processes announce their maximum resource usage as follows: A: max(4), B: max(8), C: max(6). (X: max(n) specifies that process X will never request and use more than n resource instances.)

Then, the processes make the following resource allocation requests that are successfully granted: A: alloc(3), B: alloc(5), C: alloc(3). (X: alloc(n) specifies that process X requests n resource instances with the aim of using them.)

Justify your answer by writing down the initial matrices, (Allocation matrix C, Need matrix N) and vectors (Existing resource vector E, Available resource vector A) of the safety or banker's algorithm, respectively.

(a) (5%) Is the resulting state a safe state?

Now (after the above requests have been granted), process C releases one instance of the resource: C: release(1) and after that, process B requests two further instances of the resource: B: alloc(2).

(b) (5%) Use the banker's algorithm to decide whether the system should grant the above request of process A for two further resources or not. If the allocation request is **not** granted: what action is appropriate on the scheduler level?

Svar við a-lið:
Yes, the resulting state is a safe state since all of the processes can get their needed resources if the order of process schedule is A, B, C after the initial resource allocation. See allocation of resources in the following iterations (https://docs.google.com/spreadsheets/d/1ITx8YWnLo9EDg1BOAqPmiZLEmiBK_jRZYOsH6yftiRI/edit?usp=sharing):

	A	B	C	D	E	F	G
1	**Process and resource assumptions**						
2	Max resource usage M						
3	Process A	4					
4	Process B	8					
5	Process C	6					
6							
7	Existing resources				Available resources		
8	E	12			A	12	
9							
10	**Iteration 1**						
11	Existing resources				Available resources		
12	E	12			A	1	
13							
14	Allocation matrix C				Need matrix N		
15	Process A	3			Process A	1	
16	Process B	5			Process B	3	
17	Process C	3			Process C	3	
18							
19	**Iteration 2**						
20	Existing resources				Available resources		
21	E	12			A	4	
22							
23	Allocation matrix C				Need matrix N		
24	**Process A**	3			**Process A**	1	☑
25	Process B	5			Process B	3	
26	Process C	3			Process C	3	
27							
28	**Iteration 3**						
29	Existing resources				Available resources		
30	E	12			A	9	
31							
32	Allocation matrix C				Need matrix N		
33	Process A	3			Process A	1	☑
34	**Process B**	5			**Process B**	3	☑
35	Process C	3			Process C	3	
36							
37	**Iteration 4**						
38	Existing resources				Available resources		
39	E	12			A	12	
40							
41	Allocation matrix C				Need matrix N		
42	Process A	3			Process A	1	☑
43	Process B	5			Process B	3	☑
44	**Process C**	3			**Process C**	3	☑

Svar við b-lið:
The system should not allocate 2 instances of the resource to process B because that leads to a state where all processes still need resources but there are no available resources (A=0). The scheduler should instead put process B to sleep and search for another route to complete all processes (https://docs.google.com/spreadsheets/d/1ITx8YWnLo9EDg1BOAqPmiZLEmiBK_jRZYOsH6yftiRI/edit?usp=sharing):

Process and resource assumptions									
Max resource usage M									
Process A	4								
Process B	8								
Process C	6								
Existing resources			Available resources						
E	12		A	12					
Iteration 1									
Existing resources			Available resources						
E	12		A	1					
Allocation matrix C			Need matrix N						
Process A	3		Process A	1					
Process B	5		Process B	3					
Process C	3		Process C	3					
Iteration 2									
Existing resources			Available resources			C: release()	1		
E	12		A	0		B: alloc()	2		
Allocation matrix C			Need matrix N						
Process A	3		Process A	1					
Process B	7		Process B	1					
Process C	2		Process C	4					