1. **T/F**
    a. False. Threads share an address space.
    b. False. A binary semaphore can be released by a thread that does not own it.
    c. True.
    d. True.
    e. False. Processes have a single address space.
    f. True.
    g. True.
    h. False. Single level page tables can be any size.
    i. True.
    j. True.
    k. True.
    l. False. Only the running process.

2. **Short Answer**
    a. Does not succeed. 9382 is not a child of process A, and waitpid can only be called on a child process.

    b. If the wait channel is locked after releasing the spinlock, it would be possible for the thread which owns the lock to release the lock and wake up any waiting threads before other threads waiting to acquire are able to block. These other thread could be blocked forever.

    c. No. Because no changes to the variable is ever made.

    d. Steps of cv_wait
        i. Check if you own the lock
        ii. Lock the wait channel
        iii. Release the lock
        iv. Wchan_sleep
        v. Acquire the lock

3. **Stacks**

User	Kernel
getpid	Trapframe Mips_trap Syscall Sys_getpid Trapframe Mips_trap Mainbus_interrupt Timer_handler Thread_yield Thread_switch switchframe

4. Deadlock
    a.

```
acquire( lockA )
while()
    If ( !try_acquire( lockB ) )
        release( lockA )
        acquire( lockA )
        Continue
    If ( !try_acquire( lockC ) )
        release( lockA )
        release( lockB )
        acquire( lockA )
        Continue
    Else
       Break

FUNCTION CODE

release( lockA )
release( lockB )
release( lockC )
```

b.

```
// Assign each lock and queue a number

Minlock = min( a.num, b.num, c.num )
Middlelock = middle( a.num, b.num, c.num )
Highlock = max( a.num, b.num, c.num )

acquire( minlock )
acquire( middlelock )
acquire( highlock )

FUNCTION CODE

release( highlock )
release( middlelock )
release( minlock )
```

5. **Virtual Memory**
    a. 2^32 bytes
    b. 2^24 bytes
    c. 2^24/2^10 = 2^14 entries
    d. 14
    e. 10
    f. 2^3*2^14=2^17 bytes
    g. Addresses

    32 bit physical address with 1KB frames
    24 bit virtual address with 1KB pages
    There are 2^32/2^10 frames = 2^22
    There are 2^24/2^10 pages = 2^14
    22 bits are used for frame number, 14 for the page number
    Page offset 10 bits.

    i. 16
    16 KB / 1KB = 16 pages

    ii. Exception
    0xA5A5A5 = 1010 0101 1010 0101 1010 0101
    Pg. Num = 10 1001 0110 1001 = 0x3969
    But, only the first 16 pages are valid -> exception

    iii. 0xC16
    Virtual Address 0x16 = 0000 0000 0000 0000 0001 0110
    Page number = 0000 0000 0000 00 = 0x0
    Page offset = 00 0001 0110
    Frame number = page number + 0x3 = 0x0 + 0x3 = 0x3
    = 0000 0000 0000 0000 0000 11

    Physical Address = 0000 0000 0000 0000 0000 1100 0001 0110 =
    0xC16

    iv. 0x8A6
    Physical Address
    0x14A6 = 0000 0000 0000 0000 0001 0100 1010 0110
    Frame number = 00 0000 0000 0000 0000 0101 = 0x5
    Page number = 0x5 - 0x3 = 0x2

    Offset = 00 1010 0110 = 0xA6

Virtual Address = 0000 0000 0000 1000 1010 0110 = 0x8A6

v. Exception
Physical Address = 0x1000 0000
Frame number > 16 + 3 -> exception

6. **Thread counts**
    a. 10

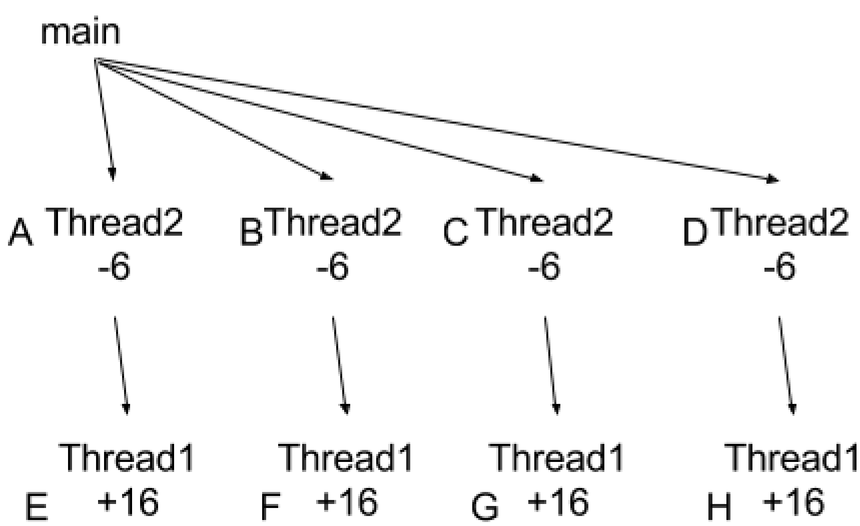
    b. No

    c. A global barrier semaphore can be used. After main loops (and forks threads), P the barrier semaphore 9 times. Thread1 and Thread2 both need to V the barrier semaphore just before returning.

    It is also possible to use a global counter with a lock and cv.

    d. 16, 0, 34

    Main forks FOUR Thread2 with value 4
    Each Thread2 WITHDRAWS 0 + 1 + 2 + 3 = 6 (subtracts 6 from balance), then
    Thread2 then forks ONE Thread1
    Each Thread1 DEPOSITS 4 + 4 + 4 + 4 = 16 (adds 16 to the balance)
    Thread1 does not fork



16: Execution order A, B, C, D. Balance = -24. While the first iteration of DEPOSIT runs for E, F and two iterations of G execute and are overwritten when we context switch back to E. Balance is now -20. The remainder of E executes, leaving balance at -8. Then the remainder of G executes. Balance is now 0. Then H executes. Balance is now 16.

0: Execution order A, B, C, D. Balance = -24. While the first iteration of DEPOSIT runs for E, F, G and 2 iterations of H execute and are overwritten when we context switch back to E. Balance is now -20. E finishes. Balance = -8. Then, H finishes. Balance = 0.

34: Execution order A, B, C, E, F. Balance = 14. Before G can update balance with the first DEPOSIT of 4, D executes, then 3 of the 4 loops for H. At this point, G updates balance to 18 --- overwriting any work done by D or H. G completes, and balance is now 30. H’s last DEPOSIT loop executes, leaving balance at 34.

7. VMem

a. 2

b. 3 page table base registers (one for each segment) and a single limit register (since all segments have the same maximum size).

c. After extracting the segment offset, check if that offset is less than the limit. If so, extract the page number from the offset and check if the valid bit for that page number is set in the PTE.

d. Change the page table base registers on the MMU to the page table addresses for the new process.

e. PTE requires a Read-Only/Dirty bit. MMU must check this bit on write-attempt and throw exception. Exception must be handled by kernel.

   OR, the MMU has an extra register for each segment indicating if that segment is read-only which is checked on a write.

8. Sys_kill
    a. Steps
        i. Sys_kill takes PID as param
        ii. Check if PID exists; if not return error
        iii. Check if process with PID is alive; if not return error
        iv. Terminate process with given PID; remove running threads and delete address space.
        v. If parent of process alive
            1. set exit code to KILLED andsignal parent to wake if it is blocked on PID
        vi. Otherwise, delete entire process structure and allow PID to be reused.

    b. A table of users and their access rights (i.e., are you root or not).

    c. The user id of the user that created the process

    d. Changes:
        i. Check if owner (user) of the process that called kill has root access or is the same owner (user) of the process to be killed
            1. If yes, terminate process
            2. If no, return error
        ii. No other changes required.