TENTAMEN (EXAMINATION)

4

Tentamensdatum/*Examination date:* 2019-03-23
(åå-mm-dd/*yy-mm-dd*)

	Ifylles av student	Ifylles av vakt
AID-nummer *AID number*	1655	1655
	Completed by student	*Completed by supervisor*

Kurskod/*Course code:* TDDB68 **Provkod/*Exam code:*** TEN1

Kursnamn/*Course title:* Concurrent Programming and Operating Systems

Institution/*Department:* IDA

Jag intygar att varken mobil eller något annat otillåtet hjälpmedel finns tillgängligt under tentamen.
I confirm that no mobile or other non-permitted aids are available during the examination. ☒

Inlämnat: antal lösblad 11 **tentamensformulär** ☒
Enclosed: number of sheets ***exam booklet***

Markera behandlade uppgifter med X/*Mark tasks attempted with an X*

X här/*here*	1 X	2 X	3 X	4 X	5 X	6 X	7	8	9	10	11	12	13	14	15
Erhållna poäng *Points obtained*	9	10,5	5,5	2	6	3,5									
X här/*here*	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
Erhållna poäng *Points obtained*															

Anvisningar/*Instructions*

1. Skriv AID-nummer, datum, kurskod och provkod på varje blad som lämnas in/
   Write AID number, date, course code and exam code on every sheet that is handed in
2. På varje papper får högst en uppgift lösas om inget annat anges/
   Maximum one task per sheet unless otherwise instructed
3. Skriv endast på papprets ena sida om inget annat anges/
   Use only one side of each sheet unless otherwise instructed
4. Numrera de papper som lämnas in/*Number every sheet that is handed in*
5. Använd inte röd penna/*Do not use a red pen/pencil*

Sen inlämning ☐
Late hand in

Klockslag _____
Time

Orsak _____
Reason

_____

∑ Poäng/*Points:* 36,5+4=40,5 **Betyg/*Grade:*** 5

Examinator/*Examiner:* _____

LINKÖPINGS UNIVERSITET

AID : 1655 Date: 2019-03-23 Sheet number: 1.

Course code: TDDB68 Exam code: TEN1

Multiple choice form for answering question 1. Please put X:s in the appropriate cells:

	A	B	C	D
1 a)	X		X	X
1 b)	X	X		X
1 c)			X	
1 d)	X		X	X
1 e)	X			X
1 f)	X			
1 g)	X		X	X
1 h)	X		X	
1 i)	X	X		
1 j)	X			X

1 1 0 1 1 0 1 1 1 1

/9

Optional: if you feel you need to clarify your interpretation of the question you can do so here. This is not needed if your answer is correct.

1 a)	
1 b)	
1 c)	
1 d)	
1 e)	
1 f)	
1 g)	C: They need special instructions, e.g. ~~test~~ test_and_set or atomic de-/incrementation
1 h)	
1 i)	
1 j)	

↓

No [redacted] busy wait and e.g. Petersons alg. can do the trick.

LINKÖPINGS UNIVERSITET

AID-nummer: *AID-number:* 1655	Datum: *Date:* 2019-03-23	Blad nummer: *Sheet number:* 2.
Kurskod: *Course code:* TDDB68	Provkod: *Exam code:* TEN2	

2. a) Yes. If there are no ideas in the buffer, the while-loop on line 36-38 busy-waits until there is at least one idea in the buffer. 1/1

b) Add struct semaphore idea_count; to the idea_buffer struct. In idea_init, add sema_init(&buffer->idea_count, 0); before or after the for-loop. Add sema_signal(&buffer->idea_count); before return true in idea_add. Finally, add sema_wait(&buffer->idea_count); at the very start of idea_get. With these changes, the internal counter of idea_count will keep track of the number of unused ideas. Idea_get will yield the cpu if there are no available ideas. 2/2

c) Assume that buffer->ideas[5] != NULL. Thread t1 and t2 simultaneously finds this idea and proceeds to line 40, overwrites the idea simultaneously and both sets buffer->ideas[5] to NULL. This is an example of A, assuming a multiprocessor system. Note that the semaphore added in b) does not prevent this issue if there are >1 ideas in the buffer.

Situation B might have occured in a similar way by two threads simultaneously finding the same empty position on line 19. One thread would then have overwritten the other one's. Since both would signal idea_count, this might cause another thread to once again busy-wait on lines 36-38 while looking for the missing idea. 2/2

2. d) No more then one thread may execute lines 19-22 for the same i/found. Similarly, only one thread may execute lines 36-41 for a given pos. It is assumed that idea_init will only be called once and hence need no synchronization.

From this we can deduce that each position in buffer -> ideas must be protected from concurrent access by more than one thread running idea_add or idea_get. The semaphore operations are assumed to be atomic and thus (access to) idea_count needs no synchronisation. /2

e) In struct idea_buffer add

```
struct lock add_lock[BUFFER_SIZE];
struct lock get_lock[BUFFER_SIZE];
```

In idea_init add

```
lock_init(&buffer->add_lock[i]);
lock_init(&buffer->get_lock[i]);
```

inside the for-loop. Add

```
lock_acquire(&buffer->add_lock[i]);
```

on line 19 and

lock_release(&buffer->add_lock[i]);

on lines 23 and 25. Add

lock_acquire(&buffer->get_lock[pos]);

on lines 36 and 38. Finally, add

lock_release(&buffer->get_lock[pos]);

on lines 37 and 42.

The additions outlined above eliminates the issues described in a). Note that no issues will arise from allowing threads (executing idea_add) concurrent access with threads executing idea_get, as the NULL-conditions ensure these threads won't enter lines 21-22 and 40-41 simultaneously for the same idea.

This approach uses a lot of locking but allows high concurrency. If lock operations involves a context switch or other high overhead operation, the overhead might be unacceptable. However, since the locks are only held for a short time, spin-locks might provide high performance despite the many lock operations.

Not entirely true. Accessing a variable like this is generally undefined as you may try to read and write concurrently. -0,5

3,5

3. a) Assume there are n processes in a system. A state is defined as safe if there exists an ordering of the n processes such that any resource request required for the i:th process to finish can be satisfied when process 1..o(i-1) has finished and consequently released any held resources.

Deadlock avoidance is achieved by monitoring the system state and only granting resource requests that would transition the system to another safe state. 1,5p

b) The requested resources are available and P4 has not exceeded its maximum number of resources required to finish. We pretend we have granted the request and examines if the resulting state would be safe. We define finish as a vector describing if each process can finish and initialize it to false. The state can then be described as ✓

	R1	R2	R3
P1	I(1)	O(3)	I(1)
P2	O(2)	O(1)	I(4)
P3	I(1)	O(0)	I(1)
P4	O(1)	I(3)	O(1)

Available = [0, 2, 5] ✓

finished = [F, F, F, F]

P3 could be allocated all its required resources, so it can finish. When P3 has finished, we would be in the following state

	R1	R2	R3
P1	1(1)	0(3)	1(1)
P2	0(2)	0(1)	1(4)
P3	0(1)	0(0)	0(1)
P4	0(1)	1(3)	0(1)

Available = [1, 2, 6] ✓

Finished = [F, F, T, F] ✓

Next, we find that $Finished_4$ = F and P4 can be allocated all its required resources. The state after P4 has finished is as follows

	R1	R2	R3
P1	1(1)	0(3)	1(1)
P2	0(2)	0(1)	1(4)
P3	0(1)	0(0)	0(1)
P4	0(1)	0(3)	0(1)

Available = [1, 3, 6] ✓

Finished = [F, F, T, T] ✓

In the same way we can now allocate all required resources to let P1 finish, and once P1 has finished so can P2. Thus (P3, P4, P1, P2) is a sequence described in a), and the new state is safe. We may thus grant the request from P4. ✓

4

4. i) I assume that new processes are put at the back of the queue and that at the end of each time quanta, the running process is preempted even if it shared the quanta with another process. The Gantt chart is as follows

why 1 run for 2nd time after P4 and P5?

t=	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	1	1	2	2	3	3	4	5	1	3	5	5	3	3

At t=2, P1 is preempted by P2. At t=4, P2 finishes and leaves room for P3. At t=6, P3 is preempted by P4, which finishes at t=7 leaving room for P5. P5 is preempted by P1 at t=8, which finishes at t=9 leaving room for P3. At t=10 P3 is preempted by P5. P5 finishes at t=12, leaving room for P3, which finishes at t=14.

/6

ii) The Gantt chart is as follows:

t=	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	1	1	1	2	2	4	5	5	5	3	3	3	3	3

✓

When P2 arrives at t=1, P1 and P2 has the same time remaining. P1 is not preempted. At t=3 P1 has finished and P2 is now the shortest job. P2 and P4 are tied at t=4, so P2 is allowed to continue. At t=5 P2 has finished and P4 is now the shortest job. At t=6 P5 is shortest, and P3 finally starts running when P5 finishes at t=9,

/1

4. iii) The Gantt chart is as follows:

t=	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	1	1	1	2	2	4	5	5	5	3	3	3	3	3

✓

Since no process is preempted, P1 is allowed to run starting at t=0 until it finishes at t=3, despite has a higher priority and arrived at t=1. At t=3, P2 has the highest priority and finishes at t=5. P4 has higher priority than P3, so it runs until it finishes at t=6. P5 has now arrived and has higher priority than P3. P3 gets to run once P5 finishes at t=9.

1

total 2

5. a) Fragmentation:

Paging suffers from internal fragmentation. This is the result of the operating system always granting an entire page at a time, even if the process only requires a single byte of additional memory. On average, we expect to lose 1/2 page per process due to internal fragmentation.

Segmentation avoids internal fragmentation but instead causes external fragmentation. The processes are only allocated the memory they actually need, but when processes release memory, this might cause "holes" of free memory that are hard to allocate to other processes.

Memory overhead:

Traditional paging requires that each process has a page table containing the physical address of each virtual address. For large virtual address spaces and/or small page sizes, this requires unacceptable amounts of memory. For example a 32-bit virtual address space and 1024-byte pages requires $3 \cdot 2^{22} = 12\text{MB}$ of memory, if the physical addresses are also 32 bits. To alleviate this problem, multi-level paging, where the page-table is itself paged, or an inverted page table, where the PID and virtual page number for each physical page is stored in a global structure, might be used.

AID-nummer: *AID-number:* 1655	Datum: *Date:* 2019-03-23	Blad nummer: *Sheet number:* 10.
Kurskod: *Course code:* TDDB68	Provkod: *Exam code:* TEN 1	

LINKÖPINGS UNIVERSITET

5. cont...

a) These techniques might however cause decreased performence for address lookup.
Segmentation might be more memory efficient than paging if the allocated segmets are large. An 8 kB segment only requires one segment table entry, while it would require 8 entries in a 1024-byte paging scheme. However, for small segments the reverse is true. Keeping track of unused memory also requires larger amounts of memory as fragmentation increases. A segment of 4 free bytes might require 8 bytes to keep track of: 4 bytes for the segment start address and a 4 byte integer to store the segment size.

4/

b) The fork system call is commonly followed by an exec system call, where the child starts executing a different program than its parent. Without shared pages, the fork system call would require allocating new pages to the child and copying the contents of the parent's pages to the child's. This consumes substantial time, and might be completely wasted if the child immediately calls exec. Page sharing allows the OS to only copy pages once its absolutely needed, saving start-up time.

2/

6. a) The main purpose is to improve the performance of the file system. Finding a file on disk is an expensive operation, especially as it might require multiple disk accesses. Requiring that files are opened before use means that the OS only has to do this once. For subsequent reads/writes, the OS can consult its saved file information and quickly find the files location on disk. Additionally, an opened file is likely to soon be read. This information can be used by the OS to provide caching of opened files, greatly improving file system performance.

The open system call modifies the OS list of open files and possibly the file system cache. It returns a file handle/descriptor used to identify the file for subsequent file operations. 1.5

b) This vulnerability compromises the system integrity by allowing Mallory to edit the inode without being authorized to do so.

If Mallory edits the inode information regarding the file's location on disk, she might be able to write to or read from files she normally would not have access to. This in turn compromises the confidentiality of the file system. If the disk contains swap space, she might even be able to access memory belonging to other processes. 2p /3,5p