4 marks 1.a) What is the difference between an exception and an interrupt? Which is needed to implement a preemptive scheduler (and why)?

Exception: caused by something the program did. Interrupt: caused by something else going on inside/around the machine.

Interrupts: need to get control independent of what program is doing.

4 marks b) Two of the registers (k0 and k1) in the MIPS R3000 are reserved for use by the kernel in OS/161. Why is this necessary (or at least extremely helpful) in implementing the kernel? What would happen if a user program did make use of these registers?

Registers are needed to hold addresses, temporary results, etc., so it's hard to write code that doesn't modify one or more registers. Having two that can be used without first saving them makes it possible to write some reasonable code for the very beginning of an interrupt/exception handler.

If a program used k0 and/or k1, much of the time all would be fine, but occasionally the values in those registers would change "spontaneously" (when there was an interrupt or exception), causing the code to do unpredictable things.

4 marks c) Does the MIPS R3000 have a hardware-loaded or software-loaded TLB? Which kind of TLB gives an operating system more flexibility in designing the data structures supporting virtual address spaces (and why)?

Software-loaded TLB.

Software-loaded is more flexible. If hardware-loaded, the O/S sets up page tables and the hardware uses them during translation, meaning their format is largely or fully dictated by hardware. If software-loaded, make up whatever you want, since only kernel code will be using them.

2. Two special hardware instructions (swap and test-and-set) were mentioned in class as useful in providing simple code for the critical-section problem. Consider an alternative instruction, test-and-decrement, which tests the value in a storage location and decrements it (by 1) only if the value is positive. No access to the storage location is allowed between the test and the decrement. A library routine has been created allowing C-language programs to execute the instruction: `int TestAndDecrement(int X)`. The effect is to test the value of X and if it is positive, decrement it, returning 1 if the decrement took place, 0 otherwise.

2 marks

a) What property of TestAndDecrement allows a solution to the critical-section problem significantly simpler than Peterson's algorithm?

It is an atomic operation containing both a test and a modification of a stored value.

4 marks

b) Implement locks using TestAndDecrement(). Specify the values used to represent the locked and unlocked conditions and give the code for the lock and unlock operations.

lock variable = 1, unlocked; = 0, locked

```
lock(X):
   while (TestAndDecrement(X) == 0) ;

unlock(X):
   X = 1;
```

3 marks

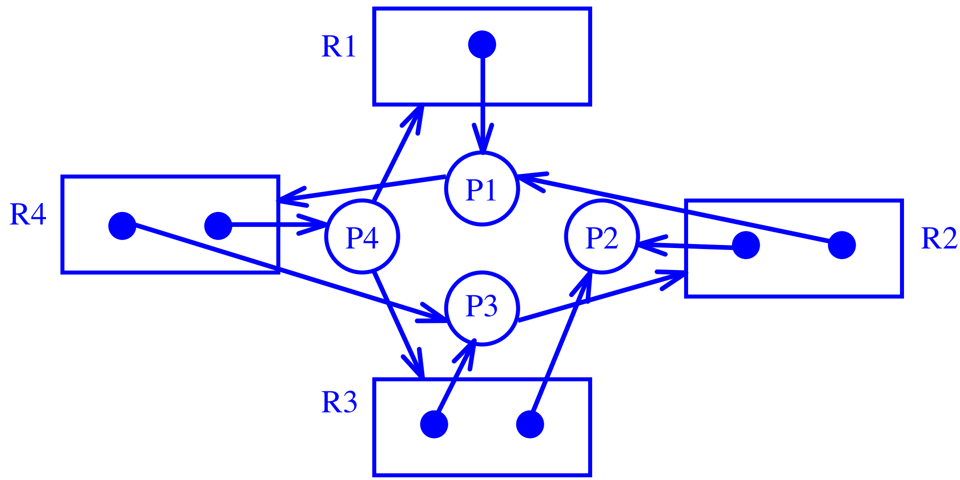
c) The TestAndDecrement instruction closely resembles the P() operation on a general semaphore. Could you use TestAndDecrement to provide a direct implementation of general semaphores, i.e., not using it simply to implement locks protecting the critical sections in the code for P() and V()? Why or why not?

No: TestAndDecrement does indeed work well for P(), but you can't do V(). Incrementing the value will be something like load value into register, add one to register, store register, which isn't atomic. If TestAndDecrement looks at/modifies the value in the middle of this sequence, Bad Things Will Happen™.

3. Suppose you are given the following demand (Di), allocated (Ai) and unallocated (U) resource vectors:

   A1 = (1,1,0,0) D1 = (0,0,0,1)
   A2 = (0,1,1,0) D2 = (0,0,0,0)
   A3 = (0,0,1,1) D3 = (0,1,0,0)
   A4 = (0,0,0,1) D4 = (1,0,1,0)
   U = (0,0,0,0)

4 marks a) Draw the resource-allocation graph defined by the given vectors.



6 marks b) Use the deadlock-detection algorithm discussed in class to determine if the threads are in a state of deadlock. Show your work, i.e., indicate the major decisions in the algorithm and the updates to its important state information. If the threads are deadlocked, report the threads and resources that are the cause of the deadlock.

R = U = (0, 0, 0, 0)
D2 <= R, so flag process 2, add A2 to R
R = R + A2 = (0, 1, 1, 0)
D3 <= R, so flag process 3, add A3 to R
R = R + A3 = (0, 1, 2, 1)
D1 <= R, so flag process 1, add A1 to R
R = R + A1 = (1, 2, 2, 1)
D4 <= R, so flag process 4, add A4 to R
R = R + A4 = (1, 2, 2, 2): All done, so no deadlock

2 marks c) The deadlock-detection algorithm used above essentially simulates a possible future execution of the system described by the set of vectors. It contains two key components, a test on the current state of a thread and a state update if that test is satisfied. Describe what each of these means, in terms of the threads and resources, rather than the values of variables manipulated by the algorithm.

The test (element-wise <= between Di and R) determines whether the currently available resources are sufficient to meet the current demand for process i.

The update simulates the completion and release of all resources by process i (by adding Ai into R).

2 marks d) In some cases, finding a cycle in the resource-allocation graph is sufficient to demonstrate the existence of a deadlock. Why is the more-complicated deadlock-detection algorithm required for the situation described in this question?

Cycle detection is sufficient only if there is a single unit of each resources. In this question, three of the four resources have multiple units, so a more-complicated algorithm is required.

4. An ingenious hardware designer has noted that a process generally uses its virtual address space from the two ends: starting at address 0, the code and statically allocated data, followed by the heap, which is potentially growing toward higher addresses; starting at the maximum address, the stack, potentially growing toward lower addresses. The unused part of the virtual address space is the (usually very large) “hole” between the end of the heap and the lowest address allocated to the stack.

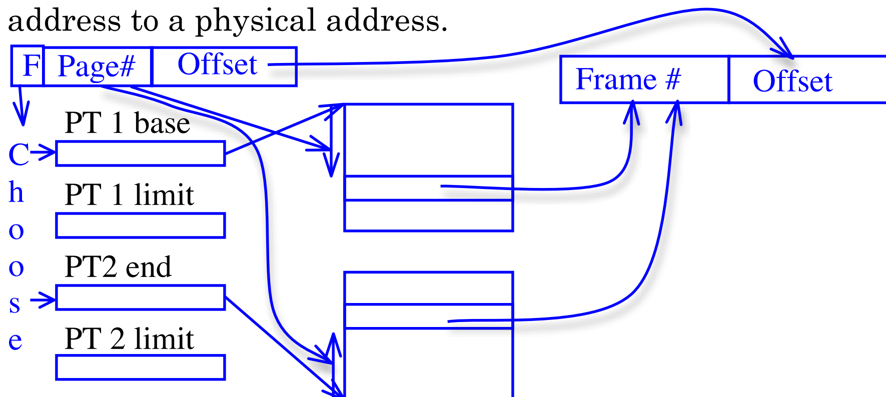
   This observation led to a design that avoids allocating page-table space for the unused part of the address space by using two page tables, one for each end of the address space. Specifically, addresses beginning with a 0 bit use the first table and addresses beginning with a 1 bit use the second table.

   Assume that addresses are 32 bits (including the bit just described), pages are 16K bytes (16384 bytes), and page-table entries are 4 bytes.

3 marks a) Describe the structure of a virtual address (as viewed by the MMU): how many sections it is divided into, how many bits in each section, and the purpose of each section.

1 bit for flag; 17 bits for page number; 14 bits for offset.

8 marks b) Assuming that address translation is done in hardware, draw a diagram of the registers and physical-storage tables required, showing the actions required for address translation. Also provide a description in words of the sequence of actions performed by the hardware to translate a virtual address to a physical address.



pick PT1 or PT2 based on flag; check page number against limit register, index into page table (forward for PT1, backward for PT2), check valid bit in entry, concatenate frame number from entry with offset from address.

2 marks c) From an application-programmer's perspective what disadvantage(s) does this design have relative to a scheme with a single page table for the entire virtual-address space?

The design divides the 4GB address space into two 2GB sections, one for the stack and one for everything else. The stack doesn't need nearly that much space, so much of the 4GB is essentially unavailable to the program.

2 marks d) What are the added complications for the operating-system kernel in managing memory using this design?

The page tables need to be contiguous and will potentially grow, so you have all the complications of variable-size space allocation (external-fragmentation, etc.) in handling the page tables.