AID: 1835 Datum: 17-06-07 Blad nummer: 1

Kurskod: TDDB68 Provkod: TEN1

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

	A	B	C	D	
1 a)	X		X		1
1 b)			X		1
1 c)	X	X			1
1 d)	X		X		0
1 e)	X	X	X	X	0
1 f)	X				1
1 g)		X	X		1
1 h)	X		X		0
1 i)	X	X			1
1 j)	X	X	X	X	1

/7p

2.a) If two processes execute the part of the "book" function which loops through all seats concurrently and find the same free seat (i.e. found_k remains 1 for a certain k) before either has had time to actually change the array values, the seat will be double-booked. (1)

b) In "book", the critical section is the whole content of the outer foor loop (excluding the declaration of "found_k"), since if a seat is found to be free it must be set as occupied before the process can be preempted to prevent situations like in 2.a)

In "cancel", the critical section is ~~the line inside the for loop.~~ the entire loop (1.5)

2.c) The single mutex lock is declared outside the functions to be accessible within both functions.

```
lock matrix_lock;          // assumed to be initialized somewhere
int book(unsigned int i, unsigned int j)
{
  int k, t;
  for (k=0; k<5; k++) {
    int found_k = 1;
    lock_acquire(&matrix_lock);
    for(t=i; t<=j; t++) {
      if(seat[k][t]) {
        found_k=0;
        lock_release(&matrix_lock);
        break;
      }
    }
    if (found_k) {
      for(t=i; t<=j; t++)
        seat[k][t]= 1
      lock_release (&matrix_lock);
      return k;
    }
  }
  return -1;
}
```

2.c) (Continued)

```
void cancel(unsigned int k, unsigned int i,
            unsigned int j)
{
    int t;
    lock_acquire(&matrix_lock);
    for (t=i; t<=j; t++)
        seat[k][t]=0;
    lock_release(&matrix_lock);
}
```

1

Comment: the two release calls in "book" are needed to cover both ways that the outer for loop can exit. The one right above the "break" statement could have been placed at the end of the outer for loop, but this way decreases the time holding the lock.

2.d) The cancel-function cannot be made more fine-grained.

As for book, I cannot find a way to make the critical sections smaller either, since the lock must be held both when reading the current values and writing to them. With my implementation in 2.c), I already release the lock as soon as found_k is set to 0, at which point we know that the array for this seat will not be written to.

Something else I could imagine increasing the performance of the system however would be having separate locks for each seat, drastically reducing the time waiting for other processes.

1.5

Code is missing!

2.e) A reader-writer lock is a mechanism where the "read" operation can be executed concurrently by multiple threads, but "read" and "write" cannot be executed concurrently and neither can "write" and "write".

These are suitable when reading and writing files since allowing concurrent reads increases throughput while causing no harm since reading does not change the file.

1.5

3.a) The extra field will store the ID of the process/thread currently holding the lock. Let's call this field "holder_id" ant let it be initialized to -1:

```
• lock_acquire(mutex_lock *lock) {
      if (lock->holder_id == own_id){
          //Is ok to enter critical section!
          return;
      } else {
          sema_down(&lock);
          //when we get here, has
          //access to critical section
          lock->holder_id = own_id;
      }
  }
```

raise exception

or &lock->sema or however the semaphore is accessed

2.5 → 3

```
• lock_release(mutex_lock *lock){
      //Set to default in the case
      // of no waiting threads
      lock->holder_id = -1;
      sema_up(&lock);
  }
```

unblocked thread will set holder_id here!

3.a) (Continued)

This prevents self-deadlocks since a thread already holding the lock will not call sema_down and thus not be blocked.

3.b) The request should not be granted. This is because processes 1, 2 and 3 will not be able to satisfy their remaining needs ([3, 1, 0] for P1, [0, 0, 2] for P2 and [3, 0, 1] for P2) with the remaining available resources [0, 0, 1]. Even though P4 which needs [0, 0, 1] will be able to have its need satisfied, it will not help the other processes since no more resources than that single instance of R3 would be freed up. Since granting the request means there is no possible execution sequence where all processes terminate, the request should not be granted.

(4)

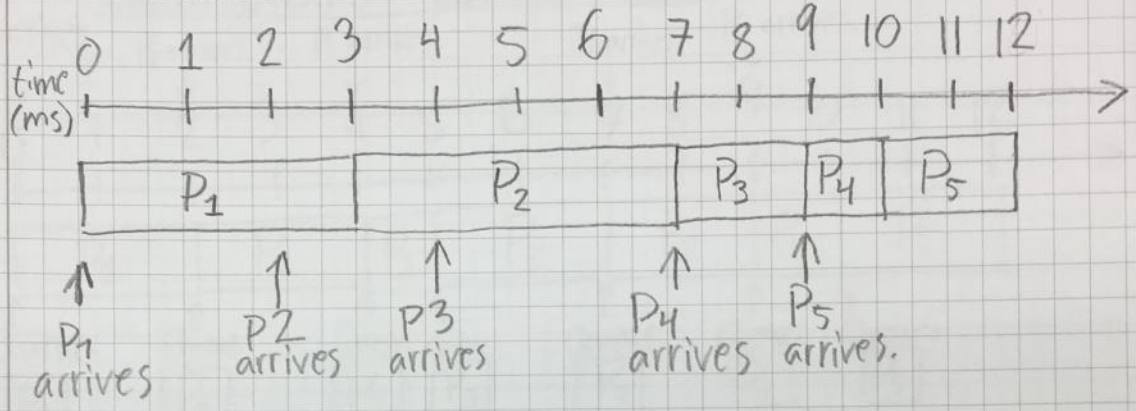
4.a) The PCB stores information about currently active (not necessarily running at the moment) processes, and is used when switching between running processes e.g. to switch out the content of CPU registers and program counter when a new process is to be run. It stores, amongst other things,:

- Process status informations
- CPU register contents for process
- Program counter
- Scheduling information
- List of open files

2p

4.b)

- FIFO:



With FIFO, processes are simply executed in the order they arrive, and this particular sequence gave no gap between them.

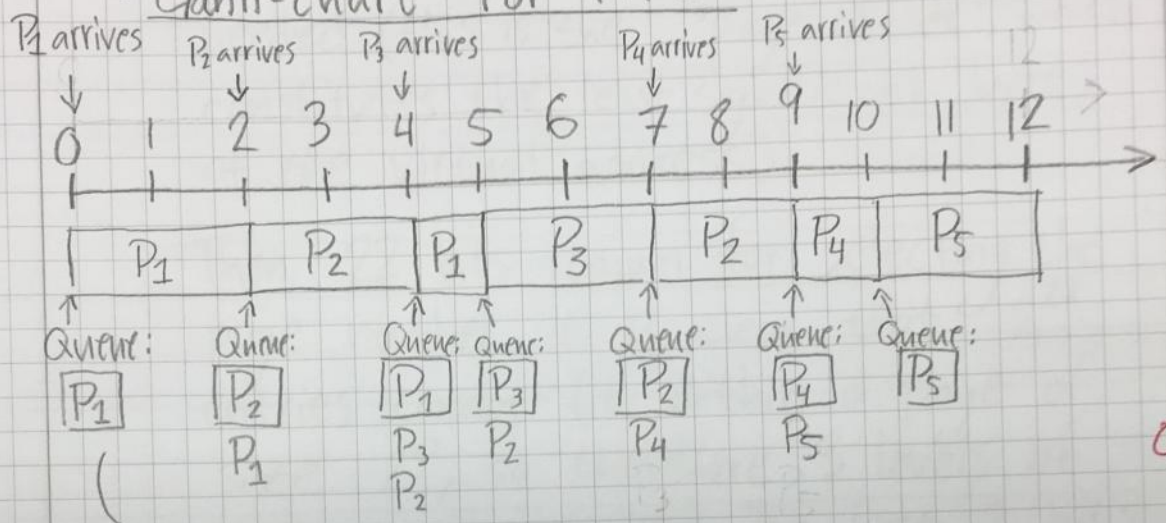
- Round-robin: here, each process is allowed to run for 2ms before being placed at the back of the queue. I assume that if a new process arrives at the same time as another is preempted, the new process will be placed before the preempted process in the queue.

Ok

(Continued on next page)

4.b) (Continued)

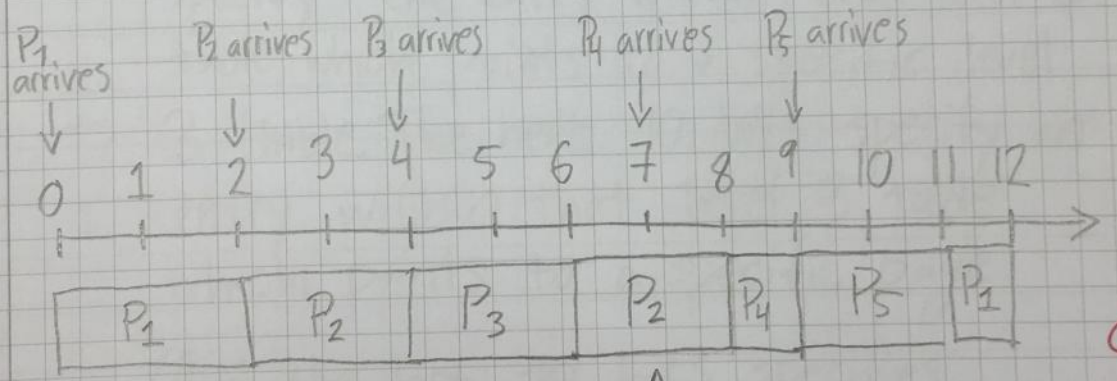
- Gantt-chart for RR:



06

(squared process is the one first in queue)

- Priority-based scheduling (with preemption)



06

The available process with highest priority is always chosen to run first.

/3p

5.a) To address a specific byte among the 2^{10} that are in a page, 10 bits are needed (this is the "page offset" which constitutes the least significant bytes of the virtual address).

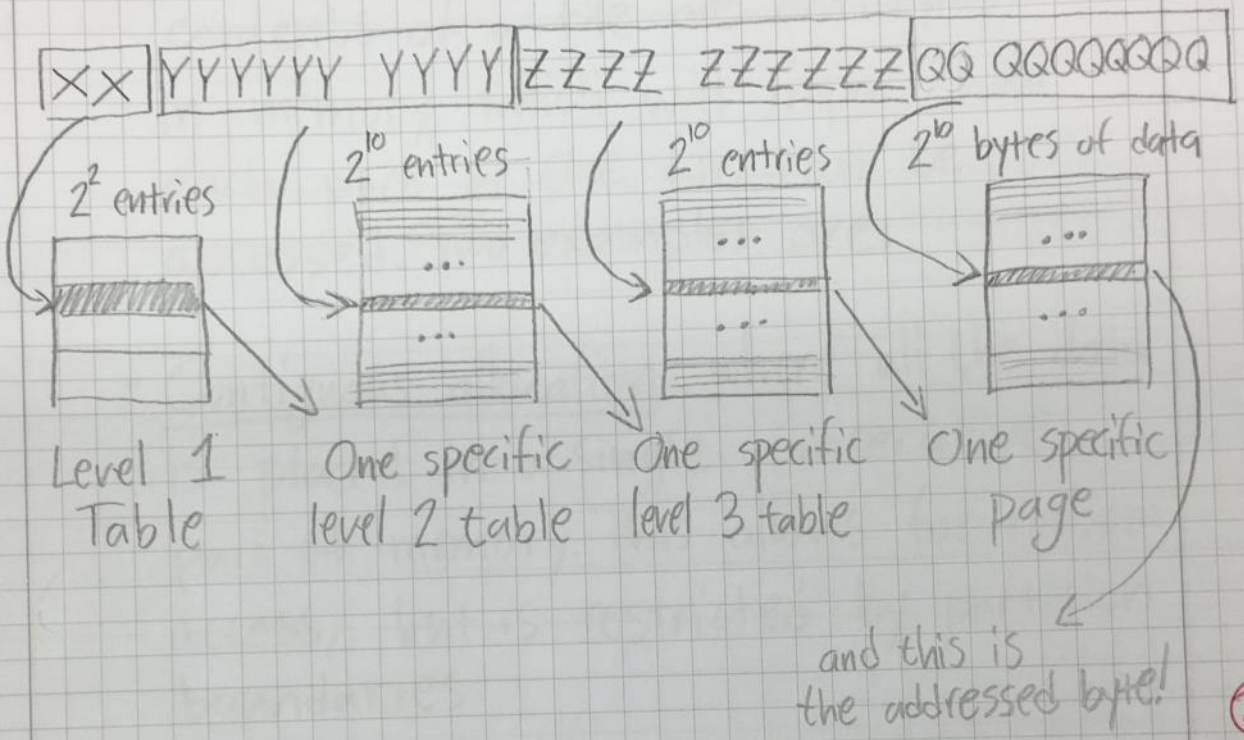
Then, 22 bits remain for selecting the page number. It is clear that 2^{22} entries cannot fit within a page of size 2^{10}, so therefore 3 levels of paging are required. The first 2 bits (most significant) specify an entry in the Level 1 table which in turn points to a specific level 2 table. The next 10 bits specify the entry within that level 2 table which points to a specific level 3 table. The next 10 bits specify an entry within that level 3 table which points to a specific page.

(And the last 10 bits specifies which byte in the page is addressed)

2p

5.b)

Virtual address: (where each letter is a 0 or 1)



2p

c) Because pages are fixed-size, internal fragmentation will occur when the whole page is not needed by e.g. a process.

Also, you could probably say that the unused entries that can exist within the page tables could constitute internal fragmentation as well.

1.5p

6.a) Allocation in the context of a file system describes how the contents of a file are distributed in memory, which will impact how each part of the file can be accessed. Two examples are:

- Contiguous allocation, where all the data is placed in a single consecutive part of memory. This allows for random access, but is restricted by partition boundaries.
- Indexed allocation, which stores pointers to different parts of the file in a list. This allows for freedom in where different parts of the file are stored, but adds overhead to accesses because the pointers must first be taken from the list.

3p

6.b) The "open" system call serves the purpose of moving the contents of a file from secondary storage to main memory for faster access.

The "open" system call manipulates the "inode" data structure in that it uses the inode to find the physical location of the file, and the inode also updates its stored information about how many/which processes have opened that file.

The call returns a file descriptor, which is an integer that points to the position of the file in the table of open files that the process has. This file descriptor is then used as a parameter for system calls that access the file contents, such as "read" and "write".

1P