Solution

Problem 1:

1 [1] policy [2] mechanism

[3] policy [4] mechanism

2 [1] A [2] A [3] B [4] B

[5] B [6] B [7] B [8] B

[9] A [10] A [11] B [12] B

3

1) Achieve lower turnaround time without a priori knowledge of job length.
2) More friendly to short jobs.

(Any reasonable answer would be right)

Problem 2:

1. [1] 3,4,5 [2] N [3] 4,5,2 [4] N

[5] 5,2,1 [6] N [7] 5,2,1 [8] Y

2. [9] 2,3,4 [10] N [11] 4,2,5 [12] N

[13] 4,5,2 [14] Y [15] 5,2,1 [16] N

[17] 2,1,5 [18] Y

3. 4/9

Problem 3:

1 [1] 16 [2] 8K [3] 13 [4] 19 [5] 2

2 [1] N [2] Y [3] -- [4] --

[5] Y [6] N [7] 0x366d03 [8] Y

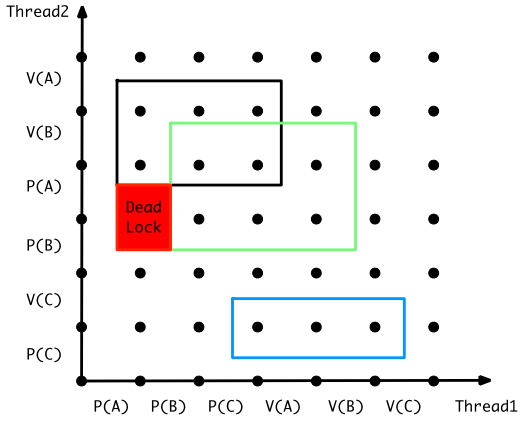
3

No. Because TLB flushes and L1 page table base address in CR3 changes upon context switch, it is very likely that the same virtual address does not map to the same physical address.

(Although there's still possibility that the mappings happen to be the same, such as in the case of shared mmap)

Problem 4:

1 Yes.



The rectangles are unreachable regions for this program. If the program goes into the DEADLOCK region, it can never go out, since the program can only move up or move right in this graph.

2 [1] 1 [2] 2 [3] 1

[4] P(a) [5] P(c) [6] V(c)

[7] P(b) [8] P(c) [9] V(c)

Problem 5:

1. 1) In the modified ticket lock, a lock waiter will idle (instead of always spinning in the original ticket lock) some amount of time proportional to his distance between the lock holder based on the fact that late-comers will wait more time before the lock is passed to them. This can greatly reduce the number of memory access from the lock waiters.

   2) However, it is hard to estimate an appropriate idle time since the time of critical section is unknown and variant. It may slow the whole queue if one of the waiter's idle time is too long.

   3) In ticket lock, the time a thread will wait on the lock is proportional to his distance between the lock holder. So IDLE(myturn - lock->turn) saves memory access with respect to his waiting time.

2. 1) The 5 threads would wait forever.

2) The original intention of using barrier is to make sure that no threads would pass through the barrier if there is someone lags behind. However, if the barrier is initialized with total = 4, then 4 threads can pass through the barrier even if there is 1 thread having not reached the barrier.

3) line 15 updates b->local_sense to let other waiting threads know all threads have reached barrier_wait and they can proceed now. If it is removed, other waiting threads will not go on.

4) line 14 restores b->count so that this barrier can be re-used next time.

5) No. Considering the following case: if threads are synchronized twice using this barrier. During the first synchronization, after the last-comer (say T5) setting b->sense but before b->count, other waiting threads (T1-T4) can proceed to the next barrier and FetchAndAdd the b->count. Then T5 sets b->count to 0. Then they will all stuck at the second synchronization.

6) Here is a possible solution. Before fork, parent mmap a SHARED, ANONYMOUS memory region:

```
m = mmap(NULL, 4096, PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
```

Then place the barrier at this memory region:

```
barrier_t *b = (barrier_t *)m; barrier_init(b);
```

Then after fork, barrier_wait(b) in parent and child will be synchronized.