CS347: Operating Systems

Problem Set 2: Solutions

1. (a) Semaphores variables:

```
pt_waiting = 0
treatment_done = 0
doc_avlbl = 1
```

   (b) Patient process:

```
down(doc_avlbl)
consultDoctor()
up(pt_waiting)
down(treatment_done)
noteTreatment()
up(doc_avlbl)
```

   (c) Doctor:

```
while(1) {
down(pt_waiting)
treatPatient()
up(treatment_done)
}
```

2. (a) Producer:

```
int produced = produceNext();
shptr->field1=produced;
shptr->field2 = 1; //indicating ready
while(shptr->field2 == 1); //do nothing
```

   (b) Consumer:

```
while(shptr->field2 == 0); //do nothing
consumed=shptr->field1;
consumeNext(consumed);
shptr->field2 = 0; //indicating done
```

3. `sem lock = 1; sem writer_can_enter = 1; int readCount = 0;`

```
readLock:
down(lock)
readCount++
if(readCount ==1)
  down(writer_can_enter) //don't coexist with a writer
up(lock)

readUnlock:
down(lock)
readCount--
if(readCount == 0)
  up(writer_can_enter)
up(lock)

writeLock:
down(writer_can_enter)

writeUnlock:
up(writer_can_enter)
```

4. (a) Read lock

```
lock(mutex)
while(writer_present || writers_waiting > 0)
    wait(reader_can_enter,mutex)
readcount++
unlock(mutex)
```

(b) Read unlock

```
lock(mutex)
readcount--
if(readcount==0)
    signal(writer_can_enter)
unlock(mutex)
```

(c) Write lock

```
lock(mutex)
writer_waiting++
while(readcount > 0 || writer_present)
    wait(writer_can_enter, mutex)
writer_waiting--
writer_present = true
unlock(mutex)
```

(d) Write unlock

```
lock(mutex)
writer_present = false
if(writer_waiting==0)
    signal(reader_can_enter)
else
    signal(writer_can_enter)
```

5. The accounts must be locked in order of their account numbers. Otherwise, a transfer from account X to Y and a parallel transfer from Y to X may acquire locks on X and Y in different orders and end up in a deadlock.

```
struct account *lower = (from->accountnum < to->accountnum)?from:to;
struct account *higher = (from->accountnum < to->accountnum)?to:from;
dolock(&(lower->lock));
dolock(&(higher->lock));

from->balance -= amount;
to->balance += amount;

unlock(&(lower->lock));
unlock(&(higher->lock));
```

6. If one acquires multiple spinlocks (say, while serving nested interrupts, or for some other reason), interrupts should be enabled only after locks have been released. Therefore, the push and pop operations capture how many times interrupts have been disabled, so that interrupts can be reenabled only after all such operations have been completed.

7. No, this design can have starvation. To fix it, keep a pointer to where the wakeup function stopped last time, and continue from there on the next call to wakeup.

8. One possible place is the scheduler code itself: while going over the list of processes, it can identify and clean up zombies. Note that the cleanup cannot happen in the exit code itself, as the process memory must be around till it invokes the scheduler.

9. Sleep continues to hold ptable.lock even after releasing the lock it was given. And wakeup requires ptable.lock. Therefore, wakeup cannot execute concurrently with sleep.

10. If marked runnable, another CPU could find this process runnable and start executing it. One process cannot run on two cores in parallel.

11. It releases ptable.lock and preserves the atomicity of the context switch.

12. (a) Cache locality. No contention for the common queue.
    (b) Better load balancing across cores.

13. A

14. BC

15. D

16. B