This exam is closed book, no materials except for one 8 × \( \frac{1}{2} \) inch paper sheet of notes. Incomprehensible answers get zero points! Precise answers that demonstrate clear understanding earn more points than “logically” correct but vague or uninformative answers. All technical words signify their meanings given in this subject.

**Part SA**

A system call crosses the user/kernel protection boundary but an ordinary function call remains on the user side of that boundary. This summarizes their difference.

In x86 Linux, a system call (from user space) is performed by an `int` (interrupt) or a `syscall` instruction, but an ordinary function call is performed by a `call` instruction.

Right after the `int`, `syscall` or `call` instruction is executed, the program counter \( %eip \) is set to a address different (see below for an unusual case when this isn’t strictly true) from the address of the instruction right after the `int`, `syscall` or `call` instruction. That address is called the **destination address**.

Describe the role(s) of the user code and/or OS software in determining the destination address,

(1) of an ordinary function call

**USER CODE DETERMINES THE DEST.** \( \text{It is an address} \)

(2) system call

(3) The code below is written in assembly language. The assembler makes a label like **LabelA** into a symbol and puts it into the symbol table. The symbol table stores what each symbol symbolizes or means.

What does a symbol like **LabelA** symbolize? ____________________________

Unusual code which finds the address of an instruction by pushing it onto the stack:

```assembly
call LabelA
LabelA: popl %eax /*Now, the address of the popl instruction is in register eax*/
```
This exam is closed book, no materials except for one $8 \times \frac{1}{2}$ inch paper sheet of notes. Incomprehensible answers get zero points! Precise answers that demonstrate clear understanding earn more points than “logically” correct but vague or uninformative answers. All technical words signify their meanings given in this subject.

Part SA

A system call crosses the user/kernel protection boundary but an ordinary function call remains on the user side of that boundary. This summarizes their difference.

In x386 Linux, a system call (from user space) is performed by an int (interrupt) or a syscall instruction, but an ordinary function call is performed by a call instruction.

Right after the int, syscall or call instruction is executed, the program counter %eip is set to a address different (see below for an unusual case when this isn’t strictly true) from the address of the instruction right after the int, syscall or call instruction. That address is called the destination address.

Describe the role(s) of the user code and/or OS software in determining the destination address,

(1) of an ordinary function call

(2) system call

(3) The code below is written in assembly language. The assembler makes a label like LabelA into a symbol and puts it into the symbol table. The symbol table stores what each symbol symbolizes or means.

What does a symbol like LabelA symbolize? ________________________________

Unusual code which finds the address of an instruction by pushing it onto the stack:

call LabelA
LabelA: popl %eax /*Now, the address of the popl instruction is in register eax*/
This exam is closed book, no materials except for one 8 × ½ inch paper sheet of notes. Incomprehensible answers get zero points! Precise answers that demonstrate clear understanding earn more points than “logically” correct but vague or uninformative answers. All technical words signify their meanings given in this subject.

Part SA

A system call crosses the user/kernel protection boundary but an ordinary function call remains on the user side of that boundary. This summarizes their difference.

In x86 Linux, a system call (from user space) is performed by an int (interrupt) or a syscall instruction, but an ordinary function call is performed by a call instruction.

Right after the int, syscall or call instruction is executed, the program counter %eip is set to a different address (see below for an unusual case when this isn’t strictly true) from the address of the instruction right after the int, syscall or call instruction. That address is called the destination address.

Describe the role(s) of the user code and/or OS software in determining the destination address,

(1) of an ordinary function call

(2) system call

(3) The code below is written in assembly language. The assembler makes a label like LabelA into a symbol and puts it into the symbol table. The symbol table stores what each symbol symbolizes or means.

What does a symbol like LabelA symbolize?

Unusual code which finds the address of an instruction by pushing it onto the stack:

call LabelA
LabelA: popl %eax /*Now, the address of the popl instruction is in register eax*/
Part PP

The Unix (Posix) `fork()` system call adds a new process.

(4) What is the program that the new process runs?

The `fork()` system call, like any other system call, is performed (or executed) by a particular user space thread. Each thread has its own, private stack within the virtual memory of that thread’s process.

(5) What happens to that stack when the `fork()` is performed?

The Unix (Posix) `exec()` system call changes the program that the current process runs.

(6) What determines the new program?

(7) The `exec()` too is performed by a particular thread. What happens to that thread’s own, private stack?

Part SB

The semaphore is a popular synchronization and mutual exclusion primative often described as having a `count` variable that is always either zero or positive, never negative.

The `down()` (or `get()`, `take()` or `P()`) subtracts 1 if and only if the value of count is > 0.

(8) The implementation of `down()` makes the thread calling `down()`:

- **× sometimes** but **not always** sleep (or block or wait).
- **always** sleep (or block or wait).
- **never** sleep (or block or wait*) during the call.

(*It might busy-wait for a very short time for entry into the semaphore’s critical section. Please ignore this possibility.)*

The operation `up()` (or `put()`, `give()` or `V()`) operation always adds 1 the count.

(9) The implementation of `up()` makes the thread calling `up()`:

- **× sometimes** but **not always** sleep (or block or wait).
- **always** sleep (or block or wait).
- **never** sleep (or block or wait*) during the call.

(*It might busy-wait for a very short time for entry into the semaphore’s critical section. Please ignore this possibility.)*

(10) In addition to the integer count variable (or field or member) each semaphore object (instance) has two other components. What are they? (Hint: The semaphore’s critical section must be protected and the kid returning a toy must know which classmates to wake up.)

- [ ] some kind of lock / barrier
- [ ] wait queue
- [ ] `up()`, `down()`
Part PC

Suppose a producer and a consumer share the same bounded buffer, and the bounded buffer is implemented with semaphores.

(11) The producer blocks (or waits or sleeps) when:

- Buffer is full.
- Buffer is empty.

(12) The consumer blocks (or waits or sleeps) when:

- Buffer is full.
- Buffer is empty.

(13) Describe what some other producer or consumer thread is doing in the situation when say the producer is blocked (or waiting or sleeping) even though the buffer has both a lot of items and a lot of empty slots. Hint: A critical section needs to be protected.

Part S

(14) The 3 circles below represent the 3 basic states of a thread: Blocked (or sleeping or waiting), Ready (or runnable), and Running. The 4 arrows represent things or situations that cause the OS to change the state of the thread.

Write the names (Blocked, Ready, Running) into the circles and label the arrows with A, B, C and D to answer which thing causes which state change.

A: Scheduler pre-empts this thread (and picks another).

B. Scheduler picks this thread (after pre-empting another thread or another thread blocks).

C: This thread blocks, say for input.

D: What this thread is blocked on occurs, say its input becomes available.

(15) We have seen that the Linux thread_union (containing the in-kernel stack) and thread_struct (embedded in the task_struct) have storage areas for the IP (program counter) and the other CPU registers.

Mark CLEARLY which statement(s) are (or is) true:

- These saved register values are up-to-date when the thread is running in user mode, actively using one of the CPU cores.
- These saved register values are up-to-date when the thread is in the ready state, after being pre-empted.
- These saved register values are up-to-date when the thread is in the blocked, waiting or sleeping state.
Part P

You may either answer the two questions below separately, or include your answer to the first within your answer to the second.

(16) Suppose once every 7 days you have an hour (60 minute) dinner at a nice restaurant but for the other 6 days you dine on pizza or other fast food for 15 minutes. Write how to calculate the average time per day you take for dinner, averaged over a long period.

(17) Relate this problem to the famous formula that people used to solve problems like 10 (to find the average time to access a memory word) in Tanenbaum’s Ch.1:

\[
\text{Avg. time} = \text{(Hit Rate)} \times \text{(Hit Time)} + (1 - \text{(Hit Rate)}) \times ((\text{Hit Time}) + \text{(Miss Penalty)})
\]

Specifically, what values are used for (Hit Rate), (Hit Time) and (Hit Time)+ (Miss Penalty) when you apply this formula to compute you average dinner eating time.
Part M

If you did the Racing Threads lab. homework properly, your archive would NOT contain a file named raceExperiment. Please assume this. Also assume your Makefile is correct. The raceExperiment half would be correct if you simply added to my Makefile the following 2 lines:

```make
raceExperiment: raceDriver
<invisible tab> time ./raceDriver 10
```

Consider all the compiling and linking needed for raceDriver to be one step. (18) Suppose the raceDriver executable file has NOT been built (compiled and linked using the gcc command) or is not up to date compared with its sources. Describe (briefly, three or four words for each step) everything that should happen when Tom types “make raceExperiment”.

(19) Now suppose the raceDriver executable file IS up-to-date. Describe (briefly, three or four words for each step) everything that should happen when Tom types “make raceExperiment”.

---
PART LSD: This is actual Linux source code from the down() function in semaphore.c:

```c
spin_lock_irqsave(&sem->lock, flags);
if (likely(sem->count > 0))
    sem->count--;
else
    __down(sem);
spin_unlock_irqrestore(&sem->lock, flags);
return;
```

and here is how the fast path (case of count>0) was assembled, for a 64-bit machine:

```assembly
    callq  _spin_lock_irqsave
111:  mov  0x4(%rbx),%edx /*Copy sem->count from shared memory to %edx*/
114:  test  %edx,%edx
116:  je   130 <down+0x30> /*Jump to else __down(sem) if count==0*/
118:  sub  $0x1,%edx
11b:  mov  %edx,0x4(%rbx) /*Copy our decremented value back to sem->count*/
11e:  mov  %rbx,%rdi /*prepare an argument for spin_unlock*/
121:  mov  %rax,%rsi /*prepare another arg for spin_unlock*/
124:  callq  _spin_unlock_irqrestore
130:  /*code to make down() return would be here.*/
```

(The numbers with colons like 111: are the addresses of the instructions relative to the beginning of the code assembled from semaphore.c)

There is a good reason the critical section (below) in the fast path case was protected by the spin lock!

```c
if(sem->count > 0) /*TRUE, > 0 in the fast path case*/
    sem->count--;
```

(20) Demonstrate a FAILURE that can (Think WILL!) occur IF THEY LEFT OUT THE SPIN LOCK PROTECTION. Start with the shared sem->count containing 1 and with two concurrent kernel control paths (threads) A and B.

1. A copies sem->count into its edx register.
2. Then A is interrupted.
3. B is scheduled.
4. B copies sem->count into its (different!) edx register.

You figure out the rest of this scenario which leads to BOTH A and B returning from down(), both thinking they have “taken” the single available “toy from the toybox”.

Demonstrate your understanding of this race failure and sequence diagrams by drawing a sequence diagram showing exactly which instructions run by A interleave with which instructions run by B. Then write a comment on the diagram that indicates exactly WHEN the failure is manifest.

You should identify the instructions by labels 111: 114: 116: 118: 11b: 130:, skipping of course the code related to the spin lock!
Finish my sequence diagram below:
Part LSL

We have seen how a Linux spin lock is used to implement mutual exclusion protection of the Linux code that implements semaphores. Assume a multi-core (multiple CPU) system. When the lock() operation is done on a spin lock, a shared lock variable (in memory) is accessed and set using an atomic instruction. That instruction is within a busy wait loop that loops until the accessed value is found to represent the unlocked state. This accessed value is the value that was originally in shared memory at the time the atomic instruction began to execute. The spin lock is needed because some kernel control paths cannot use a sleeping mutex or semaphore because those control paths must not put themselves to sleep. 

(21) Why is it better to use a sleeping mutex or semaphore instead of a busy-wait loop when you do have that choice?

(22) The atomic instruction (like Tanenbaum’s TSL) copies the value accessed from the shared variable into a register. When that value is found to represent the unlocked state, explain briefly how some system feature (hardware or software, please specify) guarantees that any other core that accesses that shared variable will find it locked (until after the locker unlocks it).

On a multi-core system, the Linux spin lock BOTH disables interrupts and uses an atomic instruction to test and set a shared lock variable in a busy-wait loop. 

(23) Disabling interrupts on the core taking the lock will prevent race condition failures due to racing against

☐ the same core only.
☐ the other cores only.
☐ both the same core and the other cores.