Flush TLB:use thai code any time when perform a context switch since the mapping of the page of two processes would not be same. 1. Explain the advantages of abstracting a memory map as a set of contiguous regions rather than relying directly on the page tables for descriptive information. Hint: don't forget one the original goals of using virtual memory, in addition to any other advantages. A.Less overhead when accessing memory - don't have to go through computations of paging .Less external fragmentation overall and easy to implement 2. Describe the advantages and disadvantages of dynamically allocating objects for a device driver using slab cache relative to allocation using the kmalloc interface. A.Advantage: faster initialization of data because data stored in cache are of the same type/struct. Don't need to free and reallocate like in kmalloc only mark as not present.Disadvantage: a lot of internal space would go unused if the data doesn't use the same type. 3. Threads share a single address space, which in Linux is represented as a memory map. Explain how Linux supports sharing of memory maps between threads.   Two threads's mm_struct* field will point to the same mm_struct if the clone_mm bit is set when "fork". 4. Why does a Linux memory map maintain two data structures (a linked list and a red-black tree) for keeping track of virtual memory regions? A.Linked list if need to traverse all regions, red black for finding specific region including a specific address.11. Describe two ways in which a signal handler stack frame can be torn down (that is, removed) from the stack. A.One is using the sigreturn system call handler. The other can be done by the signal handler with a call to something like longjmp.

14. What events could cause an OS scheduler to switch the system to executing a new task? A.An interrupt could cause a scheduler to switch particularly IRQ0 better known as the PIT. Moreover, if the currently running process calls 'schedule()', it will yield to the processor, allowing another task to be scheduled. 15. What should happen when the interrupt handler for IRQ0 causes a context switch from task A to task B and returns? What if task A was already interrupted by IRQ6 when IRQ0 arrives and causes a context switch? A.When task A is resumed, the current task status should be maintained. If IRQ6 had interrupted, it must continue the interrupt from where it left off.打断会被打断打断。

18. Using the x86 ISA as an example, explain the concept of a continuation and how it relates to the state of the machine. A continuation is a representation of the control flow of your program at any point in time, essentially the stack19. Why are the file structures used in the kernel not directly accessible to the program using the file? A.1.To provide a layer of abstraction between the user and all of the files in the system. The user sees only one filesystem, while the OS sees numerous file systems for different devices 2.So the user cannot modify the file structure in any way that the OS doesn't want it to.20. How does the Linux kernel track the number of uses of a file structure (or memory map)? Why does it do so? A.Uses an atomic counter to keep track of the number of uses. If the file structure is shared between two tasks, one of them might close the file while the other might not. To make sure that the file structure doesn't actually close, we maintain a atomic_t f_count in the file structure. The release call is made when all instances of the file are closed.21. Explain the purpose of major and minor numbers for device drivers. A.The major number explains the type of device (cd rom, keyboard, etc.) and minor helps keep track of the number of instances of said device 22. How are a device's numbers (major and/or minor) used to allow a program to interact with a device? A.It allows the program to get the corresponding FOPS table to get the operations for the particular device. System call 调用 fops 里的 function23. You are charged with writing a driver for a display consisting of an LED array with 20 rows of 300 LEDs. Assuming that the primary role of this device is to support scrolling text, outline a good set of operations to make the device easy to use. Specify the system calls to be used as well as the data to be sent to/from the kernel for each operation. A.Sys_write - send an array of 300 x 20 bytes to set the state of the LED. Each byte indicates whether a particular LED must be on or off. Sys_read - read state of leds IOCTLS: shift_left/scroll_left(char* buf) - send 20 bytes (1 column) of LED data. Shift all the columns to the left by one and insert input column to the right. shift/scroll_right(char* buf) - same as left but right. Clear - sys_write(zeroes)

24. Explain the pros and cons of providing a simple kernel abstraction for a device (or even direct access to the device) rather than a complex abstraction (such as a network protocol stack). A.There is less overhead from the OS to have to deal with these abstractions where with a complex abstraction it involves more on the part of the OS.

25. Why is it acceptable for most device drivers to skip implementation of the readv and writev file operations? Do such calls simply return failure if a driver does not implement them? If not, how are they handled? Writev: write to several regions of a file (gather) (shouldn't this be gather?)Readv: read from several regions of a file (scatter) (and this scatter?)It does not fail, there's default implementation, (emulated if pointer is null) 26. Can a program executing in a system call keep its semaphores and other locks when it sleeps on a wait queue? If not, what implications does this fact have for the design of the data structures used to check for wakeup? No. This has the implication that anything checking on whether or not to wake the program must also consider whether any of the relinquished semaphores are now free.28. The strategy adopted for freeing user data when an active Imail user is deleted can be viewed as a form of lazy deallocation. Another approach to solving this problem is to put the user data into an 'almost free' list, to use a counter of active uses of the structure by the driver functions, and to free fully released user data structures from the 'almost free' list periodically or in response to some other event, such as adding a new user. The latter approach allows us to solve a problem that the transfer of responsibility could not solve (forcing Prof. Lumetta to change his specification). What is that problem? The problem of reauthenticating after deleting a user, since now we can have multiple users using the same file.

29. Why are support functions for a device potentially more complex than those used in most user-level code? You have to worry about synchronization since multiple users will probably using the device simultaneously30. Explain the testing methodology used with Imail.

Big security vibes31. Define a fixed-size data structure to represent an element of a tree with arbitrary node degree, in other words, a tree in which a given node may have an unbounded number of children nodes. What if each node had a pointer to it's child node and it's next sister node. So for each node, all its children nodes are in a linked list. Struct node{node* child;node* next_sister;}32. How are unaligned memory accesses caught and handled by an operating system running on an ISA that does not allow such things? Unaligned memory accesses occur when you try to read N bytes of data starting from an address that is not evenly divisible by N (i.e. addr % N != 0).

For example, reading 4 bytes of data from address 0x10004 is fine, but reading 4 bytes of data from address 0x10005 would be an unaligned memory Access."

34. As part of a particular system call, a user program passes a pointer to a region of memory along with the size of the region. Explain why the OS must check the validity of the region and how it can do so using x86 hardware support, OS abstractions, or a combination of the two. Note that there are several possible methods—explain the tradeoffs amongst them. Important to control access rights on a computer - prevents a process from accessing memory that hasn't been allocated to it. Prevents a bug or malware within a process. Segmentation - dividing computer memory into segments Paged virtual memory - maps virtual memory to physical memory Protection keys - divides physical memory into blocks of certain size, each with a numerical value called protection key 35. As you saw with I-mail, locks are often associated with dynamically-allocated data structures in a kernel. Why is it hard to acquire two such locks for a critical section? What can be done to solve the difficulty?user data semaphores are not ordered with respect to one another. As a result, no code should ever acquire more than one user data semaphore. Order by memory address of lock, or by user list. 36. Why are some major numbers further subdivided into distinct devices? There are 256 major numbers available (8 bit). What if we need for than 256 devices. Solution: connect multiple devices to one #, #10 in linux .37. Explain how the file operations structure is used in a typical operating system. Contains function pointers to different operations (open, close, read, write) corresponding to a file type. A pointer to an fopes structure is stored in each (in use) fdt entry of a process. One file operation structure per device type.38. Explain how the value returned from a program to the OS is then delivered back to the execute system call that started the program in the MP3 operating system. A.The value will be in eax insured by the halt. And therefore when it is returned to the parent, the return value is what we want it to be. 39. Explain what must be done to switch between terminals in MP3: what state must be copied, what pointers/registers must be changed (and how), and what processor state must be flushed (and why). A.The state of the vidmem must be copied to the buffer of the current terminal and the buffer state of the new terminal must be copied to vidmem.The paging must eh changed so that the process in the current terminal is printing to its buffer instead of the video mem. The TLB must be flushed since paging has been changed 40. Consider a single region of memory allocated using the buddy system. The region has a unique buddy region, but there are actually two adjacent regions of the same size in memory. Why doesn't the buddy system attempt to merge with either region instead of restricting itself to a single buddy? Because they are not buddy, like 4 and 6 can't be merged.This is to try and maintain the exponential size of the bins (as in make sure they are always in powers of two so they can be recursively merged to the highest size)41. How is the address of a memory region's buddy calculated in the buddy system? Address XOR with size   (4kb is x1000)(4kB is 0x1000) <- this 42. Why can buddy bit information not be stored directly in the memory regions themselves, as with most user-level memory allocation implementations?

The partially busy bit is changed with recursion. Can't use recursion in kernel.

==

ECE 391 Final, Fall 2012

Question 1: Short Answer (28 points) Please answer concisely. If you find yourself writing more than a sentence or two, your answer is probably wrong.

a) **(2) What is the difference between masking a signal and ignoring it?**

When a signal is masked, the signal will be marked as pending thereafter, so it will never be sent to that program again. Whereas, when a signal is ignored, it can keep being sent.

" Analogous to masked interrupts, a signal sent to a program but masked by the program is held by the operating system until the program unmasks the signal, at which point the signal is delivered immediately.
If the signal is being ignored, it is simply discarded " - Course note

b) **(3) In what cases is it useful to force a program to receive a particular signal, even though the program has asked that the signal be masked out?**

To deliver a signal generated by exception, since the processor cannot continue executing the program in that case.

c) **(4) When does a signal get generated? Check all that apply.**

During a kill system call, after a terminal event (CTRL-C) , When an interrupt occurs

When an exception occurs ?? yes

d) (2) When does a signal get delivered? Circle best option.

A. During a kill system call
B. After a timer interrupt
C. After any interrupt
D. When the receiving process gets scheduled

e) (4) Describe one advantage and one disadvantage of lazy algorithms, such as those used for demand paging.

Advantage: Avoid unnecessary work, e.g. in demand paging, no need to load pages into physical memory if those pages are never accessed

Disadvantage: Increased latency during execution, e.g. in demand paging, a program has to wait for any page to be loaded in before it can use it for the first time

(2) Explain what this code does:

```
movl %cr3,%eax
movl %eax,%cr3
```

This code flushes the TLB,

f) (3) When would you want to use the above code? Explain why.

You would want to use this code every time you perform a context switch since the mapping of the pages for two processes would not be the same.

Want to use this code any time any paging structure is changed

g) (4) When does the CPU set the accessed and dirty flags in the page table? And how are they used by the operating system?

Read or write for the access flag, on write for the dirty flag. Access flag is used to determine which pages to remove from physical memory when demand for memory is higher than supply, dirty flag to determine whether the physical memory needs to be written to disk or not before deletion.

h) (2) Which of these is typically the smallest?

A. Rotational latency
B. Seek latency
[✓ C. Transfer latency]

Seek Latency is moving the arm around the disk. Can be very slow.
Rotational is rotating the disk to find the location
Transfer is moving data to memory from disk. Fastest

i) (2) The use of block groups in EXT2 is primarily to reduce:

A. Rotational latency
B. Seek latency
C. Transfer latency
[✓ D. Internal fragmentation]

Block groups wouldn't necessarily affect where the data is stored, but it would decrease fragmentation between blocks of data.

Question 2: MP3 (11 points)

(5) Your MP3 partner implemented the following system call entry routine:

```
Syscall_enter:
Pushfl
Cli
decl %eax # eax=eax-1
cmpl $10,%eax
jae badcall
cmpl $0,%eax
jb badcall
pushl %ebp
pushl %edi
pushl %esi
pushl %edx
pushl %ecx
pushl %ebx
call *jump_table(,%eax,4)
popl %ebx
popl %ecx
popl %edx
popl %esi
popl %edi
popl %ebp
exit: popfl
Iret
Badcall:
movl $-1, %eax
jmp exit
```

You realize that some of the code here is redundant and can be eliminated. Explain what you would remove and why. (You may use the next page to continue your answer)

At the very least, the cmpl $0, %eax followed by the jb is useless since you can't be below 0 in an unsigned comparison

You also shouldn't have to save the caller saved registers since system calls are always expected.

Also you don't need to save ebp (cause we didn't save it for our mp3 and it worked)(not sure abt this. Can someone confirm?)

Couldn't you also get rid of the dec %eax by just making the syscalls 0 indexed to begin with an have users pass in values starting at 0 no i don't think we can modify user behaviour. (agreed with blue)

b) (2) What's the best way to avoid synchronization conflicts that may arise when accessing the MP3 file system?

Any ideas here? I thought the MP3 file system is read-only so there's no need to enforce synchronization ← agreed

c) (4) To implement terminal switching, your partner created a separate buffer for each terminal. When switching terminals, the contents of the screen gets copied into the previous terminal's buffer, and the contents of the new terminal's buffer gets copied onto the screen. To make this work, you also need to update process page tables of the process during a terminal switch. Explain what you need to change and why. Be detailed.

Need to change the video memory in the pages so that the process currently running would print to the correct place in memory until the scheduler changes the paging. (Does this make sense?)

Question 3: Synchronization

(14 points) Below is an implementation of synchronization using wait queues.

```
char pipe_buf[4096];
unsigned int head, size;
DECLARE_MUTEX(pipe_lock);
DECLARE_WAIT_QUEUE_HEAD(readQ);
DECLARE_WAIT_QUEUE_HEAD(writeQ);

void init_pipe(void) {
        down(&pipe_lock);
```

```
          head = 0; size = 0;
          up(&pipe_lock);
}
/* read single char */
void write(char c) {
          down(&pipe_lock);

          DECLARE_WAIT(wait);
          while (1) {
                    /* add self to wait queue */
                    prepare_to_wait(&writeQ, &wait, TASK_UNINTERRUPTIBLE);
                    if (size < 4096)
                              Break;
                     /* go to sleep (potentially) */
                    schedule();
          }
/* remove self from queue */
finish_wait(&writeQ, &wait);
pipe_buf[(head + size) % 4096] = c;
size++;
wake_up(&readQ);
/* wake up one sleeping reader */
up(&pipe_lock); }
char read(void) {
          char c;
                    DECLARE_WAIT(wait);
                    down(&pipe_lock);

                    while (1) {
                              /* add self to wait queue */
                              prepare_to_wait(&readQ, &wait, TASK_UNINTERRUPTIBLE);
                              if (size > 0)
                                        break;
                              /* go to sleep (potentially) */
                              schedule();
}
          /* remove self from queue */
          finish_wait(&readQ, &wait);
          c = pipe_buf[head];
          head = (head + 1) % 4096; size--;
```

```
        wake_up(&writeQ);
        /* wake up one sleeping writer */
        up(&pipe_lock); return c;
}
```

Pork belly
Beef fried rice
Large sesame chicken

a) (3) There is a critical mistake in how the wait queues are used. What is it?
   Is it that the reader/writer should not go to sleep in the wait queue while holding the semaphore? Otherwise it can't possibly be woken up, since it's a mutex.
b) (2) Why is a while(1) loop used? Explain how a writer could end up going to sleep multiple times.
   In the write function, first the program checks if the buffer is full or not. If it is full, the program will yield the process and will go to the beginning of the loop the next time it is scheduled. If the buffer is still full, it will go to sleep again. Only when read is called, the size will decrease and the buffer will have space. Similarly, read will yield and go to sleep if the buffer is empty (size = 0).
c) (3) wake_up wakes up a single waiting task (i.e., it is similar to notify). What would happen if wake_up_all (equivalent to broadcast) were used instead?
   If multiple write calls are waiting, and there is only one free spot in the buffer, all of them will try to write to it.

   All of them will try to down the pipe lock and all but one will fail.
d) (6) Re-implement the scheme using two semaphores. Complete the code below. (d)

```
char pipe_buf[4096];
int head, size;
DECLARE_MUTEX(pipe_lock);
struct semaphore readsem, writesem;
void init() {
        down(&pipe_lock);
        head = 0; size = 0;
        sema_init(&readsem, 0);
        sema_init(&writesem, 4096);
        up(&pipe_lock);
```

```
}

void write(char c) {
down(&writesem);
down(&pipe_lock);
pipe_buf[(head + size) % 4096] = c;
size++;
up(&pipe_lock);
up(&readsem);
```

Why did this down a write sem then up a read sem?

```
} char read(void) {
// YOUR CODE HERE
char c;
down(&readsem);
down(&pipe_lock);
c = pipe_buf[head];
head = (head + 1) % 4096;
size—;
up(&pipe_lock);
up(&writesem);
return c;
}
```

Question 4: Scheduling

a)(21 points) Consider the following set of processes:

Process	Arrival time	Run time
A	0	4
B	2	3
C	4	5
D	6	1

a) (8) Show the scheduling order of these processes under the following schedulers:

- **First-in First-out (FIFO)**
- **Shortest remaining time first (SRT)**
- **Round-robin with a quantum of 2 time units (RR)**

Assume that the context switch overhead is 0 and, all else being equal, new processes should be scheduled first**. A process arriving at time t can first be scheduled for slot t**

Time Slot	Fifo	SRT	RR
0	A	A	A A
1	A	A	A A
2	A	A	B B
3	A	A	B B
4	B	B	C A
5	B	B	C A
6	B	~~B~~ D	D C
7	C	~~D~~ B	A C
8	C	C	A B
9	C	C	B D
10	C	C	C C
11	C	C	C C
12	D	C	C C
13			
14			
15			
16			
17			
18			

(4) Calculate the turnaround time of each process, as well as the average.

Scheduler	A	B	C	D	Average
FIFO					/4
SRT					/4
RR					/4

I believe turnaround time includes the waiting time in the beginning

Turnaround time must be time at which process completed - arrival time of process

Scheduler	A	B	C	D	Average
FIFO	4 3	3 5 4	5 8 7	17 6	13 24 20/4
SRT	4 3	3 4 5 4	5 9 8	1 2 1	13 /4 14/4
RR	9 8	8 7	9 8	1 0	27 23 /4

c) (6) For each scheduler, explain one advantage of using it.

i. FIFO - Simple Implementation, Low overhead from context switch, no starvation every process guaranteed to terminate

ii. SRT - Ensures Jobs with Shorter Runtime get run first

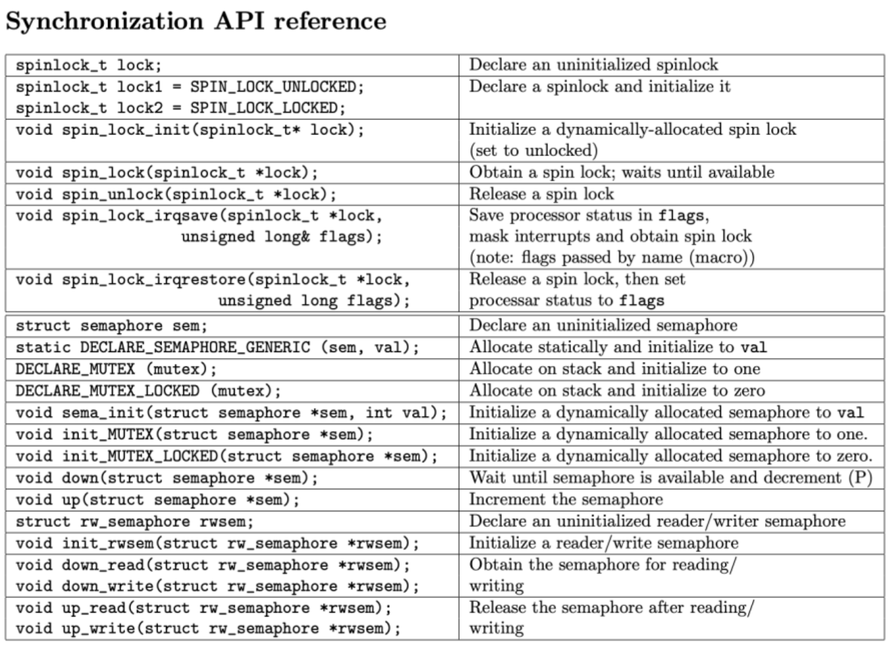
Optimal w.r.t average wait time of a process

iii. RR - Ensure some notions of fairness

d) (3) Explain the tradeoff in selecting the round-robin quantum size. In particular, what benefit do you get from increasing or decreasing the quantum?

A shorter quantum size allows the processor cycle through each process quicker and more accurately fit the illusion of a multiprocessor system. However, it does increase the impact of the overhead from a context switch. By increasing the quantum size, you decrease the impact of the overhead from a context switch but decrease the amount of time the

processor gets to cycle through the processor. This impacts the virtualization of the processor.



Synchronization API reference

`spinlock_t lock;`	Declare an uninitialized spinlock

`spinlock_t lock1 = SPIN_LOCK_UNLOCKED;`
`spinlock_t lock2 = SPIN_LOCK_LOCKED;`
	Declare a spinlock and initialize it

`void spin_lock_init(spinlock_t* lock);`	Initialize a dynamically-allocated spin lock (set to unlocked)
`void spin_lock(spinlock_t *lock);`	Obtain a spin lock; waits until available
`void spin_unlock(spinlock_t *lock);`	Release a spin lock
`void spin_lock_irqsave(spinlock_t *lock, unsigned long& flags);`	Save processor status in **flags**, mask interrupts and obtain spin lock (note: flags passed by name (macro))
`void spin_lock_irqrestore(spinlock_t *lock, unsigned long flags);`	Release a spin lock, then set processar status to **flags**
`struct semaphore sem;`	Declare an uninitialized semaphore
`static DECLARE_SEMAPHORE_GENERIC (sem, val);`	Allocate statically and initialize to `val`
`DECLARE_MUTEX (mutex);`	Allocate on stack and initialize to one
`DECLARE_MUTEX_LOCKED (mutex);`	Allocate on stack and initialize to zero
`void sema_init(struct semaphore *sem, int val);`	Initialize a dynamically allocated semaphore to `val`
`void init_MUTEX(struct semaphore *sem);`	Initialize a dynamically allocated semaphore to one.
`void init_MUTEX_LOCKED(struct semaphore *sem);`	Initialize a dynamically allocated semaphore to zero.
`void down(struct semaphore *sem);`	Wait until semaphore is available and decrement (P)
`void up(struct semaphore *sem);`	Increment the semaphore
`struct rw_semaphore rwsem;`	Declare an uninitialized reader/writer semaphore
`void init_rwsem(struct rw_semaphore *rwsem);`	Initialize a reader/write semaphore

`void down_read(struct rw_semaphore *rwsem);`
`void down_write(struct rw_semaphore *rwsem);`
	Obtain the semaphore for reading/
	writing

`void up_read(struct rw_semaphore *rwsem);`
`void up_write(struct rw_semaphore *rwsem);`
	Release the semaphore after reading/
	writing

Fall 2011

Answer the following questions concisely. You should not need to write more than a sentence or two; if you are writing more, it is probably wrong.

Part A (5 points): Certain types of device drivers, such as those typically used with graphics cards, provide applications with direct access to device memory. In other words, memory that physically resides on the device is mapped directly into an application's virtual address space.

Explain the primary advantage of this direct mapping relative to restricting access to device memory to the kernel (and to user programs via system calls).

Quicker 太简洁了...

要清楚 device driver 是什么

PartB (5points): Device drivers can execute in ring0(kernel level) on an x86 CPU, and many operating systems use this model. Explain the primary disadvantage of this approach relative to requiring device drivers to execute with less privilege.

If device drivers are allowed to operate at ring 0, it allows them full access to hardware and memory. This allows for things to crash and be unrecoverable if device operations aren't handled correctly? (Possibly insufficient answer)

Part C (5 points): Ben Bitdiddle just finished writing his code for MP3 checkpoint ~~4~~ 5. However, when he types fish in his first shell, he notices that the program executes randomly in one of the three shells. That is, sometimes it runs in the first shell, sometimes in the second, and sometimes in the third. He launches GDB to debug and confirms that he indeed typed fish after switching to the first shell, but other shells do in fact sometimes launch the job when he hits ENTER.

In relation to the round-robin scheduler used with the MP, what is the most likely cause of the problem, and what should he do to fix his bug?

? 这跟 scheduler 有啥关系

He most likely did not map Vid Memory correctly or page correctly as fish always writes to the active process instead of the terminal's process it was started in.

Problem 2 (20 points): System Calls

叫包装函数
wrapper function ☆

Part A (5 points): Why is assembly linkage necessary for system calls? In other words, why not simply create a C function, *e.g.*, `system_call`, and fill an IDT (interrupt descriptor table) vector with the address of that function?

Can't pass syscall # through stack, only through register, need assembly

应该是如果想对 stack 进行改变，就要用 assembly

Part B (5 points): ~~Below is the entry point~~ after calling `int $0x80`. Briefly explain all actions that your code should take between starting the system call and using the jump table.

```
int_80:
    # What do you need to do here?
    call system_call_jmp_table(,%eax,4)
    # Some other code that you need not write nor comment upon.
    iret
```

③ ① ②

Check for a valid system call number, store all the registers and flags, then push the arguments.

Mp3 里面写了 SysCall_Linkage.S

Save all registers

Push parameter to stack

Check for valid syscall # **(This implementation works but provides unnecessary code as if invalid number is passed, will need to pop all parameters and pop registers again.)**

这可是你写的！

Part C (5 points): Your friend has written a system call for `read`, but forgot something. What did they forget?

```
my_file* fd_array[8]; // array of file pointers for current task

int32_t sys_read (int32_t fd, uint8_t* buf, int32_t nbytes)
{
    if (NULL == fd_array[fd] || NULL == fd_array[fd]->fops ||
        NULL == fd_array[fd]->fops->read) {
        return -1;
    }
    return fd_array[fd]->fops->read (fd_array[fd], buf, nbytes);
}
```

flag 吧 if current_pcb → fd_array[fd].flags == 0

+ return -1;

Forgot to check if the passed in parameters to the function are valid or not. e.g if buf == NULL, fd is outside bounds

Forgot to check if the corresponding pcb is being used.

Pass a pointer to a list of these floating point numbers into the system call to reduce the number of variables inputted

Part D (5 points): Linux uses registers to pass arguments to system calls, thus the number of parameters is limited by the number of registers. Explain how you might accommodate a new system call that requires passing 42 double-precision floating-point values.

咋做？

***__Part D__ (5 points): Before a kernel executes a user-level signal handler, the kernel moves all architectural state (saved registers, flags, and so forth) from the kernel stack to the user stack. When the signal handler finishes executing, the state is checked for accuracy (*e.g.*, to prevent privileged execution of user code) and then copied back to the kernel stack.

This copying allows user-level programs to view and change their architectural state when a signal occurs (*e.g.*, to inspect the reason for an exception, or to switch to another user-level thread).

Why else must the kernel move the state to the user stack? *Hint: think in terms of malicious programs.*

他上课讲了.

Avoid kernel stack overflow

这玩意应该不考吧

Parts C and **D** pertain to the data block organization used in the `ext2` filesystem. As you may recall, an `ext2` inode contains 15 block pointers. The first 12 point directly to data blocks. The next—indirect—points to a data block filled with data block pointers. The next is doubly indirect, and the last is triply indirect.

Part C (5 points): Assuming 4-byte block indices and data blocks of size B, compute the total amount of data that can be stored at each level indirection in `ext2` in terms of B.

Direct:	Doubly Indirect:
Indirect:	Triply Indirect:

Part D (5 points): The FAT32 file system is organized as a singly-linked list of data blocks. The file allocation table (FAT) includes an array indexed by data block number and holding the index of the next data block in the same file (or 0 to indicate the last data block in the file).

Explain one disadvantage of the `ext2` approach relative to the FAT32 approach.

C: Direct: 12B

Doubly Indirect: (B^2)/4 <- Indirect? We divide by 4 as the first/original block must be divided into 4-byte block indices.

Indirect: (B^3)/16 <- Doubly Indirect? We divide by 4 twice over for the same reason above.

Triply Indirect: (B^4)/32

Triply Indirect: (B^4)/64

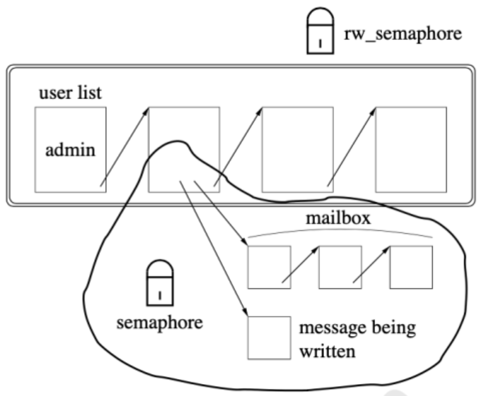
Also this assumes that B is the size of data blocks in bytes and not in the 4-byte indices.

D: Faster to get to an index in an array than walking a linked list

That’s an advantage of the ext2 approach, no? ext2 is generally more complex to implement.

Problem 3, continued:

这也是跟 semaphore相关的



Parts C through **E** make use of the figure to the right, which shows the locking strategy used with I-mail. The user list and authentication data are protected by a reader-writer semaphore, and the remainder of each user data structure is protected by a semaphore in the structure itself. The lock order specified for I-mail specifies only that the user list semaphore must be acquired before any user data semaphore.

Part C (5 points): Explain how a deadlock might occur if, for example, message delivery acquires both users’ user data semaphores in order to move the message from the sender’s “message being written” to the recipient’s “mailbox.”

If user A attempts to write a message to user B at the same time that user B is writing a message to user A, when both of them attempt to send their respective messages, they will be forever waiting for the other user to relinquish their lock.

Part D (5 points): Consider the problem of putting a task to sleep while it waits for a message to arrive to an I-mail "mailbox." What locks, if any, should be held while the task sleeps? Justify your answer.

No locks should be held while the task sleeps. Acquiring the user data semaphore would prevent any mail from being delivered to the user's mailbox, and obtaining either user list semaphore is functionally useless and will starve any reader/writer.

***__Part E__ (5 points): A friend wants to extend I-mail by adding a method to sleep until a message arrives from a particular user. The method of course needs to walk over the list of messages in the "mailbox" to check for an appropriate message before going to sleep, and again when waking up. Explain why this operation can not be performed safely, despite the fact that all changes to the "mailbox" list are made in critical sections protected by the user data semaphore.

I-mail is designed to only have the first message in your mailbox viewable at one time - only allowing you to view the next one after deleting the frontrunner. This potential method would require significant reorganization of I-mail in order to work properly.

Another reason: Need to check conditions before going to sleep without holding the lock. It's not safe to walk a linked list without the lock.

Fa2009

Problem 1 (20 points): Short Answers

Answer the following questions concisely. You should should not need to write more than a sentence or two; if you are writing more, it is probably wrong.

Part A (5 points): Certain types of device drivers provide applications with direct access to device memory. In other words, memory that physically resides on the device is mapped directly into an application's virtual address space. The reason is generally to improve performance by reducing or eliminating the number of user-level-to-kernel transitions necessary to interact with the device, and the approach is standard for graphics cards and fairly common for high-speed networking.

What important role does such a device driver play, since—to use the example of the graphics card—the application delivers graphics content directly to the card using a virtual memory mapping?

Allows the kernel to keep track of the device's state?

Part B (5 points): What implication does the model discussed in **Part A** have for the design of the device itself? (*Hint: think about what must be done in a typical system call.*)

lack of system calls improves performance?

Part C (5 points): The last checkpoint of MP3 required that you implement three terminals. A friend of yours asks for help in understanding why their implementation is not working.

Using regular (non-video) memory, your friend has added a buffer to each terminal to hold the screen data for that terminal. The `vidmap` system call identifies the terminal associated with the program that made the call and returns a pointer to the corresponding buffer, allowing programs like `fish` to write directly into the buffer. When a non-control key is pressed (a letter, number, backspace, Enter, *etc.*), the terminal driver writes the character into the active terminal's buffer, then copies the buffer into video memory. To switch terminals (using Ctrl-F1/F2/F3), the video memory is first copied into the buffer for the currently active terminal, and then the newly active terminal's buffer is copied to video memory.

Explain how your friend's approach can fail. **You do not need to suggest a solution.**

If, say, fish was/is running in the newly active terminal, fish would not be writing to the video memory like it should now that it's in the active terminal.

Problem 2 (20 points): System Calls

Part A (10 points): You are charged with adding a new system call, `eat`, to an operating system that your company is developing.

The structures below define operations on a file and the contents of a file (only a couple of the operations are shown). For simplicity, this problem assumes a static array of eight file structures for a single task, also shown below.

```
struct file_operations {
    int32_t (*open) (int32_t fd, const char* name);
    ...
    int32_t (*eat) (int32_t fd, void* buf, uint32_t num_bytes);
};

struct file {
    uint32_t                flags;  // 1 if valid (in use), 0 if not
    uint32_t                type;   // 0 for file, 1 for special device
    struct file_operations* fops;   // operations table pointer
    uint32_t                inode;  // inode index (file only)
    uint32_t                f_pos;  // position in file (file only)
};

static struct file files[8];        // indexed by fd (file descriptor)
```

Your task is to write the main entry point for the new system call.

Notice that the `type` field in the file structure differentiates between two types of I/O channels. You can assume that the file operations table in a valid file has been set to point to the functions (including `eat`) for the specified type.

You also know that the `eat` system call routine for files makes use of all three of its arguments, while the routine for special devices makes use of only the `num_bytes` parameter. Both routines assume that they have been invoked on a valid file structure.

Write the code as simply as possible. Return -1 for any failure.

```
// main entry point for the eat system call
int32_t sys_eat (int32_t fd, void* buf, int32_t num_bytes)
{
    /* Your code goes here.  */
```

```
    if (files[fd].flags != 1) {
         return -1;
    } else if (files[fd].type == 0) {
         if (buf == NULL || num_bytes < 0) {
              return -1;
         }
         return files[fd].fops->eat(fd, buf, num_bytes);
    } else if (files[fd].type == 1) {
            if (num_bytes < 0) {
                 return -1;
            }
            Return files[fd].fops->eat(num_bytes);
       }
       return -1;
  }
```

Problem 2, continued:

Part B (5 points): In Linux' calling convention for system calls, all registers other than those holding the return value (EAX and possibly EDX) are callee-saved. That is, no system call changes any other register value. Is this design simply an arbitrary choice, or does something about the nature of system calls make callee-saved registers a more sensible approach than caller-saved? (If so, explain.)

Because system calls pass arguments through registers, making the systems calls callee saved generally means that a user-level program does not have to do extra work of its own accord in preserving its task state.

Part C (5 points): Explain the role played by a device's major number with regard to system calls.

all device files that have the same major number and the same type share the same set of file operations, because they are handled by the same device driver.

Problem 3 (15 points): Device Drivers

Part A (5 points): A friend suggests using one spin lock per location on an 80 × 25-character text screen so as to enable maximally parallel accesses by terminal I/O drivers, RTC- and timer-chip-interrupt debugging displays, and keyboard-driven terminal switches. Aside from the additional code to obtain a lock before modifying any individual location, what other reasons make such a design unattractive relative to using one lock for the whole screen?

This design effectively allows a “first-come-first-served” basis with regards to the screen - in other words, different programs may write to the screen at the same time, pixel by pixel.

Part B (5 points): As you may recall, the Linux macro that allows a task to go to sleep until a condition becomes true takes a C expression as one of its arguments:

```
int wait_event_interruptible (wait_queue_head_t* wq, <C expression> condition);
```

The condition may be evaluated many times, and the macro returns 0 if the condition has evaluated to true, or -ERESTARTSYS if a signal is received before the condition becomes true.

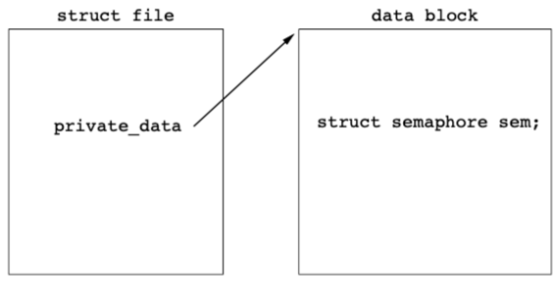
While reading a friend’s new device driver code, you see that your friend has made a call to `wait_event_interruptible` using a function call as the `condition` argument. Can you simply check the signature of the function (for example, that it returns an `int`), or do you need to read the function’s code? Explain your answer.

You should check if the function ever returns 0 as a condition. If you for whatever reason want the condition to never evaluate to true, you might as well give the condition 1 == 0.

Also need to check the side effects of the function, because it may be evaluated multiple times in the macro.

Part C (5 points): A certain Linux device driver associates a block of dynamically allocated data with an open file. One of the system calls is intended to remove the association, after which the block can be freed. Your friend has implemented this call with the following steps. Explain what can go wrong.

1. Acquire semaphore in data block to ensure atomicity.
2. Change file structure `private_data` pointer to NULL to prevent further references to the data block.
3. Free the data block.



In the case that the open file was copied over using a fork(), we don’t want to free it until all instances of the file are closed

不会做

Problem 4 (20 points): Signals

You decide to write a game similar to the maze game in MP2. The game uses MODE-X graphics, and the avatar is controlled by the Tux controller.

Part A (5 points): You do not bother to write signal handlers because your functionality does not require signals. What problem occurs in the event of an exception? How can handling signals solve this problem?

Part B (5 points): Your implementation uses several threads. One thread handles display of the clock. You arrange to have `SIGIO` signals sent to this thread using the traditional signal model whenever the real-time clock becomes readable (that is, on an RTC interrupt). The signal handler in the thread increments a local variable tracking elapsed time, then updates the time display on the status bar. Explain a potential problem with this approach.

Part C (5 points): You write a handler for `SIGINT`, then compile the code with no warnings or errors. You start GDB on the program and set a breakpoint inside your new signal handler, but when you run the program the and hit CTRL-C, the program terminates without stopping at the breakpoint. What is the most likely problem? In other words, what is the thing to check first in debugging?

GDB automatically skips over signals by default. Check to see if gdb is setup to debug through signal interrupts rather than wait until they are returned from.

Part D (5 points): Signals are the user-level analogue of interrupts. Explain one difference between the two (other than the user-level nature of signals).

Signals are generated via user level code or kernel, interrupts are generated by hardware

Problem 5 (25 points): File Systems

You must design a filesystem for a new green machine being developed to record fee payments in Jellystone National Park. The machine runs on solar power and is susceptible to frequent power loss. Your filesystem will use index nodes (inodes) and data blocks similar to those found in the `ext2` filesystem discussed in class.

This problem asks you to consider several design aspects for your filesystem.

Parts A through **C** concern the process of creating a new file, for which you must write both a new inode and one or more new data blocks.

Part A (5 points): What might happen if you choose to write the inode before the data block(s)?

If you write Inode before the data block, it might accidentally try to be accessed before the block is written thus unable to access the data that is there.

Part B (5 points): What might happen if you choose to write the data block(s) before the inode?

May be unable to access said data block without the inode

Part C (5 points): Which approach is better?

Parts D and **E** ask you to compare the data block organization used in the `ext2` filesystem with alternatives. As you may recall, an `ext2` inode contains 15 block pointers. The first 12 point directly to data blocks. The next—indirect—points to a data block filled with data block pointers. The next is doubly indirect, and the last is triply indirect.

Part D (5 points): Consider an alternative approach using a linked list of block pointers. In such a list, the first 14 block pointers are held in the inode. The last block pointer is then used as the head of a list of data blocks chained as a linked list but otherwise filled with direct block pointers. For example, a 4kB block would hold 1,023 block pointers and a pointer to the next block of 1,023.

What is an advantage of the `ext2` approach relative to the linked list approach?

Part E (5 points): Consider an alternative tree-like approach that uses 15 triply indirect data block pointers (and zero direct, indirect, and doubly indirect block pointers).

What is an advantage of the `ext2` system relative to the tree-like approach?